Thesis for the degree of Doctor of Philosophy

On the Fundamentals of Analysis and Detection of Computer Misuse

Ulf Lindqvist

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden 1999

On the Fundamentals of Analysis and Detection of Computer Misuse
Ulf Lindqvist
ISBN 91-7197-832-1

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 1530
ISSN 0346-718X

School of Electrical and Computer Engineering
Chalmers University of Technology, Göteborg, Sweden
Technical report 370
ISSN 0282-5406

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 10 00
www.ce.chalmers.se

The author can be reached at Ulf.Lindqvist@computer.org

Cover:
Some categories of security threats. Photo by the author.

Chalmers Reproservice
Göteborg, Sweden 1999

On the Fundamentals of Analysis and Detection of Computer Misuse

Ulf Lindqvist
Department of Computer Engineering
Chalmers University of Technology

Abstract

Most computerized information systems we use in our everyday lives provide very little protection against hostile manipulation. At the same time, there is a rapidly increasing dependence on services provided by these computer systems and networks, and security is thus not only an interesting and challenging research discipline but has indeed developed into a critical issue for society.

This thesis presents research focused on the fundamental technical issues of computer misuse, aimed at manual analysis and automatic detection. The objective is to analyze and understand the technical nature of security threats and, on the basis of this, develop efficient generic methods that can improve the security of existing and future systems. The work is performed from the perspective of system and information owners, a different approach compared to the many previous studies that focus on system developers only. The analysis is based mainly on empirical data from student experiments but also uses data from a security analysis, data recorded from a network server and data produced for an intrusion detection evaluation project. Throughout this work, systematic categorization of data has been used as the main method for data analysis.

The results of this work include new findings about the behavior of so-called insider attackers, a dangerous but sometimes neglected security threat. For systems that include commercial off-the-shelf components, underlying causes of system vulnerabilities are identified and discussed, a systematic procedure for vulnerability remediation is developed and a risk management strategy is proposed. Furthermore, the aspects of computer misuse that are fundamental for automatic detection are identified and analyzed in detail. The efficiency and usability of a generic expert system tool for automatic misuse detection is verified empirically. A general database format for documenting attack types and for automatically updating detection tools is outlined.

Keywords: computer security, network security, taxonomy, intrusion, vulnerability, risk, intrusion detection.

This page is intentionally left blank.

List of publications

This thesis is based on work reported in the following appended papers, referred to by capital letters in the text:

Part I: Analysis of the nature of security threats

A Ulf Lindqvist, Tomas Olovsson, and Erland Jonsson. An analysis of a secure system based on trusted components. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pp. 213–223, Gaithersburg, Maryland, June 17–21, 1996.

B Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. Analysis of selected computer security intrusions: In search of the vulnerability. Technical Report 275, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1996. Presented at NORDSEC – First Nordic Workshop on Secure Computer Systems, Göteborg, Sweden, Nov. 7–8, 1996.

C Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 154–163, Oakland, California, May 4–7, 1997.

D Ulf Lindqvist and Erland Jonsson. A map of security risks associated with using COTS. *Computer*, Vol. 31, No. 6, pp. 60–66, June 1998.

Part II: Methods to improve system security

E Ulf Lindqvist, Per Kaijser, and Erland Jonsson. The remedy dimension of vulnerability analysis. In *Proceedings of the 21st National Information Systems Security Conference*, pp. 91–98, Arlington, Virginia, Oct. 5–8, 1998.

F Stefan Axelsson, Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. An approach to UNIX security logging. In *Proceedings of the 21st National Information Systems Security Conference*, pp. 62–75, Arlington, Virginia, Oct. 5–8, 1998.

G Ulf Lindqvist and Phillip A Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 146–161, Oakland, California, May 9–12, 1999.

H Ulf Lindqvist, Douglas Moran, Phillip A Porras, and Mabry Tyson. Designing IDLE: The intrusion data library enterprise. Abstract presented at RAID '98 (First International Workshop on the Recent Advances in Intrusion Detection), Louvain-la-Neuve, Belgium, Sept. 14–16, 1998.

This page is intentionally left blank.

Contents

Preface

The voyage of discovery is not in looking for new landscapes but in looking with new eyes.

ANONYMOUS

Children are born true scientists. They spontaneously experiment and experience and reexperience again. They select, combine, and test, seeking to find order in their experiences: "Which is the mostest? Which is the leastest?" They smell, taste, bite, and touch-test for hardness, softness, springiness, roughness, smoothness, coldness, warmness: they heft, shake, punch, squeeze, push, crush, rub, and try to pull things apart.

RICHARD BUCKMINSTER FULLER (1895–1983)

When you have worked intensively and for a long time on a project, and you have finally finished, then you have gained something known as experience. This means that if you had known in the beginning of the project what you know at the end of it the result would have been much better. So, from a perfectionist's point of view, you should actually do it all over again and get it right the second time. On the other hand, it's good to know when it's time to wrap it up and move on to new challenges. Experience must be shared with others, however, to give them a chance to avoid the mistakes you made and to repeat and develop your successful actions. I chose not to do my research studies all over again, but I did choose to publish my experience in the thesis you are now reading. Along the way, I have received help and support from many persons and organizations, to whom I would now like to express my sincere gratitude.

The first person I thank is my thesis advisor, Erland Jonsson. His wise guidance and dedicated support—from the day he encouraged me to join his group as a PhD student to the day I defend this thesis—has been of immense value to me. Erland's great sense of humor makes it fun to work with him, and it has probably also helped him to put up with me for almost five years.

Next, I thank the other co-authors of my publications in this thesis (in order of appearance): Tomas Olovsson, Ulf Gustafson, Per Kaijser, Stefan Axelsson, Phil Porras, Doug Moran and Mabry Tyson. We have had some times of hard work together, but also great fun when we have prepared our papers. It has been a privilege to work with you all, and I hope that we will write more papers together in the future.

Thanks to the other PhD students in the security research group at Chalmers—Helén Svensson, Emilie Lundin, Dan Andersson, Hans Hedbom, Håkan Kvarnström, Stefan Lindskog, Florian-Daniel Oţel and Lars Strömberg—for many stimulating discussions and other, less work-related, good fun. I look forward to reading your theses in due course and wish you the best of luck with your research studies. I also thank other past and present colleagues at the Department of Computer Engineering at Chalmers for their dedicated efforts in education and research and for contributing to a friendly working atmosphere.

The summer I spent as a visitor at the Computer Science Laboratory of SRI International in California was a wonderful experience and of great value, both in terms of the research I could participate in during the visit as well as the avenues it opened for the future. I thank all the people who gave me and my family help and support and ultimately made the visit possible, especially my hosts Phil Porras and Peter Neumann for all their help with practical matters, great and small. Phil is a rare combination of a visionary in security research who also does hard work down in the programming trenches, and it is very inspiring and fun to work with him and his team. Peter has given me great inspiration, firm support and good advice, not least for the completion of this thesis.

I am grateful to all the anonymous reviewers whose comments on the manuscripts I submitted to conferences and journals greatly helped to improve the quality of my publications. I also thank all the students who participated in our various experiments and thereby provided us with valuable research data.

Many thanks to Janet Vesterlund for prompt and reliable help with the English language editing and last-minute proofreading of many of my manuscripts through the years, including this text. Any remaining typos or style-breaking sentences are the ones I did not finish in time for Janet to read and correct. I also thank Lora Sharp-McQueen and Michael D Evans for their language editing of Papers C and D, respectively.

Because I actually started my professional life as a typesetter, I can be very nitpicking about typographical details that most other people never even notice. Therefore, I am very grateful to the wonderful people who also have this particular trait and, in addition, were kind enough to write TEX, LATEX, BIBTEX, and numerous extensions to those, and made all this software public. Before I saw the light, I produced some publications using other (commercial) software packages, so I had to convert those texts before including them in this thesis. There is of course a risk that errors were introduced in the process, and I take full responsibility for those and for any other remaining errors in this thesis, although I hope that they are few.

Research usually requires financial support, and the work presented in this thesis is no exception. My main sponsors have been Chalmers University of Technology and The Swedish National Board for Industrial and Technical Development (NUTEK). In addition, several other organizations have made various contributions through grants, stipends, funds, or cooperation. These include SRI International, SKF, Ericsson, Telia and IBM. I am grateful for all the financial support that my research projects have received through the years.

Last, but not least, I dedicate this work to my family. At the time when I started my PhD studies, my wife and I had no children. Now, some five years later, we have three wonderful children. It would not have been possible to combine my two roles—a father and a PhD student—without the rock solid support from my beloved wife Lottha. Sweet Lottha, I thank you for making my dreams come true. Dear children, this thesis hopefully shows that I was successful in one of my roles, and I hope that none of you will ever need to tell a therapist that I failed in the other role. Thank you Erik, Gilbert and Bianca, for all the happiness you give us. Finally, I thank my parents Rigmor and Bertil Lindqvist, not only for breeding and feeding me, but for letting me grow.

Ulf Lindqvist, Göteborg in August 1999

This page is intentionally left blank.

Introductory summary

This page is intentionally left blank.

1 Introduction

Computer and network security is no longer a concern only for traditionally security-conscious organizations, such as military and financial institutions, but for *every* organization and individual who uses computers. This is due to the rapidly increasing trend in today's society to depend on computers for more and more of our professional (and leisure) activities and to connect these computers to networks such as the global Internet. Today, when more and more of the valuable assets of an organization are in the form of information stored in computerized information systems, the security of the systems has become a critical issue. However, it is remarkable how little attention was paid to security issues in the design of most systems that are in use today. As a matter of fact, people who should have cared about computer security issues have chosen to ignore the problem. The problem has not simply gone away, but has rather led to the dangerous situation we are in today: systems designed with little or no security are trusted with our most valuable information and are relied upon in critical situations. This has motivated the Department of Computer Engineering at Chalmers to conduct research with the overall aim of increasing security in existing as well as future information systems. This thesis presents thorough analyses of the fundamental technical aspects of the security problem and of some partial solutions.

There are many reasons why a computer system can behave in an undesired way. For a problem to be categorized as a security problem, it must in some way involve the fact or possibility that a human being does something that is not permissible. It is normally the person or organization who owns the system and/or the information who decides what is allowed and what is not. Wrongdoers can be categorized as *insiders* or *outsiders*. Insiders are persons related to the owner organization who try to misuse or extend their privileges. Outsiders are attackers who are unrelated to the owner organization. Within the community of security officers and researchers, the insider threat is considered much more dangerous than the threat from outsiders, but the media have conveyed the opposite picture to the general public.

Throughout this work, it has been an objective to have a holistic perspective on security and to consider as many different aspects as possible. However, to reach the desired technical depth, it was necessary to exclude some areas. For example, this thesis addresses many technical aspects of computer misuse but is not concerned with offenders as human beings—this should be treated in other disciplines such as criminology and psychology. Also, at the time of this writing, the so called "millennium bug" or "Y2K problem" is a topic of great concern because it is feared that many computer systems will not correctly handle the transition from the year 1999 to the year 2000. Of course, this widespread logical equivalent of a time bomb is a risk to many organizations and nations and may have security implications as well, but it is not in itself a security problem, strictly speaking, and is therefore not addressed *per se* in this thesis. However, let us hope that the experience gained from the large inventory projects currently conducted in many organizations will not be wasted when the Y2K problem has been settled. That knowledge gives us a golden opportunity to take adequate security measures for the systems that we will depend upon in the coming millennium.

2 Thesis objective and scope

Traditionally, research on computer security has focused on helping developers of systems to prevent security vulnerabilities in the systems they produce before the systems are released to customers. Also, in most studies on network security, only outsiders are considered to be potential attackers. All of these areas are important but need to be complemented with research supporting owners, developing detection and recovery mechanisms, and studying the insider threat.

In most cases, it is the owners of systems and information who risk a loss (of money, goodwill, competitive advantage etc.) when security is violated. It is also the owners who are responsible for keeping the systems secure, and they must be supported in this difficult mission. Many owners cannot (or are not willing to) dedicate large resources to security, even though their business can be severely damaged by misusers. Also, owners typically have a mix of systems in their organizations and do not have personnel who are expert in security for all these kinds of systems. Therefore, methods for improving security must be *generic* with respect to applicability to different systems as well as *efficient* with respect to demands on computing power and human involvement.

The overall objective of security research at the Department of Computer Engineering is to find efficient generic methods to improve the security of existing and future systems. This problem can naturally be divided into the following four research problems, of which *i* and *ii* have been studied in the two previous doctoral dissertations published by our group [25, 52], while problems *iii* and *iv* are the subject of the present thesis:

i) What do we mean by ‘security’? How is it related to other objectives such as ‘dependability’?

ii) How can security be quantitatively measured in order to verify whether the security of a system has been improved?

iii) What characterizes threats to security?

iv) What methods can be used to counteract threats and thereby improve security? How should these methods be designed in order to be efficient and generic?

Part I of this thesis presents studies of problem *iii*. Some early results on the subject were also presented in a licentiate thesis by the present author [40]. After detailed studies of threats, it was natural also to start work on problem *iv*, that is, methods for improving security. That work, presented in Part II of this thesis, is not focused on traditional prevention of security violations but rather on developing mechanisms that come into play after an attacker has initiated some action (see Section 3.3 for a discussion on the relationship between prevention and detection) or after a vulnerability is discovered in a system in operation. Specifically, we have chosen to study logging and detection of intrusions and remediation of vulnerabilities.

3 Background and frame of reference

3.1 Defining security

One commonly used definition is that computer, network and information security (sometimes referred to as IT security but, henceforth in this text, *security* for short) concerns the protection of computer systems, the networks interconnecting such systems and the information stored, processed and transmitted within the systems and networks against *intentional* attacks. The discipline is relatively young,[1] and the terminology is still subject to much discussion and confusion. One of the few matters on which the community is close to consensus is the definition of the following three aspects (or objectives) of security:

Confidentiality: prevention of unauthorized disclosure of information

Integrity: prevention of unauthorized modification of information

Availability: prevention of unauthorized withholding of information or resources

The definitions above are quoted verbatim from ITSEC [51] but are basically the same in the Common Criteria [11] and in the British "code of practice" [9]. These three aspects cover much of what security is about, but not everything. For example, we might wish to add *authenticity*, meaning verification of the identity of a communicating party, and *non-repudiation*, meaning prevention of the possibility for a communicating party to deny transmission or receipt of a message. There is also debate over in which of the three aspects, if any, prevention of unauthorized use of computer and network resources should be included (see for example Paper C).

It should also be noted that there are superordinate concepts in which security is included, such as *dependability* [37] and the more recently suggested *survivability* [49]. In summary, there are several properties that we expect from good[2] systems—security is but one of those properties—and it is still subject to research how security interacts and may be integrated with other such properties such as safety and reliability [26, 56]. However, this problem is beyond the scope of this thesis, in which we treat security as the root of our terminology tree[3].

An alternative to the definition of security as protection against intentional attacks is the following:

Security concerns the preservation of confidentiality, integrity and availability (and possibly additional aspects), regardless of whether the threats are intentional attacks or accidental mistakes or mishaps.

[1]In the first published collection of seminal papers from the UC Davis "History of Computer Security" project [8], the oldest paper is from 1970 [66]. This suggests that the discipline is the same age as the present author, hence "relatively young". However, studies of computer security problems were initiated in the 1960s and influenced the design of Multics [57], for example.

[2]Here, I deliberately use an as all-encompassing and general a term as possible in an attempt to be unbiased.

[3]This tree, like most trees in computer science and engineering, but unlike most real wooden trees, has its root at the top.

This definition does not make security much harder, it simply extends the responsibilities of the field, as observed by Peter G Neumann [48, p. 130]:

> There are no essential functional differences between accidental and intentional threats, with respect to their potential effects; the consequences may be similar—or indeed the same. However, there are some differences in the techniques for addressing the different threats. The obvious conclusion is that we must anticipate both accidental and intentional types of user and system behavior; neither can be ignored.

The terms *intrusion* and *vulnerability* are used very often in this thesis, and I now present the definitions that I have used in my papers:

Intrusion is a successful event from the attacker's point of view and consists of

1) an *attack* in which a *vulnerability* is exploited, resulting in
2) a *breach* which is a violation of the explicit or implicit *security policy* of the system.

Vulnerability is a condition in a system, or in the procedures affecting the operation of the system, that makes it possible to to perform an operation that violates the explicit or implicit security policy of the system.

Security policy is some statement about what kind of events are allowed or not allowed in the system. An explicit policy consists of rules that are documented (but not necessarily correctly enforced), while an implicit policy encompasses the undocumented and assumed rules which exist for many systems.

The definitions above have some interesting implications. If we do not have a security policy, then we cannot determine whether a certain event constitutes an intrusion or not—if there is no law, then no act can be illegal. It is also implied that the existence of a vulnerability is dependent on the security policy. For example, if we have two systems, A and B, that are technically identical but have different security policies, then an operation which is possible to perform in both systems, but forbidden according to the policy of A and allowed according to the policy of B, would be an intrusion in system A but not in system B. Consequently, the condition allowing the operation would be considered a vulnerability in system A but not in system B. It should also be noted that, if the configuration of an access control system does not fully conform to the security policy as intended by the information owner, then that condition is also considered a vulnerability. Finally, the definition of intrusion makes the established term "intrusion detection" somewhat too specific, because it is often desirable to detect not only successful attacks but also failed attack attempts and other suspicious behavior, as discussed in Section 3.2.

3.2 Intrusion detection: The basics

This section presents a brief introduction to the basic concepts of intrusion detection. The reader interested in an in-depth treatment of the subject is referred to other work, for example [1, 7, 14, 45].

Intrusion detection is the established name for a category of security mechanisms that may be viewed as the equivalent of a burglar alarm for the logical world of computers and networks. A component that implements such mechanisms is called an intrusion detection system (IDS). If we picture traditional preventive mechanisms as a wall that protects our resources from attackers, we would like an IDS to respond when any of the events illustrated in Figure 1 occur.

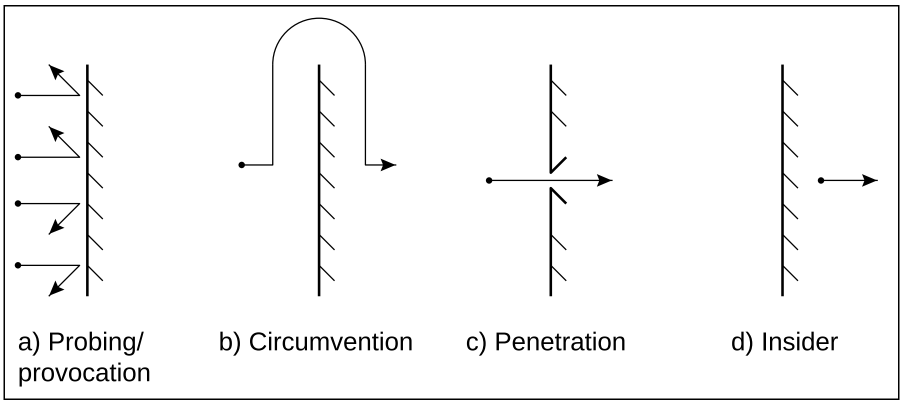


Figure 1. Events that should trigger IDS response.

A model for IDS architecture that has gained widespread acceptance is the view of an IDS as a collection of components specialized in performing their respective tasks and which communicates via message passing. An elegant categorization of IDS components was formulated by the Common Intrusion Detection Framework (CIDF) working group [64]:

E-boxes are event generators that sample events in the particular environment they are specialized for and immediately produce messages describing their observations. E-boxes do not consume or store messages.

A-boxes are event analyzers that take in messages and analyze them to produce conclusions. They send messages to R-boxes and/or other A-boxes.

D-boxes are event databases that provide persistent storage of messages for later retrieval.

R-boxes are response units that consume messages directing them to carry out some action on the behalf of other CIDF components. R-boxes also carry out the requested response actions.

The tasks assigned to each of these types of boxes all require careful design considerations and trade-offs for the IDS to be efficient and effective in large modern computer networks. For example, what events should an E-box report and in what detail? How should an A-box perform its analysis to correctly identify intrusive behavior? How should it be designed not to run out of memory in the long run? How can we ensure that its processing rate is at least as high as the arrival rate of new incoming messages? For how long should a D-box store messages? What response is it appropriate for an R-box to carry out? Should it restrict its actions to reporting to a human operator or should the IDS autonomously shut down connections, terminate processes or even launch counterattacks?

Although some of these problems have been the subject of extensive research (in particular, issues related to data analysis), much research still remains before we have solutions for them. In the remainder of this section, we focus on the data analysis components of an IDS (the A-boxes) and present a categorization of analysis principles.

On some level of abstraction, an IDS (or an A-box) can be seen as a detector similar to a signal detector, as observed by Helman and Liepins [23]. The detector may be *binary*, providing an absolute categorization of its input as either normal (0) or intrusive (1), or it may be *graded*, ranking its input on a continuous scale from 0 to 1. Helman and Liepins point out that, to every graded misuse detector MD_g, a family of binary detectors $(MD_{b,\tau})$ can be associated simply by introducing a threshold parameter $0 \leq \tau \leq 1$ and defining $MD_{b,\tau}$ as:

$$MD_{b,\tau}(x) = \begin{cases} 0 & \text{if } MD_g(x) < \tau \\ 1 & \text{otherwise} \end{cases}$$

In practice, the output of a graded detector is typically passed on to another decisionmaker—a human operator or, for example, a higher level detector (binary or graded) that correlates the output of several lower level detectors. Henceforth in this text, we will view an IDS as a binary detector to keep the discussion from becoming unecessarily complicated.

If the IDS erroneously categorizes a normal event as intrusive, it is called a *false positive* (or false alarm), and the erroneous categorization of an intrusive event as normal is called a *false negative*. Ideally, all intrusive behavior would be categorized as such (*true positive*), while the majority of events that are normal should not lead to any response from the IDS (*true negative*).

What principles can an IDS utilize when analyzing events in order to separate normal events from intrusions? Table 1 shows a categorization of IDS data analysis principles in two dimensions. The policy dimension has the two categories of *default permit* and *default deny*. Default permit means that the system has an encoded knowledge of the kind of behavior it should report as intrusive and will quietly accept all other behavior. The opposite is default deny, which means that the system has knowledge about what kind of behavior is allowed and will report all other behavior as intrusive. The knowledge dimension has the categories *static* and *dynamic*, which denote whether the knowledge (of what constitutes intrusive or allowed behavior) changes automatically and depends on previously observed behavior (dynamic) or must be changed manually (static). Before we examine each principle in detail, it should be noted that it is generally agreed among IDS researchers that a combination of analysis principles is often needed to achieve effective detection.

In box 1 of Table 1, we find the commonly used principle known as signature-based misuse detection. The principle, its possibilities and limitations are further described in Paper G. In summary, it means that the IDS is equipped with knowledge in the form of production rules, state machines, string patterns or some other

Table 1. A categorization of IDS data analysis principles.

Policy					
Default permit	Signature-based misuse detection	1	2	< ? >	
Default deny	Specification-based misuse detection	3	4	Profile-based anomaly detection	
	Static			Dynamic	Knowledge

encoding of what constitutes undesired behavior and how it should be reported. When the knowledge of the IDS should be changed—typically extended when a new type of intrusion becomes known—the change is made through a special update operation. The update is typically performed manually but could also be made automatic, for example, based on a subscription mechanism. The work described in Paper H is an effort aimed at supporting such automatic update procedures by creating a standard open format for the information. The advantages of this principle are typically a low false alarm rate and a low performance overhead, while a disadvantage is the limited ability to detect unprecedented violations (limited but existing, see the discussion in Section 7.4.4).

In box 4, we find the other major analysis principle, profile-based anomaly detection. The principle was first outlined in Anderson's seminal paper on intrusion detection [4] and further developed and implemented in IDES [15, 16]. The idea is to continuously build profiles of normal behavior for particular principals (users, processes, devices, etc.) and to report when the behavior of a principal deviates (much) from its normal profile. The main advantage of this principle is that the IDS does not need to have any knowledge about particular intrusion types, vulnerabilities or security mechanisms and is consequently able to detect previously unknown intrusion types. The main disadvantage of anomaly detection is that it detects anomalies only. Anomalies are not necessarily security violations and, in an environment with fluctuating behavior, the false alarm rate could be overwhelming. Also, all security violations are not necessarily anomalous[4].

In box 3, we find a principle not widely used in practice but fundamentally different from those described above. Instead of specifying the undesired behavior (as in box 1), we specify the permitted behavior and report everything else. This principle, called specification-based misuse detection, was proposed and applied to privileged programs by Ko *et al.* [30–32]. The advantage of this approach is the same as for box 4, namely, the possibility of detecting previously unknown intrusions. The main disadvantage is that it is usually difficult to specify all permitted behavior correctly, especially when trying to write a specification for an already existing program. The connection to access control is not very far-fetched because,

[4]For example, let us say that I am a user who regularly works with sensitive information. An attacker who has stolen my password or authentication device logs in to the system from my usual terminal at a time when I usually log in and accesses the files I usually work with. The granularity required by a profiling IDS to consider this security violation anomalous would probably produce too many false alarms to be useful.

if we can specify the permitted behavior correctly, we can in fact place a ‘guard’ in a strategic place that examines every attempted operation before it is executed. Thus, we would be able not only to report violations but actually deny them to prevent damage, as implemented in Janus [21] and TIS Wrappers [19], for example. One could argue that this is nothing but a reference monitor, a fundamental concept in computer security [3]. However, it should be kept in mind that intrusion detection and other retrofit security mechanisms try to cover cases in which the reference monitor is lacking, flawed or otherwise inadequate (see the discussion in Section 3.3).

Finally, what should be placed in box 2? Dynamic knowledge and default permit policy suggests that an IDS using such a principle would automatically learn what constitutes undesired behavior by studying past events. What would be the benefits of such a system? One could think of a semi-automatic approach where we train the IDS by telling it that “here comes an attack” and then carry out the attack. The IDS would then draw its own conclusions as to what signs are significant for the detection of that type of attack. This would relieve us of writing traditional detection rules, such as the rules examplified in Paper G. A step in this direction may be the approach by Lee *et al.* [38], in which data mining techniques are used to find the features that characterize intrusions in a large data set.

3.3 Detection versus prevention

Security for computers and networks has much to learn from security in the physical world. We are human beings, trying to protect our systems against the hostile actions of other human beings. Thus, we must study how humans during all time have protected their physical assets against other humans. The fact that our assets, threats and mechanisms are in the logical world of computers and networks makes our mission sometimes harder and sometimes easier, but not very different.

Specifically, intrusion detection is not a new idea nor is it unique to computer systems. The sentinel guarding the perimeter of a Roman military camp, the watchman in the tower of a medieval castle and the electronic burglar alarm active at night in the local grocery store are all examples of intrusion detection mechanisms. One thing that these and other examples of intrusion detection in the physical world all have in common is that they complement intrusion prevention mechanisms, for example a physically strong perimeter protection. Neither the prevention nor the detection mechanisms are infallible, but together they make it harder for the intruders to reach their goal. This is an important observation; just because a prevention mechanism (or detection mechanism, for that matter) is not perfect, does not necessarily mean that it is useless. We expect it to function correctly most of the time but, when it fails, we must be prepared and have other cards to play.

In fact, this is true not only for protection against intentional attacks but also for protection against accidents and other inadvertent failures. For example, we must accept that all materials in buildings cannot be made fireproof; we therefore have smoke detectors and heat detectors connected to alarm bells, sprinklers and fire department dispatchers. There are sensors that detect whether the temperature in a supermarket freezer rises over a certain threshold, sensors that measure the radiation around a nuclear power plant and so forth. Prevention, detection and response mechanisms go hand in hand in all areas of society where we wish to protect ourselves against undesired events. It is my firm belief that the field of computer security intrusion detection would benefit from studying the successes and failures of detection and response mechanisms in other fields. This is especially true for the more non-technical aspects, for example the interface between the system and the human operators, including alarm correlation and presentation, tolerable false alarm rates etc.

If we accept that prevention and detection mechanisms should complement rather than replace one another, we realize that they may very well be different parts of the same security package where the existence of one mechanism does not imply that the other is useless. For example, an IDS can be bundled with an operating system, a database system or a firewall product, just as other security mechanisms often are (such as access control and logging). However, this does not mean that the IDS should always be closely integrated with the preventive mechanisms—that would create a single point of failure. Again, we can turn to the physical world and see that monitoring functions are often kept separate and independent from the monitored activity.

A common objection to intrusion detection, especially misuse detection, is: “If you know of a security hole, why can’t you just fix it instead of trying to detect when someone exploits it?”. The immediate answer is that it is true that a “fix” would be more appropriate in cases where that would be possible, but it is not always possible to “fix” a hole without making fundamental changes to a system architecture. In addition, the configuration of a system is rarely static. New components are installed, new users are added and so forth. There is always a risk that a vulnerability is introduced when changes are made. Finally, both when designing and operating intrusion detection mechanisms, it is important to remember that the IDS is there to warn us when the preventive mechanisms are being provoked, circumvented or penetrated (see Figure 1).

4 Related work

Each of the papers included in this thesis include references to previous work related to the problems addressed in that particular paper. I have thus chosen to restrict this presentation of related work to a brief summary of seminal work in the field, followed by a more lengthy discussion on related doctoral dissertations published during the course of my work. By analyzing and commenting upon recent dissertations from the perspective of my own work, I would like to show where my dissertation fits into the large picture of contemporary research in the field.

4.1 Seminal work

Although the field of computer security seems to evolve very rapidly, it is wrong to believe that everything worth reading was published no more than a couple of years ago. Systems have changed radically, but many fundamental principles remain the same, see for example Saltzer and Schroeder [58].

Several discussions on the nature of security threats, based on security evaluation and penetration analysis, were published in the 1970s. Lackey presented an interesting taxonomy of penetration techniques, although with an unusual terminology [35]. Linde presented the Flaw Hypothesis Methodology [39], Attanasio *et al.* described a typical penetration analysis [6], and Neumann categorized security flaws and discussed how such flaws may be avoided [47].

With the 1980s came intrusion detection. Anderson's analysis of the nature of attackers in terms of the difficulty associated with detecting them [4] is considered fundamental and is cited in almost every published work on intrusion detection. The first IDS prototypes, with names such as IDES [16], MIDAS [59], Haystack [60] and Wisdom & Sense [65], have greatly influenced later research on intrusion detection.

4.2 Recent doctoral dissertations

4.2.1 Jonsson 1995 [25] and Olovsson 1995 [52] Two doctoral dissertations have been published by members of our research group before the present one, namely, "A Quantitative Approach to Computer Security from a Dependability Perspective" by Erland Jonsson and "Practical Experimentation as a Tool for Vulnerability Analysis and Security Evaluation" by Tomas Olovsson. These two theses describe work on defining the security concept in relation to dependability and similar concepts and on the theory and practical realization of experiments for measurement of operational security. The student intrusion experiments, which were planned and initiated by Jonsson and Olovsson as described in their theses, produced data that was useful for other studies than just the original one on measurement of security, for example work presented in the present thesis. The results of Olovsson's and Jonsson's work include a quantitative model of the intrusion process [27] and a proposed framework for integration of the concepts of security and dependability [26].

4.2.2 Kumar 1995 [34] In his thesis entitled “Classification and Detection of Computer Intrusions”, Sandeep Kumar from Purdue University presents a scheme for categorization of intrusions based on the pattern specifications that can be used to match signatures left in a system by an intrusion, typically found in an audit trail. The four categories in Kumar’s scheme, arranged in a hierarchy where the representability of intrusion signatures increases as we go from left to right, are

$$\text{Existence} \subseteq \text{Sequence} \subseteq \text{RE patterns} \subseteq \text{Other Patterns},$$

where RE patterns are extended regular expressions allowing the use of an AND primitive. The existence category is different from the others in that its patterns match an existing condition in a system (for example, the presence of a certain file) rather than an event signature.

Kumar claims that pattern matching is very suitable for misuse detection in that it can be made very efficient and generic with respect to audit trail syntax. He presents a model of matching, based on Colored Petri Nets, which uses the proposed classification. The thesis contains a thorough description of the theoretical possibilities and limitations of the proposed model. However, the difficulty for a user to encode knowledge of an intrusion signature in the proposed matching model is not analyzed; Kumar simply states that the task is “nontrivial”. For example, the simple intrusion scenario exploiting a *setuid* shell script (further described in Paper F), consisting of the commands

```
1) ln setuid_shell_script -i
2) -i
```

would be encoded as [34, p. 90]:

```
FILE1 = this[SRC_FILE] && FILE2 = this[DEST_FILE] &&
SHELL_SCRIPT(FILE1) = 1 && OWNER(FILE1) != this[EUID] &&
basename(FILE2) = "-*" &&
(FPERM(FILE1) & XGRP = 1 || FPERM(FILE1) & XOTH = 1)
```

It is left to the reader to judge whether this is easy to read and use or not, but it should be noted that it is indeed difficult to design a signature description language that is both sufficiently powerful to capture all details of any intrusion scenario and at the same time easy to use for non-experts. An alternative to Kumar’s pattern matching approach is the expert system approach described in Paper G in this thesis. Kumar’s model was later implemented in an IDS prototype at Purdue University with the somewhat provocative name IDIOT [12].

4.2.3 Ko 1996 [30] Calvin Ko from the University of California at Davis presents his work on a novel approach to intrusion detection, called specification-based monitoring, in his thesis entitled “Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach”. See Section 3.2 for a description of the specification-based analysis principle (box 3 in Table 1) in comparison with other principles.

Ko begins with a detailed description of five known security vulnerabilities in UNIX programs and how they can be exploited. Next, he presents a model for reasoning about system traces and identifies four aspects of program behavior as security-relevant: accesses, sequencing, synchronization and race conditions. Ko developed a special type of grammar, called parallel environment grammars (PE grammars) for specifying trace policies of concurrent processes. A trace policy for a program (the allowed behavior) is specified as a grammar, and it is stated that an execution trace of a process satisfies the policy if it is a sentence of the language specified by the grammar. Example trace policies that can be used to detect the previously presented exploits are written using PE grammars. The design and implementation of a system monitoring the execution of selected programs in a distributed system are presented.

The basic reason for developing the specification-based approach is the possibility of detecting attacks exploiting previously unknown vulnerabilities in programs. However, this assumes that we specify in our policy *all* allowed behavior for each program—a difficult, tedious and sometimes outright impossible task. Ko has observed this problem and suggested two alternative approaches:

1) We can focus on some important properties of a program—such as synchronization—and specify the desirable behavior with respect to these properties only.

2) We can base the trace policy on some suspected or existing weaknesses of the program.

A criticism that can be made about Ko's presentation of his work is that he has only shown examples of the latter approach, which is *not* fundamentally different from traditional misuse detection. It is then simply a question of which subset of the set of all possible events is the smallest and easiest to specify, the set of all allowed events, A, or the set of all prohibited events, $\neg A$. However, Ko has made some significant contributions to our understanding of the possible methods for intrusion detection and, if he had used examples in which previously unknown intrusions were indeed detected, he would have made it clearer to the reader that his work is more than just a negation of misuse detection.

4.2.4 Howard 1997 [24] In his thesis entitled "An Analysis of Security Incidents On The Internet 1989–1995", John D Howard of Carnegie Mellon University presents an extensive analysis of Internet security incidents handled by the CERT Coordination Center (CERT/CC). The CERT/CC database is kept confidential and is consequently not available for independent evaluation and research.

Howard starts by forming a taxonomy for categorization of attacks based on the dimensions attackers, tools, access, results and objectives. The taxonomy, shown in Table 2, and the related discussion and motiviation are in some respects strikingly similar to the taxonomy proposed in Paper C. For example, both schemes attempt to address the previous lack of a category for unauthorized use of a system, although in slightly different ways. Howard's thesis and Paper C were published almost simultaneously, and neither author was aware of the other's work.

Next, Howard applies his attack taxonomy to 4,299 security incidents from the CERT/CC database, pointing out that an incident typically consists of several attacks. The analysis, presented in a large number of statistical graphs and charts is interesting reading, especially as security researchers usually do not have access to this kind of material (see Section 5.1). In some cases, the description of selected examples of incidents is more anecdotal than strict in a technical sense, perhaps because the author is with CMU's Department of Engineering and Public Policy and not with a traditional computer science or computer engineering department.

The thesis concludes with a discussion of the usability of the taxonomy and some recommendations for CERT/CC and Internet users. Howard actually recommends the CERT/CC to start disclosing incident data based on a taxonomy, without revealing the names of the involved sites. Howard also points out that the CERT/CC database describes almost exclusively attacks performed by outsiders. This is an interesting difference between the data used in his thesis and the data used in Paper C, as the attackers in the student experiment—upon which the latter work is based—all acted as insiders.

Table 2. Computer and network attack taxonomy [24, p. 73].

Attackers		Tools				Results		Objectives
Hackers		User Command				Corruption of Information		Challenge, Status
Spies		Script or Program				Disclosure of Information		Political Gain
Terrorists	➧	Autonomous Agent	➧	. . .	➧	Theft of Service	➧	Financial Gain
Corporate Raiders		Toolkit		⋮		Denial-of-service		Damage
Professional Criminals		Distributed Tool		⋮				
Vandals		Data Tap		⋮				

Access						
Implementation Vulnerability		Unauthorized Access				Files
Design Vulnerability	➧	Unauthorized Use	➧	Processes	➧	Data in Transit
Configuration Vulnerability						

4.2.5 Mounji 1997 [44] Abdelaziz Mounji from the University of Namur in Belgium presents work on misuse detection through rule-based analysis of audit trails in his thesis entitled "Languages and Tools for Rule-Based Distributed Intrusion Detection". Three acronyms are central to this thesis: RUSSEL (Rule Based Sequence Evaluation Language), ASAX (Advanced Security and Audit Trail Analysis on UNIX) and NADF (Normalized Audit Data Format). The main part of the thesis presents the design of the rule-based language RUSSEL, which is specifically crafted for the analysis of audit trails in NADF format, and how it was implemented in an IDS project called ASAX. A summary of the ideas underlying ASAX, RUS-

SEL and NADF can be found in [22]. The thesis also presents a format adaptor generator for easier conversion of different audit trail formats into NADF. Mounji proposes an integration of intrusion detection and static configuration analysis (also known as vulnerability scanning) and, finally, shows how RUSSEL can be used in a distributed environment.

The objectives of RUSSEL are indeed similar to those of P-BEST, presented in Paper G. Mounji states that power and convenicence, reusability, efficiency and adaptability should be required by an IDS in general and a description language for misuse detection in particular. Like P-BEST, RUSSEL has a versatile interface to the C language, enabling the user to call arbitrary routines when the rule-based language is insufficient. However, while P-BEST tries to adhere strictly to the declarative programming paradigm, RUSSEL deliberately uses imperative programming. Also, P-BEST is a general expert system that can be adapted to any type of input data (for example, recorded network traffic), while RUSSEL is specifically targeteted to audit trail analysis.

4.2.6 Krsul 1998 [33] In his thesis entitled "Software Vulnerability Analysis", Ivan Krsul of Purdue University presents a categorization of software vulnerabilities applied to collected samples stored in a database. When defining a software vulnerability, he observes that a vulnerability depends on policy and defines a *policy* as rules that are actually enforced while *expected policy* are rules that users expect a system to enforce. This is similar—but not identical—to our notion of explicit policy and implicit policy (see Section 3.1). In fact, the most significant aspect of Krsul's work, in my opinion, is that he identifies users' exepectations of program behavior and programmers' assumptions about the program execution environment as underlying causes of software vulnerabilities.

Krsul attempts to show that all previously presented taxonomies in the security field[5] fail to meet basic requirements on objectivity and generality, according to the classical categorization theory, and claims that this is one of the main contributions of his thesis. It is true that many attempted categorizations of security phenomena are of limited value because of their apparently *ad hoc* nature, but Krsul's dismissal of many taxonomies based on the risk of subjective categorization appears somewhat Utopian in the light of prototype theory and the concept of embodied categories, as dicussed in Section 5.2.

Features for categorizing vulnerabilities and a database of collected vulnerability samples categorized with respect to these features are presented. The database structure shows some similarities with the structure proposed in Paper H, although the former is focused on vulnerabilities and includes some exploit information while the latter is focused on intrusion signatures and includes some information about the exploited vulnerability. Actually, each vulnerability entry in the database can include an IDIOT pattern [12] for detecting an exploitation of the vulnerability. Results of applying several data analysis and visualization tools to the database are discussed and, finally, Krsul presents a predictive categorization of vulnerabilies with respect to programmers' (incorrect) assumptions about the execution environment of the program.

[5]With the notable exception of Paper C, which Krsul does not cite.

5 Research methodology

Categorization is not a matter to be taken lightly. There is nothing more basic than categorization to our thought, perception, action, and speech.

GEORGE LAKOFF [36, p. 5]

The process of forming meaningful classifications of observed entities is a difficult intellectual task, and usually precedes the development of a theory about the entities.

ROBERT E STEPP and RYSZARD S MICHALSKI [62]

My most frequently applied research method has been one of data collection and analysis through *observation and categorization*. In this section I describe and discuss my research methodology, but first a note on the terminology: The term *classification* is often considered synonymous with categorization, and I have used it as such myself, for example in Paper C. However, as Dr. Per Kaijser so kindly pointed out to me, people working in the security field tend to interpret 'classified' as meaning 'secret', rather than 'labeled' in a general sense. Therefore, I now use 'category' and 'categorization' instead of 'class' and 'classification'.

5.1 Data collection

Empirical data is essential for the observing researcher. In the security field, it can be very difficult to obtain intrusion-related data of the desired quality, for several reasons:

- Intrusion-related data may reveal sensitive information about the victim site that in turn may be used to further threaten its security. For example, it could tell potential attackers what types of attacks the site is not protected against, what logging and other security mechanisms that are applied, details about the internal network structure, confidential messages and so forth.
- Many sites do not publicly admit that they have had security incidents because they are afraid that doing so would hurt their public image. Consequently, they cannot give away any data about such incidents.
- There can be details about an incident that are known only to the attackers, for example, how long they spent planning the attack. Unfortunately, real attackers are not very willing to share this information with researchers and, if they were, their credibility could certainly be challenged.

The resulting lack of empirical data forced us to start collecting most of our research data ourselves, as described below.

5.1.1 Student intrusion experiments With the primary objective of collecting data for finding measurements of operational security, our group has conducted intrusion experiments in which students were encouraged to attack a certain system for a limited period of time, under careful supervision and with the requirement that all their activities be reported and documented.

Since 1993, the Department of Computer Engineering has given an annual elective course entitled "Applied Computer Security" for final year students of the Master's program in Computer Science and Engineering at Chalmers. We performed one pilot experiment as a feasibility study before the course was first started, as well as three full-scale experiments on a Unix system (1993, 1995, 1996) and one full-scale experiment on a Novell NetWare system (1994). A summary of all our student experiments conducted to date can be found in [41].

We chose to use ordinary students as attackers and to provide them with standard user accounts. In this way, we would model the insider threat, that is, when legitimate users of a system for some reason decide to extend or misuse their privileges. We also ensured that each test environment represented a standard installation of a common computing system based on commercial off-the-shelf components. A successful intrusion was defined as "anything you would not normally be allowed to do on the system", and the sole restriction was that the attackers should not disrupt service to ordinary system users (other students doing lab exercises in other courses).

The data produced in these intrusion experiments has served as an essential basis for most of our research, including Papers B, C, D and F in this thesis. In other research efforts where the experiments were not the primary data source, they still provided valuable inspiration. The extensive data bank is far from completely analyzed and will most likely serve as valuable research material in future studies.

5.1.2 FTP server monitoring To be able to conduct research on how an IDS should be constructed to monitor a server for Internet file transfers (FTP), the EMERALD team at the Computer Science Laboratory of SRI International has recorded network traffic going to and from their FTP server *ftp.csl.sri.com*. The recording was performed on a computer connected to the same broadcast Ethernet network as the monitored server, using the *snoop* tool from Sun Microsystems to collect all packets going to and from network port 21 on the server (that is, all FTP control connections, but no data transfers, were recorded). The verbose output from *snoop* was later post-processed through filter programs which selected the relevant data fields and combined each client command with the resulting server reply to form transactions and associated transactions with FTP sessions. Data was collected continuously for a period of almost five months, resulting in more than 60,000 recorded FTP transactions from 4,800 sessions.

The FTP data recorded contained real attack attempts against the FTP server from apparently hostile Internet users and, for the sake of completeness, a small number of synthetic successful attacks arranged by the EMERALD team. During my participation in the EMERALD project as as summer visitor, the FTP data was frequently used both in the research on rule-based misuse detection (in which I participated) and in the statistical anomaly detection research. The data was also used in the student IDS lab exercise performed at Chalmers, described in Paper G.

5.1.3 The 1998 DARPA Intrusion Detection Evaluation The US Defense Advanced Research Projects Agency (DARPA), which sponsors EMERALD and several other intrusion detection research projects in the US, also sponsors a project whose objective is to evaluate DARPA-sponsored IDS prototypes. In this evaluation project, the Information Systems Technology Group of MIT Lincoln Laboratory has produced and distributed a large collection of data for training and testing IDSs [13].

The data consists of audit logs (Solaris BSM) and recorded network traffic, produced by a completely synthetic simulation representing the normal computer activity of a military base. A number of attacks were inserted into the simulation, with a relatively small intensity. Among the benefits of using simulation are the exact knowledge of what operations were performed ("ground truth") and the fact that no sensitive data would leak into the collection. The large size and high quality of this data collection make it a valuable source for IDS researchers. This data was used in the performance tests presented in Paper G and in related research activities in the EMERALD project at SRI International.

5.2 Categorization as a method for data analysis

The step that followed naturally after data collection was categorization of data samples. We first attempted to apply categorization schemes that had been published by other security researchers but soon discovered that these schemes were not applicable to the type of data we had collected or were not designed for our purposes. Therefore, we started to develop our own categorization schemes.

At first glance, categorization may seem like a natural and simple task that does not require much thought or theory. Although categorization is indeed natural in the sense that it is an integral part of the human thought process, it is seldom simple and there are—perhaps surprisingly—different theories on categorization. Below, I briefly present the classical Aristotelian view of categorization and its suggested successor known as prototype theory and then describe how and why I have used categorization as a method.

5.2.1 Classical categorization Taxonomic categorization has been part of the scientist's toolbox since the days of Plato and Aristotle. Perhaps the most momentous taxonomy in the history of science is the categorization of plants and animals by the Swedish botanist Linnaeus (Carl von Linné, 1707–1778). The traditionally unquestioned theory of categories can be defined as [36, p. 161]:

> All the entities that have a given property or collection of properties in common form a category. Such properties are necessary and sufficient to *define* the category. All categories are of this kind.

The properties that should be chosen as a basis for a categorization depend on the purpose of the categorization. Of course, some choices of properties serve particular purposes better than do other choices, but one should not expect to find a universal answer that is correct in all situations.

5.2.2 Prototype theory In later years, a field called cognitive science has developed, in which the classical theory of categories has been challenged as described by linguist George Lakoff in [36]. According to Lakoff, the classical theory was first questioned by Ludwig Wittgenstein, but the major pioneer in the field was Eleanor Rosch, who focused on the following two implications of classical categorization:

- If categories are defined only by properties that all members share, then no certain members should be better examples of the category than any other members.
- If categories are defined only by properties inherent in the members, then categories should be independent of the peculiarities of any beings doing the categorizing.

By analyzing studies done by herself and others, Rosch could observe that categories, in general, have best examples called *prototypes* and that human capacities do play a role in categorization. On the basis of Rosch's findings, a new theory of categorization, called the *prototype theory,* has emerged. Some of the basic results of prototype theory are [36, p. 56]:

- Some categories are graded; that is, they have inherent degrees of membership, fuzzy boundaries and central members whose degree of membership (on a scale from zero to one) is one. Examples of such categories are 'small bird' and 'green'.
- Other categories have clear boundaries; however, within those boundaries, there are graded prototype effects—some category members are better examples of the category than others. An example of such a category is 'bird', and sparrows and robins are often considered better examples of the bird category than are owls, eagles, ostriches and penguins.
- Categories are not organized only in terms of simple taxonomic hierarchies. Instead, categories "in the middle" of a hierarchy are the most basic, relative to a variety of psychological criteria. For example, 'bird' is at the basic level, while 'animal' is above and 'sparrow' is below the basic level.
- At least some categories are *embodied,* that is, not entirely external to human beings but rather dependent upon human perception, experience of a physical and social character etc.

The degrees of membership are something that protoype theory has in common with fuzzy sets [68]. Zadeh, who founded the theory of fuzzy sets, even claims that an adequate theory of prototypes requires the explicit use of fuzzy sets [69].

There is also an approach to categorization called *conceptual clustering,* considered by some a third theory between the classical theory and prototype theory. The idea is that entities should be arranged into categories that represent simple concepts rather than categories based solely on a predefined measure of similarity [62]. Conceptual clustering is strongly directed towards automatic categorization and machine learning.

5.2.3 Categorization in security research Categorization is not a goal in itself; it must bring some benefits to a field if it is to be meaningful. We believe that systematic categorization is important because

- the formation and application of a taxonomy enforces a structured analysis of the field,
- a taxonomy facilitates education and further research because categories play a major role in the human cognitive process,
- categories which have no members but exist by virtue of symmetries or other patterns may point out white spots on the map of the field and
- if problems can be grouped in categories in which the same solutions apply, we can achieve more efficient problem solving than if every problem must be given a unique solution.

Even if the meta-level discussion of categorization occupies the minds of linguists and logicians, most security taxonomies presented have shown a worrisome lack of insight even in classical categorization. When surveying existing categorizations of security threats, we noticed that very few clearly define the property on which a categorization is based. Therefore, in our categorization of intrusion techniques and results presented in Paper C, we suggested that the term *dimension* should be used for the property upon which categorization is based when we categorize abstract entities such as intrusions or vulnerabilities. For example, the categorization of identified security problems in Paper A uses the dimension of ‘cause’, while the categorization of security risks in Paper D is based on the dimension of ‘phases in the establishment of a system’.

Paper E presents a taxonomy of remedies. The fine-grained dimensions are fault location, remedy location, remedy provider and remedy impact. It is recognized that the categorization of a particular remedy will and should be site-specific rather than universal.

In Paper F, 30 types of intrusions are divided into 10 classes according to the dimension ‘traces left in different types of logs’. For the sake of brevity, we chose to describe only one member of each category in detail; the other members are only outlined. This conforms with prototype theory, because the category member described in detail was chosen because it is—in some sense—a better example of the category than were the other members.

6 Summary of papers

6.1 Part I: Analysis of the nature of security threats

6.1.1 Paper A: An analysis of a secure system based on trusted components

This paper presents a practical security analysis of a beta implementation of a commercial system based on existing trusted hardware components, such as advanced cryptographic building blocks. The system was designed to securely store and handle both sensitive and insensitive data records on individuals in such a way that it would be impossible for unauthorized parties to link sensitive records to the corresponding individuals. The analysis was performed by means of document reviews, interviews and some practical tests with the intention of finding and listing potential vulnerabilities for the knowledge of the design team. The vulnerabilities revealed are classified with respect to their cause, and possible remedies are discussed. The classification shows that the most important problem was that some system components were incorrectly handled as trusted. Finally, we observed that the problems were to a surprisingly high degree non-technical, reflecting organizational and management issues and human insufficiencies.

6.1.2 Paper B: Analysis of selected computer security intrusions: In search of the vulnerability

This paper presents an in-depth analysis of some selected computer security intrusions we have encountered during intrusion experiments and security analyses. The intrusions presented here illustrate the wide range of threats that owners and users of modern distributed computing systems must face. Several different dimensions of the intrusions are considered and discussed in detail, such as the flaw exploited in the intrusion, the cause of the presence of the flaw in the system, the method of attack, the initial result of the penetration, the possible implications and recommended remedies. It is argued that an intrusion is not feasible solely because of a single flaw, but is rather a function of a number of different flaws and characteristics of the system. This makes the job of detecting and preventing intrusions much more complicated.

6.1.3 Paper C: How to systematically classify computer security intrusions

This paper presents a classification of intrusions with respect to technique as well as to result. The taxonomy is intended to be a step on the road to an established taxonomy of intrusions for use in incident reporting, statistics, warning bulletins, intrusion detection systems etc. Unlike previous schemes, it takes the viewpoint of the system owner and should therefore be suitable for a wider community than that of system developers and vendors only. It is based on data from a realistic intrusion experiment, a fact that supports the practical applicability of the scheme. The paper also discusses general aspects of classification and introduces a concept called dimension. After having made a broad survey of previous work in the field, we decided to base our classification of intrusion techniques on a scheme proposed by Neumann and Parker in 1989 [50] and to further refine relevant parts of their scheme. Our classification of intrusion results is derived from the traditional three aspects of computer security: confidentiality, availability and integrity.

6.1.4 Paper D: A map of security risks associated with using COTS

The widespread use of commercial off-the-shelf (COTS) products in combination with increased internetworking calls for an analysis of the associated security risks. This paper presents a taxonomy of potential problem areas, illustrated by several examples. It can be used to aid the analysis of security risks when using systems that to some extent contain COTS components. Problems related to the integration of COTS components into a secure system are discussed using the privacy-oriented database system in Paper A as an example. The combination of Internet connectivity and COTS-based systems results in particular security problems, and we explain why not only the external threat but also the internal threat increases on the basis of experience from experiments that we have conducted. We also present an outline of a risk management philosophy addressing the problems presented in this article. In a separate sidebar, the so-called confinement problem and possible solutions are discussed.

6.2 Part II: Methods to improve system security

6.2.1 Paper E: The remedy dimension of vulnerability analysis

This work is aimed at supporting system and information owners in their mission to apply a proper remedy when a security flaw is discovered during system operation. A broad analysis of the different aspects of flaw remediation has resulted in a structured taxonomy that will guide the system and information owners through the remedy identification process. The information produced in the process will help in making decisions about changes to the system or procedures. A selected vulnerability that was able to be removed using three different remedies is used as an example.

6.2.2 Paper F: An approach to UNIX security logging

Off-line intrusion detection systems rely on logged data. However, the logging mechanism may be complicated and time-consuming, and the amount of logged data tends to be very large. To counter these problems, we suggest a very simple and cheap logging method, ‘light-weight logging’. It can be easily implemented on a Unix system, particularly on the Solaris operating system from Sun. It is based on logging every invocation of the *exec(2)* system call together with its arguments. We use data from realistic intrusion experiments to show the benefits of the proposed logging and in particular that this logging method consumes as few system resources as comparable methods while still being more effective.

6.2.3 Paper G: Detecting computer and network misuse through the production-based expert system toolset (P-BEST)

This paper describes an expert system development toolset called the Production-Based Expert System Toolset (P-BEST) and how it is employed in the development of a modern generic signature-analysis engine for computer and network misuse detection. For more than a decade, earlier versions of P-BEST have been used in intrusion detection research and in the development of some of the most well-known intrusion detection systems, but this is the first time the principles and language of P-BEST are described to a wide audience. We present rule sets for detecting subversion methods against which there are few defenses—specifically, SYN flooding and buffer overruns—and provide performance measurements. Together, these examples and measurements indicate that P-BEST-based expert systems are well suited for real-time misuse detection in contemporary computing environments. In addition, the simplicity of the P-BEST language and its close integration with the C programming language makes it easy to use while it is still very powerful and flexible.

6.2.4 Paper H: Designing IDLE: The intrusion data library enterprise

High quality, timely information on intrusions is crucial in the development, testing, tuning, and updating of intrusion detection systems (IDSs) and intrusion recovery systems. We present the Intrusion Data Library Enterprise (IDLE), a design and initial compilation of an extensible library of intrusion data that is efficiently parseable in both human-readable and platform-independent machine-readable forms. The IDLE format will be made available as a resource specifically for the intrusion detection community. IDLE will provide IDS developers and users with accurate field data for testing and tuning and, as new intrusion types are discovered, it will enable tools to automatically update rule sets and parameters.

7 Reflections

7.1 Identified hard problems

During the work presented in this thesis, I have seen the same type of basic security problems occur again and again. In this section, I take the opportunity to discuss my views on what I have identified as the most important problems to address in future security research.

7.1.1 The problem of highest privilege In a seminal paper from 1975, Saltzer and Schroeder presented eight primary and two secondary design principles for secure computer systems [58]. The principles are fundamental and should be compulsory reading for any computer professional, not only security people. It is historically interesting that the paper talks about firewalls and intrusion detection ('compromise recording') several years before any of these became fields of research and development. One of Saltzer and Schroeder's principles is the following [58]:

> Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur.

Unfortunately, the principle of least privilege is rarely applied in the systems we use today, and this is the cause of many security problems. We often find the inverse of the principle, which I have chosen to call *the problem of highest privilege*. It means that programs and users tend to have the highest set of privileges possible in the system, probably because it means that their operations will not be obstructed by any annoying security mechanisms. This is a little like fishing with explosives instead of traditional fishing tackle—it is efficient but dangerous (not only to the fish) and can have undesired side effects. There are several examples of this problem:

► **Example 7.1.1** In 1993, it was discovered that the X Window System terminal program *xterm* had a security flaw known as the 'xterm logging vulnerability' [10]. I have described this particular problem from different perspectives in Papers B, C, E and F. In summary, *xterm* is a large program, and the flawed version ran with constant administrator privileges, although such powers were necessary only for a small fraction of its duties. An attacker could trick the logging facility of *xterm* into creating an arbitrary new file or modify any existing file (for example, the system password file) by appending an arbitrary set of data to it. The problem of highest privilege is a direct cause of this vulnerability.

► **Example 7.1.2** Many network server programs, such as e-mail servers or Web servers, run with administrator privileges, although they often do not require such privileges except for very specific operations. Typical privileged operations include the use of network port numbers lower than 1024, writing in certain directories etc. The problem is that the server programs listen for network connections and carry out instructions sent to them on those connections. An attacker who can connect to the program and is able to trick it into performing certain illegal operations will effectively perform those operations with administrator privileges on the victim system.

▸ **Example 7.1.3** In the Microsoft Windows NT File System, user access to local system files and folders can be restricted to read-only permission to prevent accidental or intentional modification. However, many programs (including Microsoft Office 97) are written for Windows 95 or 98, which lack access control features, and the programs therefore assume that they can write to all local files and folders and do not work properly when the write operations are denied. Consequently, system files and folders must be opened for writing by the user or the user must be given administrative privileges for these programs to work. In both cases, we end up with the problem of highest privilege, and the security benefits of having access control features are effectively eliminated.

7.1.2 The problem of weakest link The statement that "a chain is only as strong as its weakest link" is particularly true in computer security. The attackers need only one traversable path through a protection barrier to reach their goal, while the defenders must make sure that no such path exists. This would be less of a problem if protection mechanisms were built in layers, resembling onions or Russian dolls, where the attackers would need to penetrate several independent layers to gain access to the protected resources. However, many systems have a 'security philosophy like an egg or a coconut, with a single-layer outer hard barrier protecting a soft, squishy core. A single weak point in that barrier will make it possible for attackers to penetrate the system, and attackers can be expected to search for and find such weak points. Designers who concentrate on their clever security mechanisms often find it "unfair" that someone attacks a much weaker part of the system. It is important to remember the nature of villains; by definition they do not play fair!

A basic design rule addressing this problem is to make sure that all links have the same strength. In fact, lopsided security is a dangerous waste of resources. For example, if passwords are protected by strong encryption when stored on disk, but are transmitted unencrypted on the network when used for remote access, only an unusually deranged attacker would try to break the encryption algorithm—all others would eavesdrop on the network and simply record cleartext passwords.

In the design of fault-tolerant systems, a number of techniques are used to avoid so-called *single points of failure*. Some efforts have been made to utilize such techniques to address this problem in security [17, 28], but there is probably much more that can be done in this area.

7.1.3 The denial of service problem If we again compare security to the reliability or fault tolerance disciplines, they all strive to uphold the *availability* of a system. The difference is that the availability aspect of security is about protecting the system from someone who actively tries to prevent the system from delivering the required service, while the others deal with random threats such as component failure. So-called denial of service attacks range from the very unsophisticated, such as cutting off the electric power or physically damaging the system, to the fairly advanced, such as utilizing flaws in communications protocols or operating systems to crash or anonymously overload the system. Availability is clearly the forgotten aspect of security, and only a few studies have been presented, for example [20, 43, 46]. By not giving the subject the attention it deserves, we are now in a dangerous situation:

- Systems do not usually have resource allocation procedures that can act fairly when misused. For example, an attacker masquerading as a number of users can easily occupy all resources.
- Security mechanisms are often designed to be fail-safe, that is, if the mechanism does not work properly, it will deny all access in order to guarantee integrity and confidentiality. It should be noted that, in the case of denial of service, this is exactly what the attacker wants.
- Designers of open services have no tradition of making their systems robust against denial of service attacks. There is no widespread knowledge in the world of "best effort" Internet services about how this should be done[6].

In applications in which strong authentication is required by users, denial of service is less of a problem, but there will always be applications in which the users are previously unknown or even forever anonymous to the service provider (current popular examples are anonymous FTP and the World Wide Web). Substantial effort must be expended to make sure that misusers cannot deny a service to benign users simply by overloading it with fraudulent accesses. Techniques used in the design of mission-critical real-time systems could probably inspire solutions to this problem. For example, an IDS could use watchdog timers and heartbeat messages to discover when a target or IDS component is unavailable or responds too slowly.

7.1.4 The Trojan horse problem: Trusting the untrustworthy

Too late-breaking news: The malicious code escaped from confinement and was immediately executed by the macroprocessor.

Basically, a computer processes, produces, stores, transfers and deletes data according to the instructions (the program) it is given. It is not far-fetched to state that you have a security problem if your data is valuable and the instructions you give your computer are untrustworthy, that is, they could have been authored or altered by a person with malicious intent. How do you know that every single instruction executed on your computer—whether the code is part of an application program, device drivers, the operating system, the firmware etc.—is trustworthy? The reader who has not yet realized the full implications of this problem is encouraged to stop reading for a moment and think about it.

[6]In fact, most services do not specify any quality of service, such as a maximum waiting time, and, as observed by Gligor [20], by definition, denial of service cannot take place under such circumstances because no service is promised. See also the discussion on security policy in Section 3.1.

The combination of insecure client hosts (typically personal computers running common commercial operating systems with disabled or missing access control mechanisms, suffering from the problem of highest privilege), internetworking and untrustworthy code is indeed very dangerous. Security researchers, including myself (see for example Paper D), have tried to warn the computer business and the public about this problem. Recently, there have been more tangible demonstrations, where potentially or directly malicious software has been released via the Internet. Program beasts with names such as “Melissa”, “Back Orifice” and “NetBus” haunt the networks of large organizations and individual home computers.

Open distribution of source code could make it easier to discover Trojan horses, but the problem would still exist. The majority of ordinary users are happy if they know enough about the system to *use* it and will never look at the source code. Even devoted experts have difficulties, owing to the size and complexity of many systems. The flip side of open source code distribution is that it could in fact make it *easier* to inject malicious code into software packages—the process would be simply to download the source code, modify whatever is desired and redistribute the modified package as source code or compiled executables to victims. The hollow wooden horse is yours; all you have to do is fill it with Greek warriors[7].

To address this problem, users must be provided reasonable technical defense measures against the threat of malicious code. The current argument from some major software manufacturers that it is the users’ own fault if they run untrustworthy code is very similar to the defense argument of the major car manufacturers when they were first accused of making unsafe cars: accidents are the drivers’ fault. Later, car manufacturers were convinced by regulations and market demand that cars should be made safer. Computer users have a certain responsibility, just like car drivers have a certain responsibility, but, in both cases, humans need technical support to achieve reasonable levels of security and safety. Should there be security regulations for software products? Will there be a market demand for IT products with better security?

In Paper D, we recommend a risk management approach consisting of several steps to counter the problem of untrustworthy components and related threats. The bottom line is that intrusion prevention must be made better but will never be good enough on its own and should be accompanied by intrusion detection and other damage-reducing mechanisms and procedures.

7.2 On the validity and accuracy of data and results

There are several concerns regarding the validity and accuracy of the data obtained from the student experiments. The first question is whether the students in our experiments constitute a good approximation of real attackers. In reasoning about security, the discussion often quickly reaches the point where the goal is to protect the system against the most skilled and powerful attacker in the world. The term “übercracker” [18] has been used for this picture of a diabolic, omnipotent adversary. We did not wish to investigate the übercracker, partly because most people

[7] I am indeed in favor of open source code distribution, but the described danger should be observed.

in the security community are already doing that. We wanted to see how ordinary computer users with academic training in computer science and engineering but no previous attacker experience would operate when they suddenly had a reason to attack the system on which they were working. For example, we found that unskilled attackers can indeed perform technically advanced attacks, thanks to so called *exploit scripts* which they download from the Internet (see Paper D).

The second question is how well the students' reports correspond to reality. The set of actions actually carried out by the attackers in our experiments is largely in agreement with the set of actions documented by the attackers in their reports, but the two are not identical. There are indeed actions reported that we seriously doubt were ever performed on the system, for example types of intrusions to which we know that the system was not vulnerable. This became painfully clear to the students who later tried to confirm the reports in the intrusion analysis exercise. It is also likely that there are actions that the attackers performed but never reported to us. Why is there not a perfect match between the two sets? For the first case, we do not really know what reason the students have for exaggerating their results. The grading of the reports was not based on their successes but on their efforts and their analysis of their work. We had also made it clear that their reports would be used as research material, and it is therefore disappointing and worrisome that some of them tried to polish their results. For the second case, we can only hope that actions possibly left out of the reports were not reported because the students considered them insignificant and not because they wanted to hide something from us.

Finally, there is the problem of using a relatively large group of humans as "guinea pigs" at an engineering department. Our expertise is in technology, not in the behavioral sciences. Factors not related to technology can greatly affect the results, for example of how instructions are interpreted, what motivation the participants have, whether they obey the rules of the experiment, whether they tell the truth etc. This is probably an area in which we could benefit from consulting external expertise.

As for the recorded FTP data, it is of high validity in the sense that it is "real". The disadvantage is that we have little or no conception of the true actions and intentions producing the data. For example, we would not know about attacks that for some reason were not recorded.

The DARPA evaluation data is definitely accurate because it is produced artificially with an exact knowledge about the events producing the data. However, it can be argued that, because it is based on the normal traffic of a military base, it might be difficult to generalize to other environments such as corporate or academic sites.

Figure 2 depicts a relative grading of our three different data sources in two dimensions: how well the data is believed to represent real security threats in general (*validity*) and how well the data can be trusted to be a true representation of the actual events that took place (*accuracy*). We can see that the use of supervised students as a data source is a compromise both in terms of validity and accuracy compared to the use of real attackers or pure simulation.

Validity or generalization of data and results is a general problem for the observing researcher. It is difficult to determine to what degree data produced under very specific circumstances represents the general case. For example, do real insiders

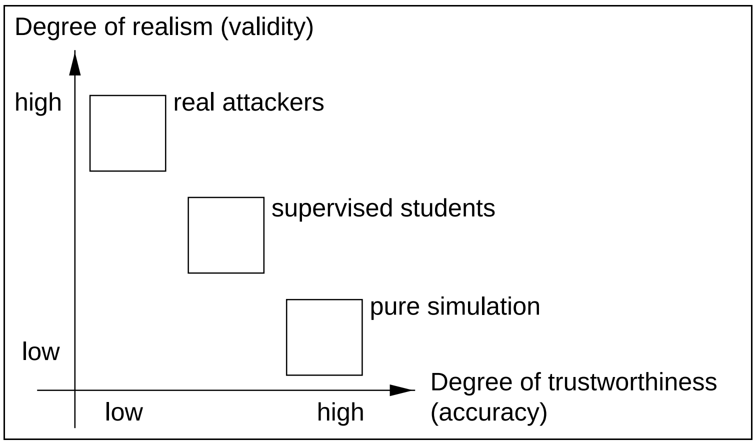

Degree of realism (validity)
high
low
real attackers
supervised students
pure simulation
low
high
Degree of trustworthiness (accuracy)

Figure 2. Relative validity and accuracy of intrusion data sources.

in general typically behave like our students, or is it only insiders who recently graduated from university who have this *modus operandi*, or are there in fact no real attackers who behave like our supervised students? The solution is typically to make extensive studies of “the real thing”, but then we have the accuracy problem when dealing with real intruders. Like Heisenberg, we must focus on one dimension at a time.

7.3 Ethical aspects

In this section, we take an external view and discuss our research methods and their consequences as they are seen outside the security research community. This discussion first appeared in [41].

7.3.1 The hacker school One concern that was voiced by several sources when our student intrusion experiments came to public knowledge was that we were training university students to become computer criminals. This was amplified by some reports in newspapers and television. A message that reached the public, and still haunts us several years later, was that Chalmers had attacked hospital computers. The journalistic logic behind this conclusion was that, because we had performed our first experiment on a UNIX system and the hospital administration in a nearby county had this type of system, hospitals could probably be attacked in a similar way and, *ergo*, Chalmers had trained students to attack hospitals.

What are the facts in this case? In the intrusion experiments, we refrained entirely from training students in attacking the system, which was necessary to ensure that the data collected was valid for security modeling. In fact, some students complained about the lack of any hints or instructions. We simply gave them access to a computer network which they were allowed to attack for a limited period of time. We also gave them access to the Internet, where they could seek information. In 1993, this was not something that students could normally arrange on their own. Today, any student can afford to set up a computer network in his or her home, consisting for example of a number of “old” PCs with an Intel 486 processor running Linux, and Internet access is something that most people take for granted today.

All along, we have informed the students about what constitutes computer crime according to national laws and why certain behavior is inappropriate or illegal.

7.3.2 Informed consent An important concept in research ethics is *informed consent*, which means that experiment participants should be informed of all possible consequences and risks of the experiment, that they should understand that information and that they should be given the possibility to refuse participation. In the intrusion experiments, the students who played the role of attacker were of course informed about the experiment but were not informed that they could refuse and be given another assignment instead, simply because we assumed that they all wanted to participate.

The situation was different for the students who were ordinary users of the system. They were not informed about the experiments because we did not want them to be more concerned about security than they normally are. Unknowingly, they played the role of victims. Some were deceived by forged e-mail to send the attackers their passwords, others had their passwords revealed in other ways and, worst of all, some passwords to other systems were monitored by the attackers when lab computers were used as terminals for remote access. This is perhaps the most questionable part of the experiment, with a trade-off between realism and ethical concerns. The users were not as easily fooled in the later experiments because they had heard about our previous activities. In fact, any computer malfunction was blamed on the security experimenters.

In the intrusion detection exercise described in Paper G, we wanted primarily to use real, recorded data to make the students feel the realism and relevance of their assignment. Users accessing the monitored FTP server had been warned through a login banner message that their activities were monitored, and the passwords for non-anonymous users were never recorded. Still, there can be innocent users whose transactions appear suspicious in the log file or other privacy concerns. Again, there is a trade-off between realism and ethics.

7.3.3 Related discussions on ethics In a discussion of the ethical aspects of spreading information on methods for computer crime, Parker claims that the intent of the publisher is what matters [53]. This view is shared by Spafford [61]. If the intent is to raise awareness and protect systems, then it is ethical (and legal). If the intent is to encourage people to attack systems, then it is unethical and probably illegal.

At the 21st National Information Systems Security Conference in Arlington, Virginia in October of 1998, there was a panel discussion entitled “Do attack/defend exercises belong in the classroom?”. It is interesting to note that all panelists from academia were in favor of such exercises as a part of security education, and it was difficult for the organizers to find a panelist with the opposite opinion. One concern brought up in the discussion was that students could use skills acquired in the exercise for evil purposes. This is however a general concern that is not restricted to computer security; any tool or skill can be used for evil purposes. A similar view is presented by White and Nordstrom [67], who claim that it would be more dangerous *not* to educate future system administrators in the details of attacking techniques because they would otherwise be “sitting ducks” for attackers who possess these skills.

7.4 Contributions in perspective

This section summarizes the main contributions of the work presented in this thesis. Our results are discussed in the light of other research in the field.

7.4.1 Intrusion analysis We have conducted extensive analyses of the characteristics of the phenomenon known as intrusions. The specific dimensions of intrusions that we have focused on are those that are fundamental for detection and diagnosis: intrusion technique and result (Paper C) and observable traces (Paper F). To perform manual or automatic detection and/or diagnosis of a phenomenon, a prerequisite for success is that you know what to look for, and it is likely that you will be more successful the more knowledge you have of your target. In the first published textbook on intrusion detection from a research perspective, Edward Amoroso quotes our taxonomy from Paper C and states that “The reason that intrusion taxonomies and refinements such as [the one presented in Paper C] are so valuable to intrusion detection system designers and users is that they provide insight into the target objects being detected by these systems, particularly in the area of indicators of intrusion” [1, p. 107]. In Paper H, we propose how intrusions may be documented in a database format for the rapid sharing and archival filing of knowledge produced in intrusion analysis aimed at detection and diagnosis.

7.4.2 Studies of insider techniques In the student intrusion experiments, the students acted as insider attackers. Insiders typically have other motives and skills than do specialized outsider crackers, and we have observed that insiders are very much helped by so-called exploit scripts which are attack tools published on the Internet (see Paper D). Technically speaking, it can be argued that an outsider becomes an insider as soon as he or she has gained remote access to an account on the system, but there are also issues of social interaction and physical presence. For example, the attack based on manipulating the boot process described in Paper C requires physical access to the workstation console. An insider can make real-world ‘out of band’ observations that can be helpful in guessing passwords or selecting accounts that are likely to have extraordinary privileges, for example. Even if vendors of commercial IDSs often refer to the outsider threat in their advertising, the difficulty associated with the prevention of insider attacks is one of the major traditional motivations behind IDS research [15] and, consequently, studies of insider behavior are at least as important as studies of outsiders. Many of the attacks suggested in Paper A represent the insider threat, and all our results based on the student experiments (see Papers B, C, D and F) apply primarily to insiders, although several of the attacks are also typical for outsiders as observed in other studies [24, 63].

7.4.3 Risk management techniques: accepting the inevitable In our work on analyzing security vulnerabilities presented in Papers A and B, we show that vulnerabilities are typically caused by a combination of technical and “soft” issues such as human-computer interaction and organizational concerns. We also confirm observations made by other researchers [5, 29] that vulnerabilities are inevitable in systems in operation. Therefore, we propose risk analysis (Paper D) and vulnerability remediation (Paper E) strategies. We recommend system owners to apply such risk management methods when relying on systems based on commercial components for security-critical applications. In fact, today, it is difficult to find systems that are *not* based on commercial components, even in military applications. Risk management techniques have thus become an essential complement to traditional risk avoidance methods, and we see it as our mission to transfer this knowledge from the research community to system owners in industry and other organizations.

7.4.4 Intrusion result detection Common objections to misuse detection are that it requires detailed and specific *a priori* knowledge about every type of intrusion it is capable of detecting and that previously unknown attacks and variations of known attacks will go undetected. This is true if we only consider attack *techniques*, but if we can step up one level on the abstraction scale and look instead at the *result* of the attack (the breach), these objections become less valid (see Paper C for an analysis of intrusion techniques and results). In addition, rules detecting results will not reveal methods that can be used to design an attack tool, because such rules do not specify how the detected result may be achieved. Let us illustrate the above points with some examples:

► **Example 7.4.1** Let us assume that we have a misuse detection system that analyzes network traffic going to and from a server for anonymous file transfer (FTP). This IDS could be configured to warn us whenever an anonymous user issues a destructive command, for example a command to delete a file. If the warning indicates that the command was unsuccessful, we might be under attack but the system may still be undamaged. However, if the destructive command was successful, our IDS will tell us that the preventive mechanisms were penetrated (or perhaps circumvented) although we need not know *how* it was done.

► **Example 7.4.2** If we have an IDS that monitors a host audit trail, the IDS may be set to detect the execution of commands with administrator privileges by a normally unprivileged user. Alternatively, it could detect that the privileged state was not reached by using an appropriate procedure. Again, the misuse detection system detects the result of the intrusion and does not need to know the method used to reach the result.

The Information Systems Technology Group of MIT Lincoln Laboratory have developed an IDS method called "bottleneck verification" [13], which detects when a transition from an unprivileged state to a privileged state occurs and an approved procedure is not used for the transition. Bottleneck verification would typically detect the situation described in Example 7.4.2 above. If we picture an intrusion as an attack which causes a transition from system state s_1 to s_2, where s_2 represents a breach or compromised state, then traditional misuse detection detects the attack, bottleneck verification detects that an approved state transition mechanism was not used while result detection detects that the system has reached the illegal state s_2.

Result detection, in summary, has the following benefits compared with traditional methods for misuse detection:

- Rules can be independent of vulnerabilities.
- Rules can be independent of attack methods.
- Rules can to a certain extent be independent of the characteristics of specific operating systems.

The disadvantage of result detection is that it detects only completed intrusions and does not allow the opportunity to act before any damage is done, which for example state transition analysis can do [54].

7.4.5 Verified usability for IDS tool In Paper G, we studied the usability of a tool for building IDS analysis components, in terms of performance, ease of use, expressive power and adaptability to different event data streams and intrusion types. The expert system tool P-BEST was used to build analysis components in the MIDAS [59], IDES [42] and NIDES [2] systems and is currently being used in EMERALD [55]. By letting students use P-BEST for an IDS lab assignment, we were able to verify that the system was easy for non-experts to use. Furthermore, in the EMERALD project, we designed P-BEST detection rules for different types of intrusions, requiring analysis of both host audit trails and recorded network traffic, applied these rules to large data sets and measured the resulting performance. We thereby showed that P-BEST is well suited for constructing real-time misuse detection components of IDSs that are targeted for contemporary computing environments.

8 Conclusions and directions for future work

The knowledge acquired and presented in this thesis contributes to our understanding of the fundamental technical aspects of computer misuse, both in terms of manifestation of misuse and possible countermeasures, and is a step on the road to improved security in the systems we depend on today and those we will depend on in the future. Our analyses of vulnerabilities in systems based on commercial components show that security problems have intricate causes and that it is extremely difficult, if not impossible, to build systems that are free from vulnerabilities. It is concluded that risk management techniques are necessary when risk avoidance is insufficient, and we have specifically studied vulnerability remediation and intrusion detection as examples of such techniques. Our systematic analysis of intrusion techniques and results, together with our studies on collection and documentation of observable intrusion traces, pave the way for efficient and effective intrusion detection systems.

We can and should always collect more data on intrusions and vulnerabilities, provided that we respect ethical and privacy issues. As shown in this thesis, although it is not obvious how we should best catalogue such data, the data can be of immense value to those who develop security protection mechanisms. It is often necessary also to collect data on normal system events together with the intrusion data to be able to analyze the relative frequency of intrusions etc. It would indeed be interesting to continue development on a common format for documentation and distribution of attack signatures.

How should intrusion detection systems be designed and implemented to be truly useful in large integrated networks? There are a number of concerns that must be kept in mind, including ease of use, scalability, results correlation, false alarm suppression, survivability and interoperability. Research has only just been initiated for some of these issues, and there is much work to be done.

The remaining difficult problems that were identified in this thesis—highest privilege, weakest link, denial of service and Trojan horses—are all important to address in future research. The market requires technical solutions that solve these problems without adding cost to products or adding to the workloads of users or administrators. This is indeed challenging.

References

[1] Edward Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Traps, Trace Back, and Response*. Intrusion.Net Books, Sparta, New Jersey, 1999.

[2] D Anderson, T Frivold, and A Valdes. Next-generation intrusion-detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1995.

[3] James P Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, U.S. Air Force Systems Command, Electronic Systems Division, October 1972. In [8].

[4] James P Anderson. Computer security threat monitoring and surveillance. Technical report, James P Anderson Co., Box 42, Fort Washington, PA 19034, USA, April 15, 1980. In [8].

[5] Ross J Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[6] C R Attanasio, P W Markstein, and R J Philips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[7] Stefan Axelsson. Research in intrusion-detection systems: A survey. Technical Report 98-17, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, December 15, 1998.

[8] Matt Bishop, editor. *History of Computer Security Project CD-ROM*. Number 1. Department of Computer Science, University of California at Davis, Davis, CA 95616-8562, USA, October 1998. Available from *http://seclab.cs.ucdavis.edu/projects/history*.

[9] British Standards Institution. *Code of Practice for Information Security Management*, 1995. BS 7799.

[10] CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. *xterm Logging Vulnerability*, November 11, 1993. CERT Advisory CA-93:17.

[11] Common Criteria Implementation Board. *Common Criteria for Information Technology Security Evaluation*, May 1998. Version 2.0. See also ISO/IEC 15408.

[12] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT users guide. Technical Report TR-96-050, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, September 4, 1996.

[13] Robert Cunningham et al. Intrusion detection research at Lincoln Laboratory. In *Proceedings of the UC Davis Intrusion Detection and Response Data Sharing Workshop*. Department of Computer Science, University of California at Davis, Davis, CA 95616-8562, USA, July 15, 1998.

[14] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, April 1999.

[15] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[16] Dorothy E Denning and Peter G Neumann. Requirements and model for IDES—a real-time intrusion detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, 1985.

[17] Yves Deswarte, Laurent Blain, and Jean-Charles Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 110–121. IEEE Computer Society Press, Los Alamitos, California, May 20–22, 1991.

[18] Dan Farmer and Wietse Venema. Improving the security of your site by breaking into it. Posted on *comp.security.unix* and several other Usenet newsgroups, December 1993.

[19] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Oakland, California, May 9–12, 1999. IEEE Computer Society Press, Los Alamitos, California.

[20] Virgil D Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, SE-10(3):320–324, May 1984.

[21] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th USENIX UNIX Security Symposium*, San Jose, California, July 22–25, 1996. USENIX Association.

[22] Jani Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In Yves Deswarte et al., editors, *Computer Security – Proceedings of ESORICS 92*, volume 648 of *LNCS*, pages 435–450, Toulouse, France, November 23–25, 1992. Springer-Verlag.

[23] Paul Helman and Gunar Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, September 1993.

[24] John D Howard. *An Analysis of Security Incidents On The Internet 1989–1995*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 7, 1997.

[25] Erland Jonsson. *A Quantitative Approach to Computer Security from a Dependability Perspective*. PhD thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1995.

[26] Erland Jonsson. An integrated framework for security and dependability. In *Proceedings of the New Security Paradigms Workshop*, pages 22–29, Charlottsville, Virginia, September 22–25, 1998. ACM Press, New York.

[27] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[28] Mark K Joseph and Algirdas Avižienis. A fault tolerance approach to computer viruses. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 52–58, Oakland, California, April 18–21, 1988. IEEE Computer Society Press, Los Alamitos, California.

[29] Jay J Kahn and Marshall D Abrams. Contingency planning: What to do when bad things happen to good systems. In *Proceedings of the 18th National Information Systems Security Conference*, pages 470–479, Baltimore, Maryland, October 10–13, 1995. National Institute of Standards and Technology/National Computer Security Center.

[30] Calvin Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach*. PhD thesis, University of California at Davis, 1996.

[31] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Tenth Annual Computer Security Applications Conference*, pages 134–144, Orlando, Florida, December 5–9, 1994. IEEE Computer Society Press, Los Alamitos, California.

[32] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, California, May 4–7, 1997. IEEE Computer Society Press, Los Alamitos, California.

[33] Ivan V Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, West Lafayette, Indiana, May 1998.

[34] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1995.

[35] R D Lackey. Penetration of computer systems an overview. *Honeywell Computer Journal*, 8(2):81–85, 1974.

[36] George Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. The University of Chicago Press, Chicago, 1987.

[37] Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, 1992.

[38] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 120–132, Oakland, California, May 9–12, 1999. IEEE Computer Society Press, Los Alamitos, California.

[39] Richard R Linde. Operating system penetration. In *Proceedings of the National Computer Conference*, volume 44 of *AFIPS Conference Proceedings*, pages 361–368, Anaheim, California, May 19–22, 1975. AFIPS Press, Montvale, New Jersey.

[40] Ulf Lindqvist. *Observations on the Nature of Computer Security Intrusions*. Licentiate thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, December 1996.

[41] Stefan Lindskog, Ulf Lindqvist, and Erland Jonsson. IT security research and education in synergy. In Louise Yngström and Simone Fischer-Hübner, editors, *Proceedings of the IFIP TC11 WG 11.8 First World Conference on Information Security Education (WISE 1)*, number 99-008 in DSV Report Series, pages 147–162, Kista, Sweden, June 17–19, 1999. Department of Computer and System Sciences, Stockholm University/Royal Institute of Technology, Sweden.

[42] Teresa F Lunt, R Jagannathan, Rosanna Lee, Alan Whitehurst, and Sherry Listgarten. Knowledge-based intrusion detection. In *Proceedings of the Annual AI Systems in Government Conference*, pages 102–107, Washington, D.C., March 27–31, 1989. IEEE Computer Society Press, Los Alamitos, California.

[43] Jonathan K Millen. A resource allocation model for denial of service. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 137–147, Oakland, California, May 4–6, 1992. IEEE Computer Society Press, Los Alamitos, California.

[44] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Institut d'Informatique, University of Namur, Belgium, September 1997.

[45] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[46] Roger M Needham. Denial of service: An example. *Communications of the ACM*, 37(11):42–46, November 1994.

[47] Peter G Neumann. Computer system security evaluation. In *Proceedings of the National Computer Conference*, volume 47 of *AFIPS Conference Proceedings*, pages 1087–1095, Anaheim, California, June 5–8, 1978. AFIPS Press, Montvale, New Jersey.

[48] Peter G Neumann. *Computer-Related Risks*. ACM Press and Addison-Wesley, New York, 1995.

[49] Peter G Neumann. Practical architectures for survivable systems and networks: Phase-one final report. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, January 28, 1999.

[50] Peter G Neumann and Donn B Parker. A summary of computer misuse techniques. In *Proceedings of the 12th National Computer Security Conference*, pages 396–407, Baltimore, Maryland, October 10–13, 1989. National Institute of Standards and Technology/National Computer Security Center.

[51] Office for Official Publications of the European Communities. *Information Technology Security Evaluation Criteria*, June 1991. Version 1.2.

[52] Tomas Olovsson. *Practical Experimentation as a Tool for Vulnerability Analysis and Security Evaluation*. PhD thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1995.

[53] Donn B Parker. Colleagues debate Denning's comments. *Communications of the ACM*, 34(3):33–41, March 1991. Reprinted in *Ethics and Computing* by K. W. Bowyer, IEEE Computer Society Press, Los Alamitos, California, 1996.

[54] Phillip A Porras and Richard A Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, pages 220–229, San Antonio, Texas, November 30–December 4, 1992. IEEE Computer Society Press, Los Alamitos, California.

[55] Phillip A Porras and Peter G Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, October 7–10 1997. National Institute of Standards and Technology/National Computer Security Center.

[56] John Rushby. Critical system properties: Survey and taxonomy. Technical Report CSL-93-01, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1993. Revised February 1994.

[57] Jerome H Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[58] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[59] Michael M Sebring, Eric Shellhouse, Mary E Hanna, and R Alan Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, pages 74–81, Baltimore,

Maryland, October 17–20, 1988. National Institute of Standards and Technology/National Computer Security Center.

[60] Stephen E Smaha. Haystack: An intrusion detection system. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Orlando, Florida, December 12–16, 1988. IEEE Computer Society Press, Los Alamitos, California.

[61] Eugene H Spafford. Are computer hacker break-ins ethical? In Deborah G Johnson and Helen Nissenbaum, editors, *Computers, Ethics & Social Values*, pages 125–135. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[62] Robert E Stepp and Ryszard S Michalski. Conceptual clustering of structured objects: A goal-oriented approach. *Artificial Intelligence*, 28(1):43–69, 1986.

[63] Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, May 1988.

[64] Brian Tung. The Common Intrusion Detection Framework (CIDF), June 22, 1999. *http://gost.isi.edu/cidf/*.

[65] H S Vaccaro and G E Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, California, May 1–3, 1989. IEEE Computer Society Press, Los Alamitos, California.

[66] Willis H Ware. Security controls for computer systems (U): Report of Defense Science Board Task Force on Computer Security. Technical report, The RAND Corporation, Santa Monica, California, February 11, 1970. In [8].

[67] Gregory White and Gregory Nordstrom. Security across the curriculum: Using computer security to teach computer science principles. In *Proceedings of the 19th National Information Systems Security Conference*, pages 483–488, Baltimore, Maryland, October 22–25, 1996. National Institute of Standards and Technology/National Computer Security Center.

[68] Lotfi A Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, June 1965.

[69] Lotfi A Zadeh. A note on prototype theory and fuzzy sets. *Cognition*, 12:291–297, 1982.

This page is intentionally left blank.

Part I

Analysis of the nature of security threats

This page is intentionally left blank.

Paper A

An Analysis of a Secure System Based on Trusted Components

In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS ’96)*, pp. 213–223, Gaithersburg, Maryland, June 17–21, 1996.

This page is intentionally left blank.

An Analysis of a Secure System Based on Trusted Components

Ulf Lindqvist Tomas Olovsson[*] Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{ulfl, erland.jonsson}@ce.chalmers.se

Abstract

This paper presents a practical security analysis of a beta implementation of a commercial system based on existing trusted hardware components, such as advanced cryptographic building blocks. The system was designed to securely store and handle both sensitive and insensitive data records on individuals in such a way that it would be impossible for unauthorized parties to link sensitive records to the corresponding individuals. The analysis was performed by means of document reviews, interviews and some practical tests with the intention of finding and listing potential vulnerabilities for the knowledge of the design team. The vulnerabilities revealed are classified with respect to their cause, and possible remedies are discussed. The classification shows that the most important problem was that some system components were incorrectly handled as trusted. Finally, we observed that the problems were to a surprisingly high degree non-technical, reflecting organisational and management issues and human insufficiencies.

1 Introduction

The target system of this study was a beta implementation of a database application designed to deliver high-assurance security services in order to protect the privacy of the recorded individuals. It was developed by a subsidiary of a major computer manufacturer in cooperation with a small local enterprise, and its design is based on trusted components and particularly on a patent [26]. The system is intended to be used for personal registers, such as government or municipal services, offices in health care, social care and other public services, which have a need to record data on individuals. Indeed, the developers claim the system to be suitable

[*]Author's present address: Carlstedt Research & Technology AB, Stora Badhusgatan 18-20, SE-411 21 Göteborg, Sweden, tomas@carlstedt.se

for most non-military environments. The major design goal was to make it virtually impossible to link a sensitive record in the database to an individual without being properly authorized. Similar systems with this purpose have been presented previously, see for example Brandt *et al.* [6].

We were contacted in an early phase of the development process and were asked by the system developers to analyse the security of the system, both by scrutinizing the design documents and by attempting to perform intrusions—or “attacks”—on a beta installation of the system using our earlier experience of such experiments [7, 11, 22]. Our objective was to seek answers to the following questions:

- Is it possible to build a secure application based on PC workstations connected to one or more database servers, with the help of commercially available security components? If not, what are the major obstacles?
- In what areas are vulnerabilities most likely to be found and what are the underlying causes of the vulnerabilities present in the system?

Our approach is similar to a Tiger Team analysis [5, 10, 13], but has been developed further by means of classifying and analysing the results with the intention of drawing some general conclusions. Thus, Section 2 presents a system overview, and Section 3 briefly describes our analysis approach. In Section 4, a taxonomy of security problems is first defined, and the design flaws and implementation problems we found are then presented in detail and classified according to the previously defined taxonomy. Section 5 discusses the results obtained from the analysis, and our conclusions are presented in Section 6.

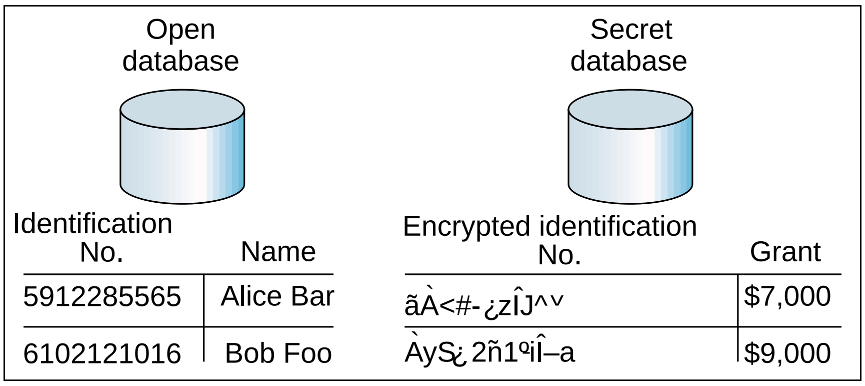
We would like to point out that the system is presented in this report as it was presented to us, and that we do not necessarily advocate the design trade-offs made by the designers. Our task was merely to find problems and to present them to the developers, so that proper improvements could be considered.

2 System overview

2.1 Logical structure

In many countries, individuals are assigned a unique number, a “personal number”, or “Social Security number”, which is used as a universal record identifier in various registers as well as in this application. To provide a database system with two different levels of security, the system utilizes separation, both by partitioning and by encryption. The basic idea is to keep two separate databases, as shown in Figure 1. The two databases can reside on the same or separate servers.

- The *open database* contains publicly available data such as name, address etc., with plaintext personal numbers as record identification fields.
- The *secret database* contains sensitive data with *encrypted* personal numbers as record identifiers. This DB is used by authorized officials to read and update records on individuals. Since some record fields may be sensitive in the sense that they might give enough information to reveal the individual behind the record, such fields will be encrypted in the database.



Identification No.	Name
5912285565	Alice Bar
6102121016	Bob Foo

Encrypted identification No.	Grant
ãÀ<#-¿zÎJ^v	$7,000
ÀyS¿2ñ1ºiÎ–a	$9,000

Figure 1. The two databases with simple examples of records.

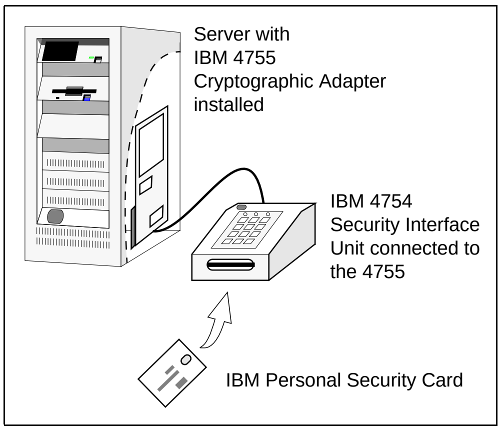

Server with IBM 4755 Cryptographic Adapter installed
IBM 4754 Security Interface Unit connected to the 4755
IBM Personal Security Card

Figure 2. Server configuration.

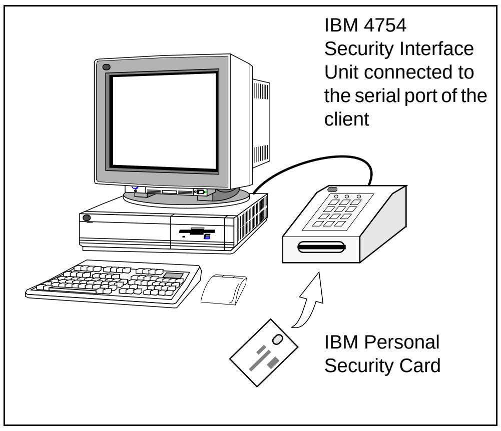

IBM 4754 Security Interface Unit connected to the serial port of the client
IBM Personal Security Card

Figure 3. Client configuration.

2.2 System platform

The target system of the present analysis was a very specific system implementation, called *the beta installation*, running an example application, called *the pilot register application*. The beta installation consists of one client and one server, both IBM PS/2 personal computers running OS/2 2.11, where OS/2 LAN Server is used for network communication. The system is based on IBM's Transaction Security System (TSS) [1] for cryptographic functionality.

The server is equipped with an IBM 4755 Cryptographic Adapter (CA, a card installed in the host and connected to the system bus) to which an IBM 4754 Security Interface Unit (SIU) is connected. The SIU is an external device used for insertion of an IBM Personal Security Card which is a credit-card-sized smartcard with memory and encryption functions. See Figure 2.

The client has an SIU connected to its serial port. There is no other special security hardware installed in the client. The SIU is used in all cryptographic operations such as user authentication and encryption of messages. To protect clients against theft, no information is stored on client hard disks by the system. See Figure 3.

The databases are implemented in DB2/2 with SQL language extensions. All cryptographic routines use the API bundled with the TSS [14, 15]. Examples of API services include data encryption and decryption of record fields, message authentication code (MAC) generation and verification, key management functions etc. The intention is for all these mechanisms and precautions to be sufficient to protect the clients, the network and the databases, and thereby maintain the stipulated high degree of security.

3 Approach

3.1 Security policy and threat assessment

The developers' intention was to make the system resistant to many different kinds of attackers, ranging from dishonest or disgruntled current or former employees to other organisations and even foreign governments. The developers summarized the comprehensive security policy of the system in two statements:

i) *Confidentiality.* Only authorized users should be able to link records in the Secret DB to a single individual.

ii) *Integrity.* Only authorized users should be able to modify records in a meaningful way.

In addition, this identification and modification of records should be possible only by using the services provided by the system, and not for example by having physical access to the databases or the communications network. The area where the client machines and the network are located should be considered as totally insecure. Servers can be protected against manipulation with burglar alarms, even though they can not be completely protected against theft.

3.2 Modus operandi

A popular approach in penetration analysis is the Flaw Hypothesis Methodology (FHM), which is recommended in the Orange Book [27]. The FHM consists of four main stages [18]:

i) Knowledge of system control structure

ii) Flaw hypothesis generation

iii) Flaw hypothesis confirmation

iv) Flaw generalization

Because of the limitations described in Section 3.3 below, we could not carry out the FHM completely. Instead, we used an *ad hoc* approach very similar to the FHM and its four stages.

To gain a knowledge of the system and its control structure, we performed document reviews, interviews with members of the development team and some practical tests in the beta installation environment. As this part of the analysis went on, we tried to think of all kinds of security problems in the system, that is we generated flaw hypotheses. We discussed the hypotheses among the members of the analysis team and interviewed the design team when things were unclear. We did not actually try to exploit any of the problems we found, but were content when we had convinced ourselves and the design team that what we found was really security problems. For every problem, we tried to see whether it could be generalized, for example if a problem found in a client was also present in the server.

3.3 Limitations

We did not have access to the source code of the system. Our specific questions on source code level were answered by the developers, although an in-depth analysis of the source code is of course necessary for a complete security evaluation of the system. The personnel resources and the time at our disposal were limited, as was the possibility to perform realistic attacks on the system.

4 Security analysis

4.1 Introduction

For every problem found, we were interested in identifying three characteristics: First, possible *points of attack*, that is to identify certain system components that seem to be more vulnerable than other components; second, to find out how the attack would be performed, that is the corresponding *means of attack*, which includes a description of the necessary methods and resources needed to perform a specific attack; and, third, to find the *cause* of the exploited problem.

We have classified the problems into four classes so that all problems in the same class have the same generic cause. The classes are:

A: *Misdirected trust*. The assumption that some system components need not be trusted was discovered to be false.

B: *Misuse of security mechanisms*. Cryptographic and other security functions are not used in the intended way or are used without considering implementation constraints.

C: *Implementation deviations from specification*. The actual implementation of the system does not follow the intentions of the original system specification.

D: *Oversights and bugs*. Important issues with influence on the overall system security have been overlooked in the design work.

4.2 Cause A: Misdirected trust

According to the system specification, some units, such as the clients and the interconnection network, were allowed to be "insecure" and did not need to be trusted, that is no specific security mechanisms were foreseen for them. The assumption was that the specified degree of security for the full system could be enforced by means of the tools and mechanisms present in other units. The following examples show that this assumption was incorrect.

4.2.1 Descriptions of problems

Problem A1 In the clients, it is relatively easy to install a Trojan horse which will monitor all records that a legal user sends or receives from the servers. The Trojan horse can do this either by scanning the memory, intercepting the network interface or taking repeated snapshots of the screen. All retrieved information can be sent to another user on the network or stored somewhere for delivery at a later point in time.

Problem A2 To further refine the attack described in Problem A1, an attacker can install a Trojan horse which will send its own requests to the server to retrieve information. From the server's point of view, this Trojan horse will act on behalf of the authorized user; it will use the user's smartcard and talk to the database server as if it were authorized. The system relies on the client software integrity check of the DBMS, but that mechanism is very simple and easy to bypass (the mechanism is designed to help the system administrator make sure that all clients use the same version of the application program, not to prevent Trojan horses).

Problem A3 Since smartcard readers (SIUs) are connected to the PC through a serial port, it is possible to connect a second cable to the reader and to use an authorized user's card from another computer. This arrangement would give the attacker the same possibilities as in Problem A2, but from another computer located elsewhere.

Problem A4 It is possible to install any kind of software and/or hardware in a client (or server) to bypass the use of encryption hardware, for example in favour of simpler algorithms. This could be done in a number of ways. For example, if the cryptographic API is dynamically linked to the application, the runtime linker could be tricked into linking a library provided by the attacker instead of the proper one.

Problem A5 The virtual memory swap file of OS/2 may contain sensitive information. The application has no control over what information is transferred to this file. It is possible for an attacker to collect information by searching this file, which can be done long after the authorized user has finished working.

Problem A6 An attacker with physical access to a server can make arbitrary changes to existing records in the data base after breaking the relatively simple protection system of the database management system (DBMS). The hardware security devices are not involved in this operation.

Problem A7 The log files of the DBMS must be treated with care. They contain, among other things, information about the origin of records, which could be used to correlate records in different databases. Again, a program could collect such information and store it somewhere else or send it over the network. It is also worth noting that, with special equipment, it is possible to recover information on disks several generations back.

Problem A8 During the process of initialization the CA, and at registration and initialization of users' smartcards, floppy disks with files containing unencrypted encryption keys are inserted. A Trojan horse can capture these keys during the initialization session. The captured information could be used for example to create new user identities or copies of existing smartcards. The designers intended to use a dedicated host for the administration of users' cards but, in the beta installation, the database server was used for this task.

Problem A9 An eavesdropping attacker can read all data fields (except for the record identifier, which is contained in the encrypted transaction header) in a transaction message sent between a client and a server. This is because the encryption of data fields takes place in the server just before fields are written to the database, and decryption is also done in the server immediately after fields are read. In both cases, *plaintext data is sent across the network*. Since the reason for performing such encryption is to hide sensitive data in records, this could be used to capture such data and possibly also to link the data to a single individual.

Problem A10 When a transaction message is sent from a client to the server holding the Secret DB, the encrypted transaction header begins with a timestamp as a defence against replay attacks. Since distributed clocks in a network usually cannot be perfectly synchronized, the use of timestamps must allow for a window of time. An attacker could use this window for a replay attack, in which the record identification field is replayed without alteration, but where the data field is altered or new fields are added. Since the timestamp is within the window, the transaction will be granted by the server.

Data sent from the server to the client can probably not be altered in this way, since the client will disregard answers without corresponding requests. Only if the attacker's message arrives before the correct one, will it be accepted by the client.

Problem A11 If an attacker can partition the network and place a bridge between a server and a client, all unprotected data sent between the client and the server can be modified on the way. The attacker can for example encrypt all data going to the server and decrypt the data going back to the client. When this function for some reason stops, it leaves the database and all backups unreadable to the system owner. A more subtle attack would be to change the values of selected fields into erroneous data, but send the original values when certain users request those records, in order to prevent detection. The purpose of this attack could for example be to improve the grades or salaries of friends or to slander enemies.

A summary of the problems caused by misdirected trust is given in Table 1.

4.2.2 Discussion It is an attractive characteristic of a secure system if it could be implemented in such a way that not all units need to be trusted, and many specifications do indeed require a system layout in which some units are insecure. However, the multitude and severity of the problems listed above indicate that it is not evident how such a design should be carried through.

There are two possible approaches to circumvent this problem. One obvious way would be to waive the original requirement of untrusted units and secure them, physically for example. However, this may not be easy to accomplish, and in many cases the requirement of having untrusted units is based on a well-founded need. The other way requires that the security mechanisms that the trusted parts provide be sufficiently robust to preserve the security policies of the system, even if all the untrusted parts are controlled by an attacker. The techniques needed to accomplish this differ between different units.

Table 1. Problems caused by misdirected trust.

Problem		**Point of attack**	**Means of attack**
No.	**Description**		
A1	Trojan horses can monitor all user commands.	Client	TH
A2	Trojan horses can ask server for arbitrary information.	Client	TH
A3	SIU connected to the serial port can be used by external system through extra cable.	Client	HW
A4	Software or hardware can be added that (invisibly) bypasses encryption hardware or weakens encryption algorithms/keys.	Client + Server	TH (HW + SW)
A5	OS/2 swap file may contain information.	Client + Server	R disk
A6	Given physical access to the server, it is easy to modify or remove existing records.	Server	W DB
A7	DBMS log files could reveal information.	Server	R disk
A8	A Trojan horse could capture the keys used for initialization of the installed hardware and users' smartcards.	Server + Adm. host	TH
A9	A network eavesdropper can read all encrypted fields, since encryption and decryption are done in server, and with that possibly identify records.	Network	Passive
A10	Replays are possible with modified data by re-using headers.	Network	Active
A11	Network partitioning can invisibly modify data, for example by encryption or subtle alteration.	Network	Active

'+' denotes logical OR.
Means of attack: TH = Trojan horse, SW =software, HW = hardware, R disk = read the local hard disk of the PC, W DB = write to the database, Passive = a passive network attack (interception), Active = an active network attack (interruption, modification or fabrication)

Network security could probably be tightened up by a more extensive and careful use of the cryptographic functions, at the cost of lower performance when more data needs to be encrypted. A careful selection of the cryptographic protocols would also improve the situation, but that selection is a non-trivial task [24].

The way in which clients should be secured against manipulation depends on the situation at the customer site. If the PCs are used almost exclusively for this application, a solution in which the PCs are unusable without an authorized user's smartcard being inserted in the reader (for example by cryptographic protection of the disks) will be appropriate. On the other hand, if the PCs are used for many other applications, and the users want to install various software freely, it is difficult to secure clients against malicious manipulation. Another approach may be to boot the system securely and to prevent subsequent manipulation. In this case, the boot must be from a secure unit, such as a server [19]. Still another possibility that may be discussed is to boot the system with the help of the smartcard.

4.3 Cause B: Misuse of security mechanisms

The target system makes use of good, well-tested security mechanisms, in particular cryptographic functions provided by the trusted hardware components. However, even if the building blocks are of a high quality, they may be used in such a way that the security improvement is inadequate or even non-existent. In other words, there are normally important constraints, for example with respect to initialization and integration of the mechanisms, which must be observed in order to achieve the full security impact. The problems below are examples of failures to observe such constraints.

4.3.1 Descriptions of problems

Problem B1 It must be simple for users to suspend their work temporarily and to take short breaks without having to restart the application or go through a complex authentication procedure. All users must feel that it is worth the trouble to remove their smartcards even for very short breaks. This is not the case in the system analysed. The user must quit the entire application to be able to remove the card. Thus, it is unlikely that a user removes the card when leaving for a short while, and consequently an attacker has a good chance of finding an unattended client that is ready for use.

Problem B2 Encryption of data is used to conceal the contents of fields in the database that can be used to identify the individual about whom a record concerns, and to conceal the record identifiers sent in transactions from a client to the server. The encryption algorithm used is CDMF [16], which is basically DES [21] but with an effective key length of 40 bits.

However, a key length of 40 bits is clearly insufficient for a system introduced today. This is the obvious conclusion from two observations:

1) Since the product is designed with the purpose of being exempted from U.S. export limitations [16], it is designed to be weak (easy to break).

2) The 56-bit key length of DES has been criticized for more than fifteen years [12]. Meanwhile, the cost of computing power has rapidly decreased.

The key-space exhaustion can be divided into many small processing parts, each executing on an otherwise idle computer. It has been shown that the computing power needed to find a 40-bit key in a matter of weeks, when using this method, is well within the hands of a large number of people in the computer community [9].

Problem B3 Encryption is performed in a block mode where the same plaintext value yields the same ciphertext for all records, and where partial changes to the plaintext make partial changes to the ciphertext. This can be used to make conclusions about the plaintext simply by looking at the ciphertexts of different records and comparing these. If the attackers know the plaintext of the field of one record, they can also identify all other records having the same value. This is a kind of *known plaintext attack*.

Problem B4 Attackers who can insert data into the database (such as authorized users with access to some, but not all records, who try to extend their privileges) can try different values and compare the resulting ciphertexts with those of the desired records. Since encryption is done in blocks of 8 bytes, it should be possible to apply dictionary attacks against some records where parts of such a block is known. For example, if 14 bytes are known and 2 bytes are being sought, an attacker has only to try 65,000 combinations, which can be done in less than a second. In fact, it is easy to do an exhaustive search for anything less than 5 bytes (40 bits) and, in all other cases, it is easier to attack the encryption key (40 bits) instead. Even worse, if the data being sought is not completely randomized, the numbers of combinations that must be tried are far fewer. As opposed to Problem B3, this is a kind of *chosen plaintext attack*.

Problem B5 When a transaction message is sent from a client to the server holding the Secret DB, the encrypted transaction header begins with a timestamp, as discussed in Problem A10. The encryption mode used is cipher block chaining (CBC) mode [20] with an initialization vector (IV) of binary zeroes. The timestamp is of the form (year, month, day, hour, minute, second) and is stored as a string of 14 digit characters, for example "19950428142439" denoting the time 14:24:39 of April 28, 1995. However, this means that the first eight characters (64 bits) are the same for all messages sent in a day. Consequently, an eavesdropping attacker is always provided with a pair of corresponding 64-bit ciphertext and plaintext blocks (since the attacker knows what day the message was captured), which is all that is needed to start an exhaustive key search since the attacker already knows the algorithm and the IV. This information is also sufficient for an exhaustive table attack [28], although such an attack is probably less cost-effective than an exhaustive key search.

A summary of the problems caused by misuse of security mechanisms is given in Table 2.

Table 2. Problems caused by misuse of security mechanisms.

Problem		**Point of attack**	**Means of attack**
No.	**Description**		
B1	Difficulty in leaving system for short breaks allows possibility to find unattended clients.	Client	Misuse of app.
B2	DES/CDMF with a 40-bit key is too weak.	Server + Network	R DB + Passive
B3	Identical fields from different records are identical after encryption (known plaintext attack).	Server	R DB
B4	Dictionary attacks against parts of the ciphertext may be possible (chosen plaintext attack).	Client & Server	Misuse of app. & R DB
B5	An eavesdropping attacker knows the first plaintext block and IV of every captured ciphertext message.	Network	Passive
'+' denotes logical OR, '&' denotes logical AND. **Means of attack:** Misuse of app. = misuse of the legal application (as opposed to a Trojan horse), R DB = reading the database (from disk or backups), Passive = a passive network attack (interception)			

4.3.2 Discussion It is obvious from the examples above that the use of encryption, tamperproof hardware devices etc., does not *necessarily* increase the security of a system. This owes to the fact that the application and integration of these products are not without difficulty [23], and that certain requirements must be respected. Also, vendors of cryptographic products tend to overestimate their customers' level of cryptologic and security design expertise [2]. Still, these requirements are often quite fundamental: use keys of appropriate length, do not encrypt plaintext that is known to the attacker, etc.

The problem with export restrictions for cryptography shows that security is really a multifaceted area. There are alternatives to DES/CDMF for customers outside the United States, for example the International Data Encryption Algorithm (IDEA) [17], but those are currently not supported by the hardware used in the target system.

Secure handling of secret IVs in message and field encryption is necessary for security, but might decrease system performance and increase the complexity of the security mechanisms.

The problem with users leaving the workstation without their card once more supports the human and user-related aspect of security. It is probably possible to devise a procedure for short breaks, so that the user does not need to quit the application to be able to remove the card and can thus make a fast re-authentication when returning to the workstation. However, great care must be taken in designing procedures involving direct human interaction. There are many examples in which such procedures have turned out to be useless as a result of the reluctance of the users to respect the constraints of the procedure.

4.4 Cause C: Implementation deviations from specification

This class compiles all problems that are consequences of the fact that requirements in the original specification were never carried through. It also includes those cases in which there is no formal requirement, but in which the problem has been correctly addressed elsewhere in the design documentation.

4.4.1 Descriptions of Problems

Problem C1 An authorized user can create a merged copy of the entire Open and Secret databases by requesting record after record. In the system analysed, there is no limit on the number of records a single user can access in a limited period of time.

Problem C2 In the system analysed, a user who is authorized to read records in the Open DB or the Secret DB may also make arbitrary changes to them. This is most likely due to the test status of the system, but is still a problem requiring attention.

Problem C3 Due to the design of the pilot register application, when a record for a person is requested from the Secret DB, the person is *always* first looked up in the Open DB. There is in fact no way to reach the secret data without first querying the Open DB when using the pilot register application. This means that a read-transaction to the Secret DB (with encrypted record identifiers) is always preceded by a read-transaction to the Open DB (with plaintext record identifiers) requested by the same client (and user). An attacker listening to the network can therefore easily combine records from the Secret DB with records in the Open DB by looking at either the user identity or the client identity in the network messages.

A summary of the problems caused by implementation deviations from specification is given in Table 3.

4.4.2 Discussion The fact that issues that were correctly addressed, solved and specified during the system design phase are still not implemented is a sign of, possibly subtle, organisational problems in the development organisation. Problems of this kind are well-known to the software community and have been so for many years. It is interesting to note that, despite this knowledge, such problems persist even in large and experienced organisations. This strongly supports the standpoint that security is not only a result of technical solutions, but that "soft", human-related issues play an important role. The solution, as usual, must be sought in the areas of good software engineering practices and traditional team management.

4.5 Cause D: Oversights and bugs

This class includes pure mistakes and "bugs" in the design and implementation, as well as problems that depend on failure to notice the security impact of certain

Table 3. Problems caused by implementation deviations from specification.

Problem		Point of attack	Means of attack
No.	**Description**		
C1	Dishonest employees can create a merged copy of the databases.	Client	Misuse of app.
C2	Read access automatically implies write access.	Client	Misuse of app.
C3	When a record from the Secret DB is requested, the Open DB is always first consulted.	Network	Passive
Means of attack: Misuse of app. = misuse of the legal application (as opposed to a Trojan horse), Passive = a passive network attack (interception)			

design areas. It is related to the previous class, but the issues presented here were never discussed or mentioned during the design phase.

4.5.1 Descriptions of problems

Problem D1 Records are supposed to have a last date for removal (periodic purge). This field could be used to match records between the databases, with some certainty.

Problem D2 The designers have no control over the ordering of records in the database. In the system analysed, records are always added in the *same order* in both the Open DB and in the Secret DB. With access to copies (disk or backups) of the two databases, it is an easy task to identify all the secret records.

Problem D3 With access to backup tapes of different generations, it should be easy to match added (and deleted) records from the two databases. It may not be possible to match records with 100% certainty, but it may give a good indication that a record from the Open DB corresponds to a record in the Secret DB. (This method is probably interesting only to an attacker if Problem D2 is solved, because if records are stored in the same order in the databases, an attacker can match records from a single generation of backup tapes.)

A summary of the problems caused by oversights and bugs is given in Table 4.

4.5.2 Discussion All of the problems presented relate to the concept of separation of open and secret data. The reason for this is probably the inherent clash of interests between the users' demand for functionality and the designers' security requirements; users want to merge information from the Open and the Secret DB when working with the application, while the system tries to keep the information separated. It is worth noting that the developers do not seem to have considered results of research and practice in database security, see for example Castano *et al.* [8].

Table 4. Problems caused by oversights and bugs.

Problem		Point of attack	Means of attack
No.	**Description**		
D1	Periodic purge info could be used to match records in the databases.	Server	R DB
D2	Ordering of records in Open DB and Secret DB is the same.	Server	R DB
D3	Backup tapes could give information about which Open DB entries correspond to Secret DB entries.	Server	R DB
Means of attack: R DB = reading the database (from disk or backups)			

This class illustrates the general problem of system *validation*, as opposed to *verification*, that is the problem of designing a system consistent with the *real* requirements of the customer, which are not necessarily the same as the *specified* requirements. Therefore, no specific security-related remedies apply to this class, and the discussion can be referred to general reports on this topic, such as Sommerville [25].

5 Results obtained from the analysis

5.1 Comments to the classes

All causes of the vulnerabilities and problems found in the system analysed have been classified into one of four classes. The first class of these, "misdirected trust", represents a real problem area that is difficult to solve, in which a trade-off must be made between practical system design and security requirements. It is often very practical if not all units in the system have to be fully trusted, but the achievement would be to fulfil this requirement without compromising the security of the total system. Our analysis indicates that there are no evident ways to accomplish this. It could rather be questioned whether this is possible at all.

The class "misuse of security mechanisms" exhibits the problem of system design. Not only is it important to use highly secure building blocks; their interface requirements must also be fully understood and taken into consideration in order for them to be integrated correctly, and the interaction between them as well as with the users must be clarified. Failure to consider these aspects may result in a deficient system design, which is aggravated by the fact that the system owner is unaware of it.

Two of the classes, "implementation deviations from specification" and "oversights and bugs", are non-technical by nature but nevertheless very important. The reasons for these problems, and consequently also the remedies, are to be found in the area of software quality control, configuration management and "human engineering".

5.2 Some technical issues

- When ***clients are physically insecure***, *all* security sensitive data processing at the client end must take place *inside* the trusted hardware. Otherwise, the data is subject to disclosure or manipulation in some way. In the system analysed, this implies that for example personal numbers would have to be entered on the keypad of the SIU (see Figure 3) and encrypted *before* being sent to the PC. Still, sensitive data must eventually be output to the authorized user (displayed on the screen) and can be captured for example by a simple Trojan horse running in the background and taking repeated snapshots of the screen. In general, it is very difficult to use physically unprotected PCs running an insecure operating system when building a secure system, because any user, authorized or not, can do virtually anything to the client machine and its internal resources.

- It is difficult to use ordinary ***commercial software*** packages, since these are often not compatible with security evaluated products [23]. In our case, the security mechanisms of DB/2 or OS/2 LAN Server are designed with ambitions far lower than those of the system analysed, which makes them likely points of attack. We have observed the problems with ordering of records, protection of logs, client software authentication and others.

- When clients and servers communicate over an ***untrusted network***, designers must always assume that every bit sent over the network is read by an attacker, and that every bit received from the network may have been inserted or altered by an attacker. Solutions to this problem are far from easy to find but, when properly used, the cryptographic functions provided by trusted components can help to design protocols that are difficult to attack.

- The constraint of using only ***data encryption algorithms*** that are sufficiently weak to be allowed to be exported from the United States is a serious shortcoming, because their only effect will be to give the users and owners of the system a false sense of security. These algorithms form nothing but minor obstacles to a determined attacker. Rather, encryption algorithms and keys must be strong enough to survive attacks during and beyond the expected lifetime of the system. Two things must be kept in mind: First, the computer power available to an attacker is steadily increasing; and, second, when in operation, systems tend to be in use for a longer time than originally expected.

- The actual ***implementation does not always follow the original goals*** and intentions of the designers. This may be the result of lack of detail and explicitness in the design documents or other insufficient communication between designers and programmers etc. Whatever the reasons, this problem must be overcome in the development of secure systems. Otherwise, systems will continue to fail.

6 Conclusions

The security analysis performed revealed a significant number of vulnerabilities and potential vulnerabilities in the investigated system. It showed that the use of encryption, tamper-proof hardware devices, smartcards etc. does not necessarily increase the security of a system because vulnerabilities are introduced in the *application and operation* of the system [2–4].

Since the target system was a beta release and not a final product, a certain number of problems were to be expected. On the other hand, given the amount of effort and money that had already been invested in the design, and bearing in mind that the analysis was made by a small team with limited time and resources, it is surprising that a relatively large number of "trivial" problems still remained in the system, for example those related to unimplemented specification requirements. Therefore, security is as much an organisational and management issue as a technical one, and it must be addressed throughout the development process, from specification and implementation to operation and maintenance.

References

[1] D G Abraham, G M Dolan, G P Double, and J V Stevens. Transaction Security System. *IBM Systems Journal*, 30(2):206–229, 1991.

[2] Ross J Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 215–227, Fairfax, Virginia, November 3–5, 1993. ACM.

[3] Ross J Anderson. Making smartcard systems robust. In *Proceedings of Cardis 94*, Lille, France, October 24–26, 1994. Out of print.

[4] Ross J Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[5] C R Attanasio, P W Markstein, and R J Philips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[6] Jørgen Brandt, Ivan Bjerre Damgård, and Peter Landrock. Anonymous and verifiable registration in databases. In *Advances in Cryptology – Proceedings of EUROCRYPT '88*, volume 330 of *LNCS*, pages 167–176. Springer-Verlag, 1988.

[7] Sarah Brocklehurst, Bev Littlewood, Tomas Olovsson, and Erland Jonsson. On measurement of operational security. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS '94)*, pages 257–266, Gaithersburg, Maryland, June 27–July 1, 1994.

[8] Silvana Castano, Mariagrazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. Addison-Wesley and ACM Press, 1994.

[9] Simson L Garfinkel. Netscape's international browser security breached. *San Jose Mercury News*, August 17, 1995.

[10] Peter D Goldis. Questions and answers about tiger teams. *EDPACS The EDP Audit, Control, and Security Newsletter*, XVII(4):1–10, October 1989.

[11] Ulf Gustafson, Erland Jonsson, and Tomas Olovsson. Security evaluation of a PC network based on intrusion experiments. In *Proceedings of SECURICOM 96 – 14th Worldwide Congress on Computer and Communications Security Protection*, pages 187–202, Paris, France, June 5–6, 1996. MCI.

[12] Martin E Hellman. DES will be totally insecure within ten years. *IEEE Spectrum*, 16(7):32–39, July 1979.

[13] I S Herschberg. Make the tigers hunt for you. *Computers & Security*, 7(2):197–203, 1988.

[14] D B Johnson and G M Dolan. Transaction Security System extensions to the Common Cryptographic Architecture. *IBM Systems Journal*, 30(2):230–243, 1991.

[15] D B Johnson, G M Dolan, M J Kelly, A V Le, and S M Matyas. Common Cryptographic Architecture Cryptographic Application Programming Interface. *IBM Systems Journal*, 30(2):130–150, 1991.

[16] D B Johnson, S M Matyas, A V Le, and J D Wilkins. The Commercial Data Masking Facility (CDMF) data privacy algorithm. *IBM Journal of Research and Development*, 38(2):217–226, March 1994.

[17] X Lai and J Massey. A proposal for a new block encryption standard. In *Advances in Cryptology – Proceedings of EUROCRYPT '90*, volume 473 of *LNCS*, pages 389–404. Springer-Verlag, 1991.

[18] Richard R Linde. Operating system penetration. In *Proceedings of the National Computer Conference*, volume 44 of *AFIPS Conference Proceedings*, pages 361–368, Anaheim, California, May 19–22, 1975. AFIPS Press, Montvale, New Jersey.

[19] Mark Lomas and Bruce Christianson. To whom am I speaking? – Remote booting in a hostile world. *Computer*, 28(1):50–54, January 1995.

[20] National Bureau of Standards, U.S. Department of Commerce. *DES Modes of Operation*, December 1980. FIPS PUB 81.

[21] National Institute of Standards and Technology, U.S. Department of Commerce. *Data Encryption Standard (DES)*, December 1993. FIPS PUB 46-2.

[22] Tomas Olovsson, Erland Jonsson, Sarah Brocklehurst, and Bev Littlewood. Towards operational measures of computer security: Experimentation and modelling. In Brian Randell et al., editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, chapter VIII. Springer, Berlin, 1995.

[23] William R Price. Issues to consider when using evaluated products to implement secure mission systems. In *Proceedings of the 15th National Computer Security Conference*, pages 292–299, Baltimore, Maryland, October 13–16, 1992. National Institute of Standards and Technology/National Computer Security Center.

[24] Gustavus J Simmons. Cryptanalysis and protocol failures. *Communications of the ACM*, 37(11):56–65, November 1994.

[25] Ian Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992.

[26] The Swedish Patent and Registration Office. *Anordning och metod för lagring av datainformation*, 1994. Swedish Patent No. 501 128.

[27] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[28] Victor L Voydock and Stephen T Kent. Security mechanisms in high-level network protocols. *Computing Surveys*, 15(2):135–171, June 1983.

Paper B

Analysis of Selected Computer Security Intrusions:
In Search of the Vulnerability

Technical Report 275, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1996. Presented at NORDSEC – First Nordic Workshop on Secure Computer Systems, Göteborg, Sweden, Nov. 7–8, 1996.

This page is intentionally left blank.

Analysis of Selected Computer Security Intrusions: In Search of the Vulnerability

Ulf Lindqvist Ulf Gustafson[*] Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{ulfl, erland.jonsson}@ce.chalmers.se

Abstract

This paper presents an in-depth analysis of some selected computer security intrusions we have encountered during intrusion experiments and security analyses. The intrusions presented here illustrate the wide range of threats that owners and users of modern distributed computing systems must face. Several different dimensions of the intrusions are considered and discussed in detail, such as the flaw exploited in the intrusion, the cause of the presence of the flaw in the system, the method of attack, the initial result of the penetration, the possible implications and recommended remedies. It is argued that an intrusion is not feasible solely because of a single flaw, but is rather a function of a number of different flaws and characteristics of the system. This makes the job of detecting and preventing intrusions much more complicated.

1 Introduction

An intrusion into a computer system is possible owing to the existence of some kind of vulnerability, that is a characteristic that can be utilized for the purpose of penetrating the system. The immediate approach to prevent the intrusion would be to remove that vulnerability. However, an intrusion is not made possible by a single vulnerability in the system, but rather by a number of different vulnerabilities and characteristics that combine to form a penetration path. In many cases it is necessary to remove only one of the items in the path to make the intrusion impossible. In other cases, such an action will not suffice, since there can be ways to circumvent the obstacle created by this removal. Thus, we face a rather complex situation, which is reflected in the intrusion process as such.

The intention of this paper is to illustrate the complexity of the system characteristics that make intrusions possible, and thereby to shed light on the corresponding

[*]Author's present address: Ericsson Mobile Data Design AB, S:t Sigfridsgatan 89, SE-412 66 Göteborg, Sweden, Ulf.Gustafson@erv.ericsson.se

intrusion process, which in turn may help to design tools for intrusion detection. Other authors have approached the same problem from various angles. For example, Ilgun *et al.* [4] used state transition analysis to represent and model intrusions, Dacier *et al.* [2] used stochastic Petri nets for the same purpose and Reid [9] studied the anatomy and exploitation of vulnerabilities in contemporary systems.

Our approach is to base work mainly on the results of realistic intrusion experiments that we conducted [3, 8], as well as on results of a security analysis of a secure database system [6]. Although the main objective of the experiments was to find operational measurements of computer security, we inevitably gained knowledge of many vulnerabilities at the same time. Furthermore, from the reports of the alleged attackers, we were able to form a very good idea about *exactly how* the intrusions were performed[1], information that is not normally available in ordinary vulnerability databases (such as CERT[2] advisories) and which has helped us in the analysis.

In the following, Section 2 presents the scheme used for the description of the intrusions. Section 3 gives a straightforward presentation of five intrusions on three different systems. Section 4 presents a refined analysis of the underlying flaws, and Section 5 discusses the outcome of this analysis and the problem of referring an intrusion to a single cause. Section 6 concludes the paper.

2 Description scheme of the intrusions

This section describes a number of selected intrusions on three different systems [3, 6, 8]. The selection is made with respect to intrusion method, primary flaw and system design, so that a fairly wide area is covered. Still, each of the intrusions represent a severe attack against the central security functions of the system, resulting in the gaining of system administrator privileges or equivalent violation of the security policy.

In the analysis of such complex events as computer security intrusions, it is important to determine exactly what dimension (aspect, attribute) of the event the analysis concerns [5]. In this paper, we have chosen to discuss in detail several distinct dimensions of every intrusion presented in order to give a complete picture of the event.

The description is given in relatively great detail and includes the following dimensions: exploited (primary) flaw, attack method and potential impact of the intrusion. Possible remedies are also discussed. The description is structured under the headings presented below.

Summary Gives a short summary of the intrusion.

Description of the flaw Identifies the immediate or major reason why the intrusion was possible, and gives a relatively detailed technical description of the

[1]Because we do not want to provide attackers with new information, we have chosen to present only flaws to which system patches have been available for some time and methods of attack which are already well known.

[2]CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA

vulnerability.

Cause Identifies the cause of the flaw present in the system.

Method of attack Describes how the flaw can be successfully exploited in an attack.

Initial result of penetration Gives the “normal” or primarily expected internal result of an accomplished penetration. The result of a penetration can be difficult to determine [5], but identification of the initial result is important in certain types of intrusion detection systems [4].

Potential impact Tries to assess the worst possible damage that can result from the penetration, although a significant additional effort might be required to move on from the initial result.

Possible remedies Discusses possible ways to eliminate the flaw.

The distinction between ‘flaw’ and ‘cause’, for example, or ‘initial result’ and ‘potential impact’ is not always clear and unambiguous. We have had the ambition to make a practical and comprehensible demonstration of the intrusions, but the names and contents of the headings are subject to discussion.

3 Description of intrusions

The intrusions presented in this section represent different systems, different causes and different methods of attack, but a common denominator is the severity of the potential impact. In all the cases described, a carefully launched attack would effectively circumvent the security mechanisms of the system. In the Unix examples and the Novell example, an attacker can gain system administrator privileges, that is, complete control of the system. The database system in the last example was built primarily to strongly protect the privacy of the recorded individuals, expectations which are dashed by the described attack.

3.1 Unix: X terminal emulator logging vulnerability

Summary The X Window System terminal program *xterm*, running with the effective user id of *root*, had a flawed logging facility (CERT Advisory CA-93:17) that could be used to create any file or append to any existing file.

Description of the flaw The purpose of the logging facility is that all the output from a terminal session can be saved in a log file. The name of a log file can be specified as a command line argument to *xterm*. The program examines whether the invoker is permitted to access the specified log file by using the *access(2v)* system call, which uses the real user and group id in the verification. However, *xterm* runs with effective user id *root*, and the logical error is that the real user id (that of the user invoking the program) is used to check the existence and protection of the log file, while the effective user id (*root*) is used for the opening of the file.

Cause The reason why *xterm* runs with effective user id *root* in the default installation is that it needs to change the owner of the pseudo-terminal slave device. The code that handles the opening of the log file was not written with the caution required for such programs.

Method of attack In a directory in which the attacker has write permission, the attacker creates a symbolic link to either a non-existing file or an already existing file. First, the attacker changes the current directory to the directory above the one containing the symbolic link. Then *xterm* is started with the symbolic link specified as the log file. If the file pointed to by the symbolic link does not exist, it will be created and will contain the output from *xterm*. If the directory containing the symbolic link is writable but not readable by the attacker, and the link points to an existing file, the output of *xterm* will be appended to that file.

Initial result of penetration Any user can create any file or append to any existing file.

Potential impact Any user can gain *root* access, for example by appending to */etc/passwd* or appending to or creating */.rhosts*.

Possible remedies The distributed patches have disabled the logging facility. A better countermeasure would be to rewrite the program to only turn on super-user privileges for the operations where that is absolutely necessary, following "the principle of least privilege" [11].

Comments A variant of this attack is to trick *xterm* into changing the owner of any existing file, as described in [1].

3.2 Unix: Writable /etc/utmp vulnerability

Summary The system logging file */etc/utmp* was world-writable in some distributions of SunOS and other Unix variants (CERT Advisory CA-94:06). This configuration flaw made it possible for anyone with access to a user account to gain *root* privileges.

Description of the flaw The */etc/utmp* file records information about who is currently logged on to the system. The file consists of a sequence of entries as declared in the */usr/include/utmp.h* include file. Each entry contains the name of the special file associated with the user's terminal, the user's login name, the remote host from which a user has logged in (when applicable) and the time of the login.

Many programs trust the information in the */etc/utmp* file. Rather than fetching the real user id of the process (by for example the *getuid(2v)* system call), the programs that trust the */etc/utmp* file collect the user id of the invoker from that file, often by using the *getlogin(3v)* library function.

Cause Some Unix variants were distributed with a world-writable */etc/utmp* file. The reason for this was that unprivileged programs that connect to a virtual terminal should be able to put entries in the file, to make the user appear to be logged in.

Method of attack An attacker can easily write a program that erases his entry from the */etc/utmp* file. This makes the attacker "invisible" from listings of programs such as *talk, rusers, finger, who* and in some cases *syslogd*. Of course, it is also possible to fool the system by changing the appropriate field instead of deleting it.

Initial result of penetration Any user can make arbitrary changes to */etc/utmp*. Programs that trust this information can be misled about the identity of the user. Also, programs that use the */etc/utmp* file for presenting the users currently logged in may be tricked into presenting arbitrary output, whereby it is possible to hide or falsify accounting information.

Potential impact In SunOS 4.1.x (SunOS 4.1.3_U1 not included), any user can gain *root* access.

Possible remedies Patches should be installed in all SunOS 4.1.x distributions (except SunOS 4.1.3_U1) for programs according to the CERT advisory mentioned. It is also possible to make */etc/utmp* writable only to *root*, with the disadvantage that unprivileged programs that connect to a virtual terminal are no longer able to put an entry into */etc/utmp*. This means that programs such as *who* that use */etc/utmp* will not know that a user is connected to a virtual terminal when an unprivileged program made the connection.

3.3 Unix: Keyboard snooping

Summary By connecting to the X Window server of a target host, an attacker can monitor and record all the user's keystrokes. Especially, passwords to the target system and to other systems can be recorded.

Description of the flaw In the X Window System, any client program that can connect to the display server can also take over the mouse or keyboard and, for example, send keystrokes to other applications or invisibly monitor all keystrokes that are sent to the server (keyboard events are shareable). There are basically two authorization mechanisms in the X Window System: the original host-based *xhost* facility and the host/user-based *Xauthority* facility (also referred to as *magic cookie authorization*) introduced in X11R4 [12]. The *Xauthority* facility was designed to prevent this particular attack by allowing server access only to the user logged in at the console (who can in turn grant access to other users), but the problem is that any *xhost* settings override the *Xauthority* settings.

Cause As the experiment target system [8] was set up, any user logged in to a workstation (not necessarily sitting by the console) could connect to the X Window server of that workstation, because the *xhost* mechanism was inadvertently set to allow this, effectively overriding the also enabled *Xauthority* mechanism.

Method of attack The attackers in the experiment found a program called *xkey* on the Internet. The program invisibly monitors all keystrokes sent to the specified X Window server and prints the characters on standard output. The printout can for example be saved in a file for later examination.

Initial result of penetration The attackers can capture everything the user types at the keyboard.

Potential impact If the user types a password, for example when starting an *ftp* or *rlogin* session, that password is captured by the attackers, who can use it to gain access to the account in question. One group of attackers let the program run in the background on all the target workstations during the duration of the experiment. This gave the attackers 69 user passwords, of which 25 were passwords to other systems than the target system.

Possible remedies If the *Xauthority* mechanism had been properly set up and enabled, and the *xhost* mechanism had been disabled, this attack would have been much more difficult to accomplish, although not impossible. The *Xauthority* mechanism is susceptible to network eavesdropping, for example, but an attacker who can tap the network can also capture transmitted passwords without using the X Window System at all. Another remedy is to use the “Secure Keyboard” option in the terminal emulator *xterm* when typing sensitive data (usually reached by clicking the left mouse-button while pressing the control key). Knowledge about this option is not very widespread among users, but it would have effectively stopped this particular attack.

3.4 Novell: NCP packet forging

Summary A deficiency in the NetWare Core Protocol (NCP) makes it possible for any ordinary user to become a supervisor-equivalent user on the NetWare network. This applies to any unpatched NetWare 3.11 systems or a NetWare 3.12 or 4.x system that does not use packet signatures.

Description of the flaw A NetWare server and NetWare client communicate with each other by means of NetWare Core Protocol packets. The packets include information about where the packet is going and where it came from, as well as a user authentication code and data. In NetWare 3.11 and earlier, there is no packet authentication done to validate the NCP packet’s origin. In NetWare 3.12 and higher, NetWare incorporates a packet signature authentication procedure. However, packet signatures is an option that must be turned on, since it is turned off by default. It must also be enabled on both the server and the clients. If the packet signatures option is not used in NetWare, it is possible for arbitrary users that have an account on the system to extend their privileges by forging packets and send them to the file server. This flaw is utilized by a well-known program called *hack.exe* or *nethack.exe,* which is written in the Netherlands.

Cause If the system administrator on a NetWare installation chose not to use packet signatures (for performance reasons, for example), the authenticity of the network packets would not be checked. This deficiency makes it possible for any user to produce a supervisor request, for example, by sending carefully constructed packets to the server. The NCP packet signatures option can be added to 3.11 through a patch, but the option is included in version 3.12 and higher. However, this is still a vulnerability and a potential threat to Novell networks both because the patch might not be installed on all NetWare 3.11 networks and because some administrators of higher versions might choose not to use packet signatures.

Method of attack Unfortunately there is not much information available about Novell NCP because this protocol is proprietary. However, a skilled attacker with insight into NCP can still write a program that extracts one arbitrary NCP packet, originating from the supervisor or a supervisor-equivalent user, from the network and fill it in carefully [13]. The attacker then sends a specific number of forged packets to the NetWare server. From the file server point of view, one of the forged packets will appear authentic and the server will without questioning perform the actions as specified in the data part of the packet. The easiest way to perform this attack is to use the publicly available program *hack.exe*. Given that the supervisor is logged in, *hack.exe* changes the supervisor password when executed. Then every account on the file server is made supervisor-equivalent and, finally, the original supervisor is logged out.

Initial result of penetration Any legitimate user in a NetWare installation may change, add or remove arbitrary privileges for any user of the system.

Potential impact A program such as *hack.exe* may exploit the packet authentication deficiency by automatically giving supervisor equivalence to all users that have accounts on the file server.

Possible remedies The NCP packet signatures patch should be installed and activated in all NetWare 3.11 installations. To prevent packet forging, administrators should enable packet signature level 3, which forces packet signatures to be used. Clients that do not support packet signatures will not be able to access the file server, meaning that they will need to be upgraded. NetWare 3.12 and 4.x include the updated software, but the administrator must still set the correct packet signature level on both server and clients. When NCP packet signatures is used, any attempt to forge packets to the server will result in a message on the server console, in the error log, and be sent to the affected client. Packet signatures works by using an additional step during the encrypted password login call, to calculate a 64-bit session key. The session key is never transmitted over the network. It is used as the basis for a cryptographic signature (MD4 [10], according to Novell documentation) which is added on each NCP packet exchange. A packet with correct signature is taken as proof that it comes from the claimed client.

3.5 A database system: Ordering of records

Summary In a commercial database system, personal data was split into sensitive and public records, which were kept on two separate databases. An advanced cryptographic separation method was able to be circumvented for the simple reason that corresponding records were always added and stored in the same order in the two databases.

Description of the flaw The target system is a database application for storage and handling of personal registers, designed to protect the privacy of the recorded individuals. Every individual record is split into two parts: a record containing sensitive data and a public record. The two types of records are stored in two separate databases. The public records contain publicly available data, such as name, address etc., and a so called "personal number" used for record identification. For the sensitive records, encrypted personal numbers are used as record identifiers, and the idea is that only authorized users should be able to perform the encryption and thereby link the sensitive part of a record to the public (identifying) part, even with access to the data as stored on disk or tape.

However, when new individuals are added to the database, a public record is created in the "open" database, and a sensitive record is stored in the "secret" database. The database management system (DBMS) is a commercial off-the-shelf (COTS) product, and the application developers have no control of the ordering of records in the database. As it turns out, sensitive and public records are listed in the same order in the two databases.

Cause The application always adds a pair of records (one sensitive and one public) for each new recorded individual, and the DBMS always adds new records to the end of each database.

Method of attack With read access to the two databases, identification of sensitive records is a trivial task, since the first sensitive record corresponds to the first public record and so on.

Initial result of penetration An attacker can identify all sensitive records.

Potential impact If the sensitive fields are not encrypted (encryption of sensitive fields other than identifiers is an option in the system), the attacker can gain knowledge of sensitive information about all individuals recorded in the database.

Possible remedies If all fields of the sensitive records were encrypted, the described attack would be far more difficult (although the flaw might ease cryptanalysis). However, because encryption affects system performance, the system is supposed to guarantee privacy even if all fields are not encrypted. We can observe that, to achieve the desired separation and to make identification of sensitive records difficult, the ordering of records must be *at least* as difficult for an attacker to determine as the encryption of record identifiers. Otherwise, it is easier to attack the ordering than the encryption.

4 A refined analysis of the vulnerabilities

This section gives a detailed analysis of the actual reasons, for which the intrusions were possible. Even if, according to a first assessment, there is a major reason for the intrusion, a more detailed analysis reveals a far more complicated situation. In most cases, there are two or three types of reasons why the intrusion is possible: related to the *design* of a specific functional (software) module, *integration and setting up* of the system, and the *administration and use* of the system. The intrusions presented in Section 3 are now re-investigated in view of this decomposition.

Unix: X terminal emulator logging vulnerability

- By default, *xterm* in some Unix systems runs with effective user id *root*.
- The segment of the *xterm* code that opens the *xterm* file logging facility was not written with appropriate precautions.

To achieve the intended functionality, *xterm* had to run as *root*, regardless of the identity of the invoking user (*xterm* is 'setuid to *root*'). However, writing such programs, especially those that run as *root*, is a very delicate matter, and mistakes that potentially abuse overall system security are easily made. The major mistake leading to this vulnerability was that *xterm* executes some system calls as the *root* user instead of the user invoking the program.

Unix: Writable /etc/utmp vulnerability

- */etc/utmp* was world-writable, so that any program could put entries into the file.
- Some privileged programs trusted the information in */etc/utmp* and believed that it contained the correct names and terminals of the currently logged-in users.

The combination of the two items above constitutes the flaw. This should have been realized by the developers before the system was distributed, but this is far from the only implicit cross-dependency in a typical Unix system. Any system administrator could have eliminated the flaw by removing the world-write permission from */etc/utmp*, but that would probably have resulted in reduced functionality of some programs.

Unix: Keyboard snooping

- Any user at a host trusted by the target host's *xhost* facility is able to connect to the X Window server of the target host. By this, any such user may tap X Window events from the console user at the target host. In this particular case, keyboard events were monitored.

- A security feature, *Xauthority*, that addresses this kind of flaw has been introduced, but any *xhost* setting overrides any *Xauthority* settings.

The problem here is that the original X authentication procedure possesses severe limitations. Even though a better X authentication method is made available to most Unix systems today, *xhost* is still used on numerous systems, primarily because of its more understandable user interface. This fact totally destroys the intention of *Xauthority*.

Novell: NCP packet forging

- Originally, the NetWare server trusted any proper packet addressed to it on the network, and it was rather easy to find out how a proper packet was constructed.

- A packet authentication patch was made available, and the feature is included in more recent distributions of Novell NetWare, although the use of the facility is optional.

There may be many different reasons why the packet authentication facility was not implemented in the first versions of NCP. Perhaps the designers of NetWare were unaware of how easily packets could be forged in such an environment. Another, more likely, reason may have been that the designers believed that, because the NCP protocol was, and still is, proprietary, it would be too difficult for potential attackers to gain sufficient knowledge about the protocol to breach the system. If this was the case, then it was a devastating assumption, since it is obviously a risk that a program such as *hack.exe* is created and widely spread. This is an example of the well-known danger of "security through obscurity".

Even if the packet authentication facility is available, it is left to the system administrator to put it to work. It is more than possible that this facility is not activated on numerous NetWare systems today. For example, we performed an intrusion experiment on a Novell NetWare 3.12 system in an educational environment during late 1994 [3]. In this target system, the packet authentication option was not enabled, in spite of the fact that this flaw has been known since 1992.

A database system: Ordering of records

- The system consists of two databases, one sensitive and one public. The security of the system depends upon the demand that only authorized users should be able to determine what public record to which a given sensitive record belongs.

- The application always adds a pair of records (one sensitive and one public) for each new recorded individual.
- The DBMS always adds new records to the end of each database.

Because the designers of the database application had no control of the commercial database management system and neglected to check how the ordering of records was made in it, the security functions of the application were easily circumvented. This shows how a "simple" mistake can ruin an advanced and expensive security system.

5 Discussion

The interaction between different parts of a system, parts which are designed to meet different requirements, is clearly a security problem. This is obvious in the */etc/utmp* case, in the database case, and perhaps also in the keyboard snooping case. Those who designed the programs that trust the information in */etc/utmp* never thought that other programs would require the file to be world-writable; the designers of the database system never bothered to examine how the COTS DBMS stored the data; users and administrators were not aware that the sophisticated *Xauthority* mechanism was turned off when they enabled the more user-friendly *xhost* mechanism.

The *xterm* logging flaw is a case in which the designers had made an attempt to create a secure feature, but their attempt was not sufficient and did not follow the engineering principles that had been known for more than a decade in the security community [11]. The Novell designers fell into the trap that we call "misdirected (or misplaced) trust" [6], that is one cannot justifiably trust any message sent from an insecure PC client machine with respect to integrity or authenticity.

Many of the security problems shown in this paper are, as mentioned, of a rather complex nature. A disappointing fact, from a security point of view, is that even though the exploitation of the flaws initially demands deep system knowledge, an easy-to-use program or a step-by-step description that automatically performs the intrusion is often available. Among the intrusions described here, such tools are available for both the packet authentication problem in Novell, the *xterm* logging vulnerability and the keyboard snooping flaw in Unix. Such tools make it possible for even a user with minimal system knowledge to abuse system security.

6 Conclusions

This paper clearly shows that an intrusion is a function of not only one, but a number of vulnerabilities and characteristics of the system and the organization. This makes the problem quite complex, but complexity is unfortunately often a *de facto* property of security problems, no matter how much we try to simplify the real world in our models. Several system development methods are designed to deal with complexity [7], but complexity is a problem not only for developers and integrators but also for users and administrators, as we have seen in this paper.

It should also be noted that none of the problems that we have presented here are really technically difficult to solve. Solutions exist, but the problem is to spread this knowledge, use it, and use it correctly. Although security very often concerns deep technical details, if a development or customer organization does not make sure that these details are addressed correctly, the situation will not improve.

References

[1] Taimur Aslam, Ivan Krsul, and Eugene H Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Information Systems Security Conference*, pages 551–560, Baltimore, Maryland, October 22–25, 1996. National Institute of Standards and Technology/National Computer Security Center.

[2] Marc Dacier, Mohamed Kaâniche, and Yves Deswarte. A framework for security assessment of insecure systems. In *Predictably Dependable Computing Systems: First Year Report*, ESPRIT Basic Research Project 6362 – PDCS 2, pages 561–578. LAAS-CNRS, Toulouse, France, September 1993.

[3] Ulf Gustafson, Erland Jonsson, and Tomas Olovsson. Security evaluation of a PC network based on intrusion experiments. In *Proceedings of SECURICOM 96 – 14th Worldwide Congress on Computer and Communications Security Protection*, pages 187–202, Paris, France, June 5–6, 1996. MCI.

[4] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[5] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, California, May 4–7, 1997. IEEE Computer Society Press, Los Alamitos, California.

[6] Ulf Lindqvist, Tomas Olovsson, and Erland Jonsson. An analysis of a secure system based on trusted components. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 213–223, Gaithersburg, Maryland, June 17–21, 1996. IEEE, Piscataway, New Jersey.

[7] Peter G Neumann. *Computer-Related Risks*. ACM Press and Addison-Wesley, New York, 1995.

[8] Tomas Olovsson, Erland Jonsson, Sarah Brocklehurst, and Bev Littlewood. Towards operational measures of computer security: Experimentation and modelling. In Brian Randell et al., editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, chapter VIII. Springer, Berlin, 1995.

[9] Jim Reid. Open systems security: Traps and pitfalls. *Computers & Security*, 14(6):496–517, 1995. Also presented at Compsec International '95 in London.

[10] Ronald L Rivest. The MD4 message digest algorithm. In A J Menezes and S A Vanstone, editors, *Advances in Cryptology – Proceedings of CRYPTO '90*, volume 537 of *LNCS*, pages 303–311. Springer-Verlag, 1991.

[11] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[12] Dennis Sheldrick. Security and the X Window System. *UnixWorld*, pages 103–110, January 1992.

[13] David J Stang and Sylvia Moon. *Network Security Secrets*. Network World Secrets. IDG Books Worldwide, 1993.

This page is intentionally left blank.

Paper C

How to Systematically Classify Computer Security Intrusions

In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*,
pp. 154–163, Oakland, California, May 4–7, 1997.

This page is intentionally left blank.

How to Systematically Classify Computer Security Intrusions

Ulf Lindqvist Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{ulfl, erland.jonsson}@ce.chalmers.se

Abstract

This paper presents a classification of intrusions with respect to technique as well as to result. The taxonomy is intended to be a step on the road to an established taxonomy of intrusions for use in incident reporting, statistics, warning bulletins, intrusion detection systems etc. Unlike previous schemes, it takes the viewpoint of the system owner and should therefore be suitable to a wider community than that of system developers and vendors only. It is based on data from a realistic intrusion experiment, a fact that supports the practical applicability of the scheme. The paper also discusses general aspects of classification, and introduces a concept called dimension. After having made a broad survey of previous work in the field, we decided to base our classification of intrusion techniques on a scheme proposed by Neumann and Parker in 1989 and to further refine relevant parts of their scheme. Our classification of intrusion results is derived from the traditional three aspects of computer security: confidentiality, availability and integrity.

1 Introduction

The first step in wisdom is to know the things themselves; this notion consists in having a true idea of the objects; objects are distinguished and known by classifying them methodically and giving them appropriate names. Therefore, classification and name-giving will be the foundation of our science.

CAROLUS LINNÆUS, *Systema Naturæ,* 1735

The work presented in this paper emanates from intrusion experiments that we conducted [20]. The objective of the experiments was to find operational measures of computer security, that is measurements which reflect the dependence on and uncertainty of the operational environment in a probabilistic way, as opposed to static measures [19, 25] which reflect instead the quality of the system design. The need for a classification scheme arose when we were refining our modelling of the intrusion process [10].

Although several classification schemes focusing on different intrusion-related properties have been proposed, there is still no established taxonomy in general use. When trying to apply these schemes to our data, we found that they either focused on aspects other than those we were able to observe or that they were too superficial to be useful. We decided to develop a scheme that would fit our data, as well as be useful to others.

The motivations for a taxonomy and the objectives of the work are further explained in Section 2, while Section 3 is a note on the terminology used in this paper. The previous work in the field is presented in Section 4, the intrusion experiment is described in Section 5, and Section 6 describes our classification scheme. The advantages and limitations of the scheme are discussed in Section 7 and, finally, Section 8 concludes with a summary of the key points presented in this paper.

2 Rationale and objectives

Why would someone want to devise a taxonomy of intrusions? Is there a need for an established taxonomy? What tangible gain, other than the abstract aesthetic value of elegant expression and order, can justify the efforts required? Indeed, these are relevant questions, and we have found several answers.

- In general, categorizing a phenomenon makes systematic studies possible. In particular, a taxonomy of intrusions enables us to compile statistics on intrusions, observe patterns and draw other conclusions from collected intrusion data. We hope that this process will extend our knowledge of the phenomenon, and that it will be possible to strengthen systems against intrusions using this knowledge.
- An established taxonomy would be useful when reporting incidents to incident response teams, such as the CERT Coordination Center. It could also be used in the bulletins issued by incident response teams in order to warn system owners and administrators of new security flaws that can be exploited in intrusions. (The CERT Coordination Center has produced an "Incident Reporting Form" [6] which lists incident categories, however this does not constitute a proper taxonomy since it mixes intent, technique, vulnerability and result categories in an informal manner.)
- If the taxonomy included a grading of the *severity* or impact of the intrusion, system owners and administrators would be helped in prioritizing their efforts.

What is required by such a taxonomy? We have identified some desired (ideal) properties which are worth focusing upon in the formation of the taxonomy.

- The categories in a taxonomy should be mutually exclusive (every specimen should fit in *at most* one category) and collectively exhaustive (every specimen should fit in *at least* one category).

- Every category should be accompanied by clear and unambiguous classification criteria defining what specimens are to be put in that category.
- The taxonomy should be comprehensible and useful not only to experts in security but also to users and administrators with less knowledge and experience of security.
- The terminology of the taxonomy should comply with the established security terminology (something that is not always easy to define).

Landwehr *et al.* made an important general observation [13]:

> "A taxonomy is not simply a neutral structure for categorizing specimens. It implicitly embodies a theory of the universe from which those specimens are drawn. It defines what data are to be recorded and how like and unlike specimens are to be distinguished."

Amoroso pointed out the following properties to consider when inventing or selecting an attack taxonomy [1, chapter 3].

- *Completeness.* The taxonomy should encompass all possible attacks on the target system.
- *Appropriateness.* The selected taxonomy should appropriately characterize the attacks to the target system, that is any constraints on the taxonomy or on the system should be specified and considered before application.
- *Internal versus External Threats.* An attack taxonomy should differentiate attacks that require insider access to a system from those that can be initiated by external intruders who may not have gained access to the system.

3 A note on terminology

The terms intrusion, penetration, attack, breach and compromise are often used interchangeably, which can be a source of misunderstanding. Informally, we consider an *intrusion* (or penetration), which is a successful event from the attacker's point of view, to consist of: 1) an *attack* in which a *security flaw* (or vulnerability) is exploited, and 2) a *breach* (or compromise) which is the resulting violation of the explicit or implicit security policy of the system. An attack that does not lead to a breach is considered unsuccessful, although it may provide the attacker with some information, at least that the attempted attack does not work for some reason. However, the distinction between breach and intrusion is neither strict nor crucially important for the following discussion.

We have adopted a wide view of the *system* concept, according to which users can sometimes be considered part of the system or at least seen as part of the system context or environment. This is common in the field of safety engineering [22] and we also find it necessary to the security perspective. One reason for including users in the system concept is that sometimes an attack will be successful only when there are other users in the system who unknowingly interact with the attacker. For example, if an attacker plants a Trojan horse, it must be run by a credulous user in order to work. Another reason for adopting a holistic view of the system, rather than studying separate components when analysing intrusions, is that it is usually not important to the attacker *how* or *where* the intrusion is made, as long as the result is the desired one.

4 Previous work

Through the years, several classifications of intrusions have been presented, some concentrated on the intruders and their methods (that is the *threat* or *intrusion technique*) and others on the characteristics of the computer system that make the intrusion possible (that is the *vulnerability* or *security flaw*). The latter classifications do not usually take into account the exploitation of the categorized flaws, while the former often describe the exploited flaw in conjunction with the exploitation technique. For the sake of completeness, both types of classification are included in this survey of previous work.

4.1 Classifications of intrusion techniques and threats

An early work is that of Lackey, in which six categories of penetration techniques were presented [12]. The classification is “based on many examples of actual system penetration”, although no references are presented.

Neumann and Parker categorized computer misuse techniques into nine classes on the basis of data from about 3,000 computer abuse cases collected by the two authors over a period of 20 years [18]. The authors emphasize that their classes are not mutually exclusive in the sense that actual computer abuse cases often involve techniques from several classes. The classes are listed in Table 1. The order is roughly from the physical world (Class NP1) to the hardware (Class NP2) to the software (Class NP3 and higher), and from unauthorized use to misuse of authority.

We found the classification suggested by Neumann and Parker interesting since it appears to be well-founded and to cover most of the known techniques. It also has an elegant feature, namely the inherent grading of the classes, from external attacks to authorized users misusing their privileges. It is not perfect, however, and some of its shortcomings are discussed in Section 7 (Neumann presented a revised and extended version of the scheme [17, chapter 3], but we prefer the original version since the new scheme does not clearly separate technique from vulnerability or result).

Brinkley and Schell [5] categorized what they call information-oriented computer misuse (regarding the security aspects *confidentiality* and *integrity*, but not *availability*, which the authors call resource-oriented computer misuse) into six different classes, which are not mutually exclusive. No specific support for the classification scheme is presented, except for a small number of examples from other cited references.

In his Ph.D. thesis, Kumar made a classification of intrusions based on the “signatures” (patterns) they leave in the audit trail of the system [11]. The classification

Table 1. Computer misuse techniques [18].

Class		Description
NP1	External misuse	Generally nontechnological and unobserved, physically separate from computer and communication facilities, for example visual spying.
NP2	Hardware misuse	a) Passive, with no (immediate) side effects. b) Active, with side effects.
NP3	Masquerading	Impersonation; playback and spoofing attacks etc.
NP4	Setting up subsequent misuse	Planting and arming malicious software.
NP5	Bypassing intended controls	Circumvention of existing controls or improper acquisition of otherwise denied authority.
NP6	Active misuse of resources	Misuse of (apparently) conferred authority that alters the system or its data.
NP7	Passive misuse of resources	Misuse of (apparently) conferred reading authority.
NP8	Misuse resulting from inaction	Failure to avert a potential problem in a timely fashion, or an error of omission, for example.
NP9	Use as an indirect aid in committing other misuse	a) As a tool in planning computer misuse etc. b) As a tool in planning criminal and/or unethical activity.

is intended for use in intrusion detection systems based on pattern matching. Consequently, it does not consider intrusions that do not leave tracks in the audit trail, for example passive wiretapping.

4.2 Classifications of security flaws

In a general sense, a security flaw in a computer system is a kind of “bug”. Beizer presented a taxonomy of bugs that concentrates on where in the software development process the bug is introduced [3].

Landwehr *et al.* constructed a taxonomy of computer program security flaws, exemplified with 50 documented case studies of security flaws in different computing environments [13]. The flaws are categorized with respect to three characteristics or, as we suggest, in three *dimensions*. The dimensions are genesis (*how* did the flaw enter the system?), time of introduction (*when* did it enter the system?) and location (*where* in the system is it manifested?).

In a classic article, Saltzer and Schroeder present eight design principles for protection mechanisms, one of them being the well-known principle of least privilege [23]. Starting from these principles, and using UNIX as an example of an “unsecure operating system”, Hogan categorized security flaws in stand-alone systems and distributed environments [8]. This classification is chiefly concerned with why the flaws are present in the system.

Based on 49 cases in which UNIX security faults have led to intrusions, Aslam devised a taxonomy of security faults, as well as a design of a database for vulnerability data [2]. Aslam provides selection criteria that enable a distinct classification of the 49 cases. Only faults embodied in software are included.

5 The intrusion experiment

This section briefly outlines the arrangement of the experiment; for details see Olovsson *et al.* [20]. The target system consisted of a set of 24 SUN ELC diskless workstations connected to one file-server, all running SunOS 4.1.2. The attackers were 24 undergraduate students taking a course in applied computer security. They were all legal users of the system with normal user privileges and with physical access to all workstations except the file-server.

During this time, the system was in operational use for other laboratory courses taken by undergraduate students at the Department of Computer Engineering. The system itself was a ‘standard’ configuration, and thus not expected to differ significantly from other similar systems in use; it was supervised by an experienced system administrator. All standard monitoring and accounting features were enabled in the system to allow us to monitor the activities of each user account and to measure the resources each attacker spent during the breach process.

Through questionnaires, we know that the attackers did not consider themselves particularly knowledgeable about computer security issues compared with other students of the Computer Science and Engineering program, except for a certain degree of interest which made them choose to take the course in the first place. The attackers worked in groups of two. It was a deliberate choice to let ‘normal’ users attack the system, as opposed to professional attackers with experience from other systems. The attackers were informed that some specific activities were prohibited, namely doing physical damage to the system, attacking other systems, cooperating between groups or affecting the operation of other users on the system without first consulting the experiment coordinator. All attacking activities were to be carefully documented and reported to the coordinator.

A major motivation for the attackers was that the experiment was a compulsory part of the course they were taking. They were also given a general description of the overall objectives of the experiment so that they had a complete understanding of why certain rules must be obeyed, and why and in what way they should report their actions.

The attackers were told that a breach occurs *whenever the attackers succeed in doing something they are not normally allowed to do*, for example to use another user’s account. It is still somewhat difficult to determine objectively whether a given event is a valid breach or not but, after analysis of attacker reports and system logs, we have acknowledged some 60 separate, valid breaches in this experiment.

6 Taxonomy

6.1 Introduction

When examining specimens for classification, it should be noted that the specimens often have many different attributes, any of which could be chosen as the basis of the classification. We suggest the use of the term *dimension* for such an attribute. Accordingly, it is important to decide exactly what dimension of an intrusion the classification should be based on, because there are indeed several possibilities: the system component that was attacked; the intent of the attacker; the technique used in the attack; the reason why the exploited flaw is present in the system; the outcome of the intrusion etc. Some classification schemes make this point very clear (for example [13]), while others are less specific.

When we tried to categorize the flaws exploited in our recorded intrusions according to the scheme of Landwehr *et al.* [13], we found that the only feasible dimension, based on the information we had, was location. Since neither the details of the system development process nor the source code was available to us, only a minority of the flaws could be categorized with respect to genesis or time of introduction. Furthermore, for many of our recorded intrusions, it is not a trivial task to determine the actual flaw. Consider for example the scenario in which an attacker feeds the password file to a password-guessing program that tries words from various dictionaries. What is the vulnerability that makes this attack possible? Is it the fact that every user can read the encrypted passwords in the password file? Or is it the fact that some users tend to choose easy-to-guess passwords? Or is the encryption method not sufficiently sophisticated? Or is a single reusable password simply insufficient for the authentication of users?

We would like to be able to make a classification from the system owner's point of view. That is why we focus on the external observations of attacks and breaches which the system owner can make. An owner of a system is usually unable to categorize security flaws in detail. This is because most of the software and hardware is purchased from system vendors; source code and internal design is most often proprietary and not available from the vendor.

We believe that the dimensions of an intrusion that are most interesting to system owners are *intrusion techniques* and *intrusion results*. Details of the *intrusion technique* are needed to gain an understanding of intruders and the threat that system owners face. In addition, with this knowledge, it is often possible for the administrator to apply a quick fix to stop further intrusions of this kind while waiting for a patch from the vendor. This quick fix can be, for example, to clear the set-user-id bit of a flawed program or to remove a service completely (this usually has a negative impact on the service to legal users of the system). Information about the *intrusion result* is needed for the system owner to judge how critical the intrusion is according to the security policy of the system. For example, in some systems, disclosure of confidential information is considered much worse than denial of service while, in other systems, it is exactly the opposite. Another important field of application for data on intrusion results and techniques is the design of intrusion detection systems.

Our classification of intrusion techniques is presented in Table 2 and our classification of intrusion results in Table 3. For each category of the two dimensions, we give the number of intrusions from our experiment that fit in the category. The number is zero in some categories; nevertheless they are included as we believe that such intrusions are possible, although they did not occur in this particular experiment. The dimensions and their categories are explained and illustrated with examples below.

6.2 Intrusion techniques

As the scheme of Neumann and Parker [18] appeared to be the most useful of the previous classifications of intrusion techniques, our first step was to try to classify the intrusions made during the experiment in those classes. Since all attackers in our experiment were authorized users of the system, we expected that most of the intrusions would fit into the higher classes. The result was that all of the intrusions could be entered in class NP5, NP6 or NP7 (see Table 1). Our goal was a more fine-grained partitioning, however; thus our next step was to define subclasses below the three classes in the Neumann and Parker scheme.

6.2.1 Category NP5: Bypass of intended controls The category *bypass of intended controls* was divided into three subclasses: *password attacks, spoofing privileged programs,* and *utilizing weak authentication.*

Password attacks, as already pointed out by Neumann and Parker, is a broad subclass that includes all intrusions in which passwords are in some way involved. We decided to further divide this subclass into the third-level categories *capture* and *guessing,* since different countermeasures apply to the two techniques. *Spoofing privileged programs* is a technique in which programs executing with higher privileges are tricked to perform illicit operations on behalf of the attacker. *Utilizing weak authentication* is the technique of taking advantage of the fact that the system does not perform proper authentication of the originator of certain requests. Examples of this subclass include: obtaining client root privileges by manipulating the boot process, obtaining server root privileges by executing a set-user-id program generated by a client root, sending email with faked headers by manually interacting with the mailer daemon, and other situations in which the system trusts an identification without requiring any authentication token at all.

6.2.2 Category NP6: Active misuse of resources The category *active misuse of resources* was divided into the two subclasses *exploiting inadvertent write permission* and *resource exhaustion.*

Exploiting inadvertent write permission includes exploitation of the fact that many system objects are by default world writable. This means that any user on the system can modify these objects, although this is seldom the system (or object) owner's intention; it is the same for group writable objects. These objects are often found by using the techniques of category NP7. *Resource exhaustion* is a technique used to cause denial of service, for example by consuming all available disk space.

Table 2. Taxonomy of intrusions: Intrusion techniques.

Category			**Number of intrusions**
NP5 Bypassing intended controls	Password attacks	Capture	6
		Guessing	12
	Spoofing privileged programs		6
	Utilizing weak authentication		13
NP6 Active misuse of resources	Exploiting inadvertent write permission		12
	Resource exhaustion		0
NP7 Passive misuse of resources	Manual browsing		1
	Automated searching	Using a personal tool	0
		Using a publicly available tool	8

UNIX is very susceptible to this kind of attack, but it is often easy to track down the source of the problem [21], making the attack only temporarily useful. The participants in the intrusion experiment were explicitly told not to use an attack of this kind, for example the command "`while true fork()`", which would effectively stop other users from starting new processes. If they had more innovative ideas for denial of service attacks that could not be traced, such attacks could be tried after discussion with the experiment coordinator at times when no normal users were present.

6.2.3 Category NP7: Passive misuse of resources The category *passive misuse of resources* is the "read" counterpart of NP6. It is natural to divide the techniques into *manual browsing* and *automated searching*; the latter involves the use of a special tool program designed to find security problems in a system. Such a program can be either constructed by the attacker for the particular attack or a general tool fetched from a public archive. Several such tools are available, for example COPS [7], which was a popular instrument among the participants in our experiment. The formation of third-level categories for distinction between *publicly available tools* and *personal tools* is motivated by detection mechanisms. It is often easy to design an intrusion detection system to recognize the characteristics of a public tool, while this is more difficult for tools that are previously unknown (compare with the problem of virus detection).

6.3 Intrusion results

What are the consequences of an intrusion? This question is more difficult to answer than might appear at first glance. Usually, it is meaningful to consider only the immediate result that characterizes a breach, because the total outcome of an intrusion depends on how the attackers move on from the initial breach. For example, if the attackers gain root access on the file-server, they can do virtually anything to the system and the final consequences are impossible to assess completely. In

Table 3. Taxonomy of intrusions: Intrusion results.

Category			**Number of intrusions**
Exposure	Disclosure of confidential information	Only user information disclosed	0
		System (and user) information disclosed	10
	Service to unauthorized entities	Access as an ordinary user account	19
		Access as a special system account	0
		Access as client root	3
		Access as server root	5
Denial of service	Selective	Affects a single user at a time	2
		Affects a group of users	0
	Unselective	Affects all users of the system	2
	Transmitted	Affects users of other systems	0
Erroneous output	Selective	Affects a single user at a time	6
		Affects a group of users	0
	Unselective	Affects all users of the system	8
	Transmitted	Affects users of other systems	3

our intrusion experiment, the attackers were told to stop when they had obtained the desired higher privileges, as we did not want them to disturb the work of ordinary system users [20]. In terms of real-time intrusion detection, another reason for concentrating on the immediate result is that it is desirable to detect the intrusion and take preemptive action as early as possible, preferably before any damage is done [9].

However, it is not obvious what should be considered the immediate result. A typical example is password-guessing. The very first result of a successful password-guessing attack is that the attackers gain knowledge of the user's password. A password is not just any piece of information, however, because the immediate implication is access to the user's account on the system. We decided to adopt a practical point of view, whereby we consider the result of a password-guessing attack to be access to the account in question.

Another example is the planting of a Trojan horse. The initial event is a modification or creation of an object in the system but, if the Trojan horse is never activated by a credulous user or system process, there is no detrimental result from the system owner's point of view. Consequently, we consider the result of the *activation* of the Trojan horse to be the result of the intrusion (although it would be desirable to detect the presence of the Trojan horse before it is activated). This example also illustrates that there is no point in considering intent when categorizing results. The creation and insertion of the Trojan horse is most likely done with malicious intent, but the activation can be considered an accident. Although we are concerned primarily with intentional attacks, the same results could in fact be caused by accidents (see [17] for more examples).

We decided to base our classification of intrusion results on the three traditional aspects of computer security: confidentiality, availability and integrity. The aspect of confidentiality is extended as suggested by Meadows [14] to *exclusivity*, to denote not only protection against unauthorized access to confidential information, but also protection against unauthorized use of the system. A breach of exclusivity results in *exposure*, a breach of availability results in *denial of service* and a breach of integrity results in *erroneous output*. Those are the top-level categories of our classification of intrusion results.

6.3.1 Exposure The exposure category is naturally divided into the subclasses *disclosure of confidential information* and *service to unauthorized entities*.

Disclosure of confidential information is further divided into the third-level categories *only user information disclosed* and *system (and user) information disclosed*, since we believe that cases of the former class sometimes (but not always) can be considered less severe than those of the latter. Examples of *disclosure of confidential information* include the following.

Reading backup tapes The tape streamer used for backups of the file-server was world-readable. The attackers in our experiment discovered that tapes were automatically ejected immediately after the backup procedure had finished writing to the tape. However, old tapes were reused and could be read from the time the tape was inserted to the start of the backup procedure. The result was that an older copy of the entire contents of the server's disks could be read by anyone on the system. (Result: *system (and user) information disclosed*; Technique: *manual browsing*).

Spoofing ARP The program */etc/arp* runs with the effective group id of `kmem` and, when a file which is readable to this group, for example */dev/kmem* or */dev/eeprom*, is fed to the program, parts of the file will be displayed as syntax error messages. (Result: *system (and user) information disclosed*; Technique: *spoofing privileged programs*).

Service to unauthorized entities is divided into third-level categories reflecting the privileges associated with the service delivered. The category *access as an ordinary user account* concerns either a legal user of the system who gains access to another user's account, or an outsider who gains access to any user account on the system. *Access as a special system account* means an account with higher privileges than an ordinary user account, but not super-user (root) access. An example from UNIX is `bin` or any other account that owns system files. The reason why we make a distinction between *access as client root* and *access as server root* is that in most client-server environments, the super-user on a client host has no special privileges on the server host. This is because users often have complete physical access to the client workstations, and consequently can manipulate the hosts in many different ways; they can reboot the machines, connect or replace storage devices or network connection cables etc. In fact, workstations to which the users have complete physical access cannot be trusted at all, although this is ignored in many systems (with the exception of the root identity on the server as mentioned above). This was realized in MIT's Project Athena, where the root password for the public workstations was not even kept secret; Kerberos was developed instead and used for user authentication [24]. Examples of *service to unauthorized entities* include the following.

Automated password-guessing The use of an automated tool for password-guessing based on dictionaries of likely passwords, a widely discussed and utilized technique, was also successfully used in our experiment. Many user accounts with simple passwords were compromised, but the root password was never guessed. (Result: *access as an ordinary user account*; Technique: *password attacks—guessing*).

Manipulating the boot process Several attackers tried to reboot a client host in single-user mode. Since this was successfully utilized in an earlier experiment to gain client root access, the system administrator had enabled the PROM password feature of the workstations to prevent this type of attack. However, some attackers found a method by which they could still reboot the host in single-user mode to become client root without being prompted for a password. (Result: *access as client root*; Technique: *utilizing weak authentication*).

6.3.2 Denial of service The subclasses *selective* and *unselective* for denial of service were suggested by Needham [16]. The third-level categories should be self-explanatory. By *transmitted*, we mean that the intrusion affects the service delivered by other systems to their users, not the service delivered by our system to other systems. In the latter case, other systems can in fact be seen as users of our system. There were no intrusions that caused denial of service on other systems in the experiment, but such intrusions are indeed possible. For example, an attacker can make a host on the system use the same IP address as a host on another system, something which normally causes both hosts to lose contact with the network. The possible range of this particular attack depends on the network configuration [4]. We have not separated transmitted attacks as selective or unselective, because it is difficult to define what unselective would mean for a transmitted attack, especially for the denial of service category. We hope that it is not possible for a computer on the Internet to cause denial of service on *all* connected systems (although the result of the Internet Worm incident in 1988 was too close for comfort). An example of *denial of service* is given here.

Causing a crash by remote copy to audio device There was a bug that caused a machine to crash immediately if the remote copy command `rcp` was invoked with the target */dev/audio*. If executed on the server, the whole system would go down. This was clearly a system bug, but the audio device should not be readable or writable to any user except the user currently logged in at the console. (Result: *unselective*; Technique: *exploiting inadvertent write permission*).

6.3.3 Erroneous output In the formation of the erroneous output category, it soon became evident that the same subcategories could be used as in the denial of service category. “Output” is used in a wide sense, and denotes more than what is shown on the user’s terminal or sent on a network connection. Modifications of system objects, such as the contents of files on hard disks or data structures in main memory, are also considered as “output”, and when that output is the result of an intrusion, the intrusion belongs to this category. Examples of *erroneous output* include the following.

Spoofing Xterm The X Windows terminal program `xterm`, running with the effective user id of root, had a flawed logging facility (CERT Advisory CA-93:17) which could be used to create any file or append to any existing file. Although this could be used to gain access as server root, we categorized the result as erroneous output, which was the immediate result. Our decision is supported by the fact that it is not obvious how to move on from the first step, that is to gain root access. (Result: *unselective*; Technique: *spoofing privileged programs*).

Faking e-mail By manually communicating with the mailer daemon, attackers can send e-mail messages with faked headers, particularly false sender identity, to any other system on the Internet. (Result: *transmitted*; Technique: *utilizing weak authentication*).

7 Discussion

The classification of intrusion techniques proposed by Neumann and Parker [18] is of course not perfect, nor is our extension of their scheme. It can be discussed for example whether all kinds of attacks involving passwords in one way or another should actually belong in class NP5, as stated by Neumann and Parker, or whether some belong in class NP7. Another problem, which always accompanies attempts to classify human behaviour, is how to obtain an unambiguous classification. The classification of intrusion techniques indirectly involves the intentions of the system owner and of the attacker, which are not always clear and logical. Therefore, it is sometimes a question of interpretation as to whether a certain intrusion belongs in one class or the other, or in both. Our subclasses are designed to be mutually exclusive with respect to technique but, as noted by Neumann and Parker, an actual case of abuse is often complex and involves several techniques. As observed by Meadows [15], it depends on the level of abstraction whether an action that is part of an attack is considered atomic or complex.

The classification of intrusion results is perhaps easier in the sense that the classes are in all essential respects mutually exclusive. The problem here lies in determining what it is meaningful to consider as the outcome of the intrusion, as discussed in Section 6.3. Although it would probably be desirable to include a grading of the severity of the intrusions, this is often a subjective and system-dependent property; it is therefore left to system owners who can judge how severe a particular result category is in their system, according to their security policy.

A significant question is whether our scheme is applicable to other systems and circumstances besides those of the experiment from which it was derived. Our proposed answer is based on the properties specified by Amoroso [1], as cited in Section 2.

- As to the *result* dimension, we believe that, with our definition of exposure, the top and second levels of our taxonomy satisfy Amoroso's *completeness* and *appropriateness* properties for most systems. The third level is more specialized and may fit only similar systems. We do not find any reason to differentiate between internal and external attacks in the *result* dimension, since the results can be the same regardless of the origin of the attack. For example, an intrusion in which an outsider guesses a user password and logs in as that user is categorized as *exposure – service to unauthorized entities – access as an ordinary user account.*

- The *technique* dimension is less general, as it is an extension of a more general scheme. For the system in our experiment, it is complete and appropriate. Since many systems in industrial and academic environments are very similar to our experimental system, we believe that our scheme is likely to have a wide field of application. Our experiment concerns only internal attacks, however external attacks are intended to fit in the lower classes of the Neumann and Parker scheme.

Although the size of the experiment is too small to draw strong conclusions about distribution in general, it is still interesting to examine the number of intrusions in the classes of the two dimensions we have studied. Figure 1 shows this distribution and is also a clear illustration of why the term dimension is appropriate. The figure shows that some techniques have a one-to-one correspondence to the result, while other techniques can be used to reach many different kinds of results.

8 Conclusions

We have presented a classification scheme for computer security intrusions, in which the classification is made with respect to the intrusion technique and the intrusion result, with the needs of system owners and administrators in mind. By using data from a realistic intrusion experiment, we have shown that the scheme is likely to be generally applicable. We believe that the proposed scheme will, with further application, evaluation and refinement, be a good candidate for a generally accepted taxonomy of intrusions.

Acknowledgments

The authors are grateful to several people who have read earlier versions of this paper and made valuable comments and suggestions. We thank especially Tomas Olovsson who also coordinated the intrusion experiment, and Per Kaijser for his support and suggested improvements.

The introductory quotation from Systema Naturæ was translated from the Latin original into English by M. S. J. Engel-Ledeboer and H. Engel, Nieuwkoop, Holland, 1964.

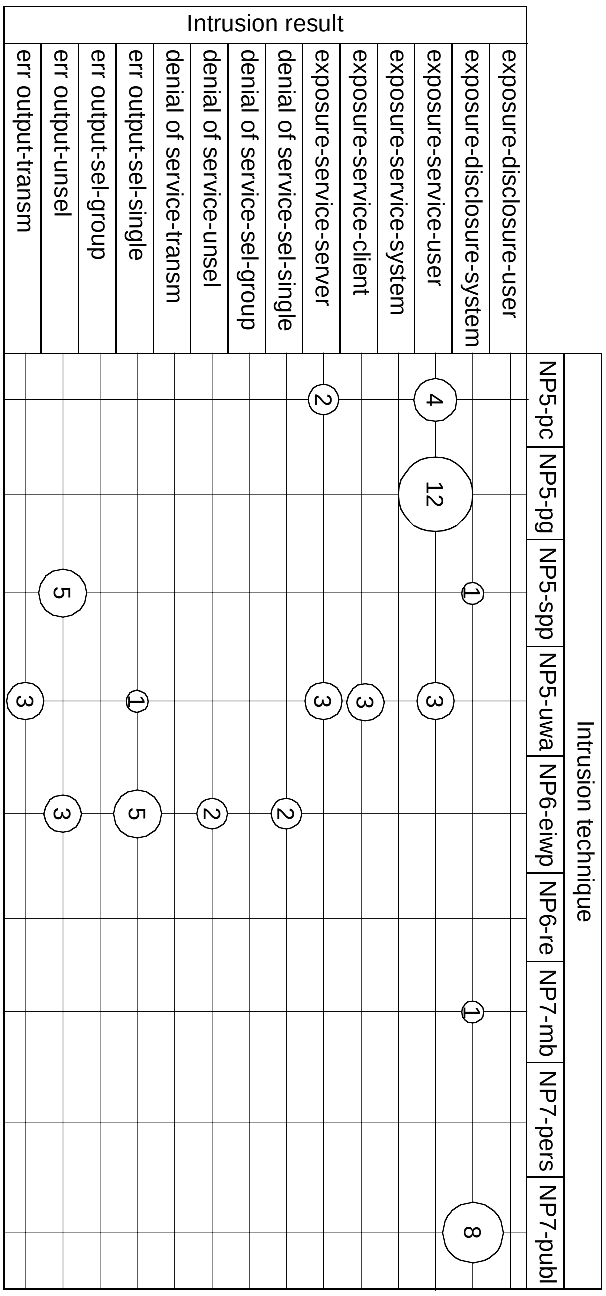


Figure 1. Distribution of intrusions in the two dimensions.

References

[1] Edward Amoroso. *Fundamentals of Computer Security Technology*. Prentice-Hall, 1994.

[2] Taimur Aslam. A taxonomy of security faults in the Unix operating system. Master's thesis, Purdue University, West Lafayette, Indiana, August 1995.

[3] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990.

[4] Scott O Bradner. A practical perspective on routers. In Daniel C Lynch and Marshall T Rose, editors, *Internet System Handbook*, chapter 7. Addison-Wesley, 1993.

[5] Donald L Brinkley and Roger R Schell. What is there to worry about? An introduction to the computer security problem. In Marshall D Abrams, Sushil Jajodia, and Harold J Podell, editors, *Information Security: An Integrated Collection of Essays*, pages 11–39. IEEE Computer Society Press, Los Alamitos, California, 1995.

[6] CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. *Incident Reporting Form*, February 28, 1996. Version 3.0.

[7] Daniel Farmer and Eugene H Spafford. The COPS security checker system. In *Proceedings of the Summer USENIX Conference*, pages 165–170, Anaheim, California, June 1990. USENIX Association.

[8] Carole B Hogan. Protection imperfect: The security of some computing environments. *Operating Systems Review*, 22(3):7–27, July 1988.

[9] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[10] Erland Jonsson and Tomas Olovsson. An empirical model of the security intrusion process. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 176–186, Gaithersburg, Maryland, June 17–21, 1996. IEEE, Piscataway, New Jersey.

[11] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1995.

[12] R D Lackey. Penetration of computer systems an overview. *Honeywell Computer Journal*, 8(2):81–85, 1974.

[13] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[14] Catherine A Meadows. An outline of a taxonomy of computer security research and development. In *Proceedings of the 1992–1993 ACM SIGSAC New Security Paradigms Workshop*, Little Compton, Rhode Island, September 22–24, 1992 and August 3–5, 1993. IEEE Computer Society Press, Los Alamitos, California.

[15] Catherine A Meadows. A representation of protocol attacks for risk assessment. In Rebecca N Wright and Peter G Neumann, editors, *Proceedings of DIMACS Workshop on Network Threats*, volume 38 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–10, Piscataway, New Jersey, December 2–4, 1996. American Mathematical Society.

[16] Roger M Needham. Denial of service: An example. *Communications of the ACM*, 37(11):42–46, November 1994.

[17] Peter G Neumann. *Computer-Related Risks*. ACM Press and Addison-Wesley, New York, 1995.

[18] Peter G Neumann and Donn B Parker. A summary of computer misuse techniques. In *Proceedings of the 12th National Computer Security Conference*, pages 396–407, Baltimore, Maryland, October 10–13, 1989. National Institute of Standards and Technology/National Computer Security Center.

[19] Office for Official Publications of the European Communities. *Information Technology Security Evaluation Criteria*, June 1991. Version 1.2.

[20] Tomas Olovsson, Erland Jonsson, Sarah Brocklehurst, and Bev Littlewood. Towards operational measures of computer security: Experimentation and modelling. In Brian Randell et al., editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, chapter VIII. Springer, Berlin, 1995.

[21] Dennis M Ritchie. On the security of UNIX, May 1975. Reprinted in *UNIX System Manager's Manual*, 4.3 Berkeley Software Distribution. University of California, Berkeley, USA, April 1986.

[22] John Rushby. Critical system properties: Survey and taxonomy. Technical Report CSL-93-01, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1993. Revised February 1994.

[23] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[24] Jennifer G Steiner, Clifford Neuman, and Jeffrey I Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, Dallas, Texas, February 9–12, 1988. USENIX Association.

[25] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

This page is intentionally left blank.

Paper D

A Map of Security Risks Associated with Using COTS

Computer, Vol. 31, No. 6, pp. 60–66, June 1998.

This page is intentionally left blank.

A Map of Security Risks Associated with Using COTS

Ulf Lindqvist Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{ulfl, erland.jonsson}@ce.chalmers.se

Abstract

The widespread use of commercial off-the-shelf (COTS) products in combination with increased internetworking calls for an analysis of the associated security risks. Combining Internet connectivity and COTS-based systems results in increased threats from both external and internal sources. Traditionally, security design has been a matter of risk avoidance. Now more and more members of the security community realize the impracticality and insufficiency of this doctrine. It turns out that strict development procedures can only reduce the number of flaws in a complex system, not eliminate every single one. Vulnerabilities may also be introduced by changes in the system environment or the way the system operates. Therefore, both developers and system owners must anticipate security problems and have a strategy for dealing with them. This is particularly important with COTS-based systems, because system owners have no control over the development of the components. The authors present a taxonomy of potential problem areas. It can be used to aid the analysis of security risks when using systems that to some extent contain COTS components.

1 Introduction

The traditional security design approach has been one of risk avoidance, not only in systems with high-security (military grade) requirements but also in medium-security systems, such as those typically found in financial institutions and corporate research departments. The design approach has been to construct mainly customer-specific solutions using security mechanisms such as physical “air gap” separation, information flow analysis, and strict or formal development and verification methods. However, there are several reasons why these techniques are applied less frequently today:

- Developing entirely customer-specific solutions is usually far more expensive and in all cases more time-consuming than purchasing COTS products.

- Some security mechanisms, particularly cryptography, have proven so difficult to implement correctly that developers should be provided with ready-made building blocks, relieving them of the risk of introducing subtle but serious flaws.

- Most organizations want connectivity and internetworking rather than physical separation. (Physical separation—also called “sneaker net”—requires manual intervention to transport data between a protected system and the outside world.)

More and more members of the security community realize the impracticality and insufficiency of risk avoidance as the sole doctrine. This was understood long ago in the reliability community, where fault tolerance was developed as a complement to fault prevention [2]. It turns out that strict development procedures can only *reduce* the number of flaws in a complex system, not eliminate every single one. Vulnerabilities may also be introduced by changes in the system environment or the way the system operates. Therefore, both developers and system owners must anticipate security problems and have a strategy for dealing with them [6]. This is particularly important with COTS-based systems, because system owners have no control over the development of the components.

2 Security-related COTS products

Any type of COTS component might have an impact on the overall system security, depending on how it is used in the system. Therefore every type of COTS product could be security-related. On the other hand, not all COTS products are designed without relevant security concerns. Some COTS products are indeed designed, implemented, and evaluated according to medium or even high levels of security functionality and assurance (although these products are in the minority). Examples of COTS products intended to improve security include cryptographic software (and hardware), network firewalls, and antivirus tools.

The open question, however, is what level of security one can attain by composing a system of different products. An ideal design goal would be to make the overall system security independent of how some untrusted components behave, but that is often difficult to accomplish in practice.

COTS operating systems deserve special attention, for several reasons:

- Operating systems are perhaps the most widespread COTS products.

- Only a handful of different basic types of COTS operating systems exist from which to choose, and they show wide variation in their security functionality and assurance.

- Many application programs rely on the operating system to enforce security mechanisms, such as user identification, authentication, and access control.

3 Taxonomy of security risks

Every situation in which the use of computers can affect something valuable (for example, human lives or health, privacy, economic assets, or national security) involves risks. Peter G. Neumann informally defines a risk as “a potential problem, with causes and effects,” although pointing out that there is no standard definition of the term [11, pp. 2, 348].

Here we are mainly concerned with security risks, which we define as

- the system, through human misuse, experiences loss of confidentiality, integrity, or availability for any of its resources; or
- the system, through misuse or by accident, experiences the introduction of a security vulnerability. (A security vulnerability is a flaw that could later be exploited to cause a loss of confidentiality, integrity, or availability.)

Our taxonomy is a map of potential problem areas. It can be used to aid the analysis of security risks when using systems that to some extent contain COTS components. It is based on the typical phases in the establishment of the system.

3.1 Component design

Some security risks originate from the design of the COTS components and are consequently beyond the control of the customers:

Inadvertently flawed component design The components may have various types of bugs, some of which may affect security.

Intentionally flawed component design The components may contain intentional security flaws, such as backdoors, viruses, or Trojan horses (for a more detailed explanation of this problem, see the sidebar “Defining the Confinement Problem.”).

Excessive component functionality A component may have many more features than the customer needs or even knows about, and so the customer might not realize the true security implications of including the component in the system.

Open or widely spread component design Although most academic security researchers (including ourselves) promote openness and public scrutiny for better security, a risk does exist if details of the component design are widely known outside the customer organization. Even worse, from a security point of view, is the common situation in which the design is known to a large development organization and its partners but not to the customers.

Insufficient or incorrect documentation The developer might not provide the customer with the documentation needed to correctly and securely integrate the component into the system.

3.2 Component procurement

There are also security risks associated with purchasing and delivering components:

Insufficient component validation A component purchase might not fully conform with the customer's *real* security requirements, which are not necessarily the same as the customer's *specified* requirements.

Delivery through insecure channel For example, in downloading a software component via the Internet, the product might be manipulated along the way by a third party (an intermediary attack) or the customer might be tricked into downloading a manipulated product from a site controlled by the attacker, instead of the real product from the vendor's site.

3.3 Component integration

Integrating components, which is a step in the design of the composed system, has the following risks:

Mismatch between product security levels A common problem when integrating different products is that the security level must be set to the lowest common denominator to make the products work together. For example, in the Microsoft Windows NT File System, user access to local system files and folders can be restricted to read-only permission to prevent accidental or intentional modification. However, for Microsoft Office 97 to work properly, the user must be given write permission for a number of system folders and files [10].

Insufficient understanding of integration requirements The integrators might not fully understand all of the preconditions for secure integration of the products, for example, that some components must be physically protected.

3.4 Internet connection of system

When the system is connected to the Internet, a number of additional risks must be considered:

Increased external exposure By connecting the system to the Internet, exposure expands to a large number of potential external attackers who otherwise would not have any data communication path to the system.

Intrusion information and tools easily available An insider who decides to attack the system can get a great deal of applicable information from the Internet.

Executable content Many World Wide Web pages have executable content (for example, Java applets) that automatically downloads and executes on a user's computer when viewing the page in a Web browser. Credulous users might well run programs that attack their system.

Outward channel for stolen information The Internet connection constitutes a channel that can covertly and conveniently export information stolen from the system, for example, by internal attackers or by programs planted by external attackers.

3.5 System use

Some risks are related to how the users operate the system:

Unintended use The system can be used in an unintended way, for example, to store and process data that are more sensitive than the system was designed to handle or to attack other systems.

Insufficient understanding of function Users might not be able to judge their adherence to the security policy if they do not fully understand a function. For example, they might not know whether or not a particular program transmits passwords in the clear over the network.

3.6 System maintenance

Finally, there are risks involved in the maintenance of the system:

Insecure updating In the same way as the initial software delivery is risky if performed via an insecure channel, software updates can be modified in transit or system owners can be fooled into installing fraudulent updates.

Unexpected side effects Any changes made to components in the system can have unexpected side effects and might introduce new security vulnerabilities.

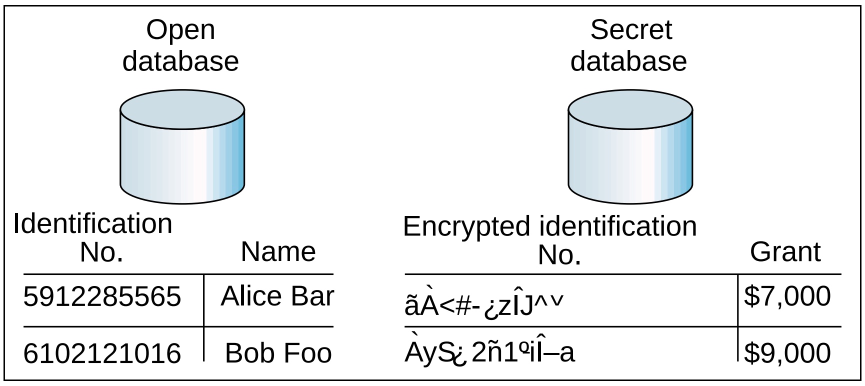
Maintenance backdoors The history of computer insecurity contains many cases in which developers left open backdoors for convenient testing and maintenance of their products. However, such backdoors can be misused by anyone who knows or finds out about them.

4 Analyzing risks to privacy in a database system

We were invited to investigate the security of a privacy-oriented database system under development. The system, which was based mainly on COTS products, was designed to strongly protect the privacy of the individuals recorded in the database. Our study revealed a large number of security problems that we reported to the developers and later further analyzed in terms of underlying causes and possible remedies [8].

The system was intended for personal registers in government or municipal services, offices in health care and in social care, and other public services. The major design goal was to make it virtually impossible to link a sensitive record in the database to an individual without proper authorization. To accomplish this separation of data, the designers used a combination of cryptographic devices and record pseudonymity. Their idea was to split identifying data and descriptive data between two separate databases, using an individual identification number, similar to a Social Security number, as the link between them. Figure 1 illustrates how data could be split between the two databases.

- The *open database* would contain publicly available data such as name and address, with plaintext individual identification numbers as record identification fields.
- The *secret database* would contain sensitive data, with encrypted individual numbers as record identifiers. That is, the records in the secret database are pseudonymous rather than anonymous. The designers did not require encryption of any other fields in the secret database unless that information could be used to link confidential information to specific individuals.



Identification No.	Name
5912285565	Alice Bar
6102121016	Bob Foo

Encrypted identification No.	Grant
ãÀ<#-¿zÎJ^v	$7,000
ÀyS¿2ñ1ºiÎ–a	$9,000

Figure 1. The two databases with simple examples of records.

The design goal was to make the database system resistant to many different kinds of potential attackers, ranging from dishonest or disgruntled current or former employees to other organizations and even foreign governments. Furthermore, even with physical access to the client hosts, to a copy of the databases from the server host, or to both, the attackers should not be able to violate the security policy of the system. The developers summarized the comprehensive security policy of the system in two statements:

i) *Confidentiality.* Only authorized users should be able to link records in the secret database to a single individual.

ii) *Integrity.* Only authorized users should be able to modify records in a meaningful way.

Most of the system components were COTS products from IBM: The computers were PS/2 PCs running OS/2 2.11, the Transaction Security System (TSS) [1] was used for encryption and authentication, the database-management system (DBMS) was DB2/2, and the OS/2 LAN Server was used for network communication between the clients and the server.

Several of the problems exposed in our study could be traced to the fact that all of the COTS components (except the TSS) were developed with lower security requirements than the composed database system. The developers had failed to make the security of the system independent of the (in)security of those components.

4.1 Risks revealed

A selected subset of the problems we found shows typical COTS-related security risks.

Trojan horse in client. Users had to present a smart card and a secret PIN to start the application and the cryptographic operations that were needed to identify records in the secret database. Owing to the lack of security protection mechanisms in the operating system, nothing could prevent an attacker with physical access to a client host from installing a Trojan horse that could, for example, record all transactions the user performs.

Information leaking to swap file. The virtual memory swap file of OS/2 might contain sensitive information. The application has no control over what information is transferred to this file. It would be possible for an attacker to collect information by searching this file after the authorized user has finished working.

DBMS log files. The log files of the DBMS contained, among other things, information about the origin of records, which could be used to correlate records in the two databases.

DBMS ordering of records. The designers had no control over the ordering of records in the database. In the system analyzed, records were always added in the same order in both the open and secret databases. With access to copies (disk or backups) of the two databases, it would be an easy task to identify all the secret records.

The last item is perhaps the most evident example of how a “simple” but serious problem can be overlooked when developers rely on general-purpose products to solve problems for which those products were not designed.

All of these problems can be categorized as insufficient understanding of integration requirements. Or, if we choose to consider the components as unsuitable for this type of application, as insufficient component validation.

5 Risks experienced in intrusion experiments

There is currently no established method to quantitatively measure the security of a system in comparison with other systems. However, there are guidelines for building systems with a certain level of security functionality and assurance, and developers can submit their system to a third-party evaluator who will try to determine whether the system was built according to the guidelines. In the reliability field, guidelines exist for building high-reliability systems, as well as methods to test the system and measure its reliability in an artificial operational environment, typically through various fault-injection methods [4].

With that analogy in mind, and with the objective of finding operational security measurements, we conducted intrusion experiments in which students were encouraged to attack a certain system for a limited period of time, under careful supervision and with the requirement that all their activities be reported and documented.

Thus far, we have performed one pilot experiment and three full-scale experiments on a Unix system (SunOS 4.*x*) and one full-scale experiment on a Novell NetWare system. We have analyzed the first Unix experiment and presented a model of the intrusion process [5], as well as a taxonomy of intrusion techniques and results. Analysis of the other experiments is in progress.

We chose to use ordinary students as attackers instead of experienced crackers and to provide them with standard user accounts. In this way, we would model the insider threat; that is, when legitimate users of a system for some reason decide to extend or misuse their privileges. We also ensured that each test environment represented a standard installation of a common COTS-based computing system.

The results of the stated experiments should be interesting to all readers who use comparable systems. Unfortunately, those results are not comforting:

- Almost all attackers performed successful intrusions.
- Several of the intrusions were indeed severe, giving the attacker administrator privileges.
- The Internet provides a vast amount of information on how to successfully attack common systems.
- Known vulnerabilities are often technically difficult to exploit. Still, many of our attackers broke into the system through such holes (often without really understanding why it was possible) by using so-called *exploit scripts* published on the Internet.

The last item describes a serious threat, of which today's system owners must be aware. A relatively small community of technically skilled crackers prepares programs that automatically exploit some vulnerability in a common type of system and makes these programs available on the Internet. Consequently, the group of potential attackers who can perform technically advanced intrusions now includes all who can find, download, and execute these programs—clearly an immensely large number of people.

The exploit scripts (which can be shell scripts, source code, or precompiled binaries) do not always work as distributed, probably to prevent people without any programming skills from using them. However, the errors are sometimes easy to fix, and many exploit scripts are ready to use. Typically, the exploit scripts take advantage of flaws in privileged programs (such as *setuid* programs in Unix) or processes (typically network server processes) by, for example, acting in one of the following ways:

- The exploit script calls the victim program with input data that were unexpected by the author of the victim program. The input data must in some cases be carefully crafted by the author of the exploit script, whereas in other cases the author simply provides an excessive amount of random data.
- The exploit script makes unexpected changes in the execution environment, for example, by moving or renaming files accessed by a privileged process.
- The exploit script retrieves the secret upon which security is based (typically a password or a cryptographic key) through, for example, shrewd guessing or an exhaustive search.

Our experiments resulted in a wide variety of intrusions, each of which would normally be possible owing to a combination of several risks rather than a single one. (Incidentally, this seems to be a general fact.) For example, one well-known intrusion uses the *setuid* mechanism of the Sendmail program. This may be related to the inadvertently flawed component design and excessive component functionality as well as insufficient component validation in the taxonomy. On the other hand, the fact that this intrusion method was known to the attackers may make it referable to the class intrusion information and tools easily available.

6 A risk management approach

The problem of protecting COTS-based systems connected to the Internet is difficult, because this combination increases outsider as well as insider threats. Simply by connecting to the outside world, a system becomes vastly more exposed to external attackers. Furthermore, as observed in our experiments, the existence and availability of exploit scripts and information about flaws also increase the threat from insiders.

Solutions must be sought in the risk management field, where the cost of protection is traditionally weighed against the potential loss caused by a violation of security. A modern risk management philosophy must also include the following:

A well-defined and relevant security policy The security policy primarily defines what is and is not allowed in terms of system security, although it should also include dictates regarding enforcement, responsibilities, and reporting. In fact, an intrusion is defined as a violation of the security policy (regardless of whether the violation comes from the inside or outside). Without a security policy, you cannot determine if an event is an intrusion, strictly speaking. Hence, the definition of a relevant security policy is a prerequisite for security risk management.

Holistic perspective System security must be viewed as a holistic property; it is not sufficient to just look at a small number of stronger parts (compare with the database system example). Thus, system security must be considered in a space as well as a time dimension. By time, we mean the system life cycle: Security risks should be estimated in the development, procurement, integration, operation, and maintenance of a system. By space, we mean the structure of the system and the environment in which it is embedded, an environment that includes humans, buildings, and organizations.

Confinement of untrusted components Processes that are untrusted should be limited in what they can do, ideally to the extent that they can do nothing else than exactly what they are supposed to do (see the sidebar “Defining the Confinement Problem”). For example, a process that normally does not perform any network communication should not be allowed to connect to the network. Confining untrusted COTS components is highly desirable, but is difficult to do in practice. It might be difficult to correctly determine the minimum set of resources that black-box components require to function in all cases. If you fail to do this, the component may not be able to operate in an unforeseen situation. Or you may specify a confinement that is too lax to provide an appropriate level of security. And if you use COTS products to perform the confinement (for example, using a standard Web browser to confine Java applets), the question becomes how far the guardians themselves can be trusted.

Partitioning The growing complexity of systems and networks is the source of numerous security problems. By partitioning a system into relatively small parts, separated by “watertight bulkheads” in the form of trusted monitoring components, the gain is twofold: Each part becomes less complex and more manageable in terms of security, and the effects of an intrusion are likely to be limited to a single section. The partitioning must be performed with caution, however, to avoid the creation of undesired “single points of failure,” for example, a single vulnerable link for the vital communication between two sections.

Contingency anticipation All systems have security vulnerabilities, and many sites will experience security violations. An organization that has accepted this and made appropriate planning will more likely succeed in limiting loss after having been the victim of an intrusion. This contingency planning, preferably performed with the support of software tools, should include methods for intrusion detection, evidence collection, and recovery.

Flaw remediation and active evolution A system owner should strive to remove all known vulnerabilities in a system as soon as they are discovered. This is not always easy, because all implications of the remedial actions must be carefully considered. Furthermore, component developers might be unwilling to produce patches. However, our experiments show that removal of all publicly known vulnerabilities would make the attackers’ task significantly more difficult. Continuous replacement of components when new security technologies emerge is also important.

Support The continuous risk management process must be supported by all levels of an organization, from top management down to ordinary users. Organizations using COTS products also need active security support from both developers and third-party vendors.

Awareness The tradition of covering up security incidents aids only the attackers and must be broken if we will ever have a chance to learn from earlier mistakes. Through education and openness concerning security, people can become more motivated and many risks can be avoided.

7 Conclusions

The use of COTS systems presents two faces, from a security point of view. On the one hand, security vulnerabilities in those systems will be continuously discovered, owing to the fact that crackers will find it more rewarding to look for flaws and write exploit scripts that can be used to attack many systems. On the other hand, a large customer base should mean that vendors can afford to make an extensive effort to fix security problems. Furthermore, such systems will be closely watched by the security community, and alerts of security problems will be readily announced. However, alerts are of little use if they are not read, understood, spread throughout organizations, and followed by appropriate measures taken by vendors, managers, administrators, and users. Thus, we believe that awareness and openness in security issues are the only means to gain manageable security in COTS-based systems.

References

[1] D G Abraham, G M Dolan, G P Double, and J V Stevens. Transaction Security System. *IBM Systems Journal*, 30(2):206–229, 1991.

[2] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the 1967 Fall Joint Computer Conference*, volume 31 of *AFIPS Conference Proceedings*, pages 733–743, Anaheim, California, November 14–16, 1967. Thompson Books, Washington, D.C.

[3] William E Boebert and Richard Y Kain. A further note on the confinement problem. In *Proceedings of the 1996 30th IEEE Annual International Carnahan Conference on Security Technology*, pages 198–203, Lexington, Kentucky, October 2–4, 1996. IEEE, Piscataway, New Jersey.

[4] Ravishankar K Iyer. Experimental evaluation. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (Special Issue)*, pages

115–132, Pasadena, California, June 27–30, 1995. IEEE Computer Society Press, Los Alamitos, California.

[5] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[6] Jay J Kahn and Marshall D Abrams. Contingency planning: What to do when bad things happen to good systems. In *Proceedings of the 18th National Information Systems Security Conference*, pages 470–479, Baltimore, Maryland, October 10–13, 1995. National Institute of Standards and Technology/National Computer Security Center.

[7] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[8] Ulf Lindqvist, Tomas Olovsson, and Erland Jonsson. An analysis of a secure system based on trusted components. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 213–223, Gaithersburg, Maryland, June 17–21, 1996. IEEE, Piscataway, New Jersey.

[9] Gary McGraw and Edward W Felten. *Java Security: Hostile Applets, Holes & Antidotes*. John Wiley & Sons, New York, 1996.

[10] Microsoft Corporation, Redmond, Washington. *OFF97: Security Requirements When Using NTFS Partitions*, August 12, 1997. Article Q169387 in Microsoft Knowledge Base.

[11] Peter G Neumann. *Computer-Related Risks*. ACM Press and Addison-Wesley, New York, 1995.

[12] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[13] Ravi S Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, November 1993.

[14] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

▸ Sidebar: Defining the Confinement Problem

The *confinement problem* is a classic computer security problem. Basically, it is a question of how to limit the actions of an executing program that normally has all the privileges of its invoking user and, therefore, can do anything the invoker can do.

One example of such a program is a Trojan horse. A Trojan-horse program appears to be benign and to behave as expected by the invoker but, in addition to or instead of the expected actions, does something malicious.

A classic and most elegant example of a Trojan horse was presented by Ken Thompson [14]. Thompson describes a portion of code hidden in the C compiler on a Unix system. When Thompson's modified compiler compiles the login program, it inserts a backdoor into the login binary that, for example, grants access to any account upon entering a "master key" password. To ensure that the Trojan horse does not disappear when the C compiler is replaced by a new version, Thompson's compiler detects when it is used to compile a new C compiler and inserts the Trojan code into the object code of the new compiler.

Unfortunately, Trojan horses present not only a theoretical problem for researchers but a serious threat to ordinary users of computer systems. In the world of computing today, factors such as increased software complexity, dynamic code linking, user abstraction from underlying functions, Internet connectivity, and frequent downloading of executable code from various sources all facilitate the insertion of Trojan horses and make them difficult to discover.

A general problem

However, the confinement problem does not apply only to Trojan horses. Untrusted software that is not malicious might still have side effects that are unexpected by the invoker and may cause security problems. This should be of particular concern to developers of systems that include COTS components.

Traditionally, computer security has focused on confidentiality, that is, making sure that only authorized subjects (people, processes) have read access to certain information. Therefore, an early definition of the confinement problem concerns information leakage: "...the problem of confining a program during its execution so that it cannot transmit information to any other program except its caller." [7] Great effort has been expended on the analysis and elimination of covert channels (unauthorized communication paths through which a process could transmit confidential information).

The basic problem can be stated more generally: "What is the appropriate way to confine an untrusted program so that it can do everything it needs to do to meet the user's expectations, but nothing else?" The imprecise definition does not give much hope for a final solution, and the real difficulty lies in correctly specifying the permitted behavior of the program.

Suggested solutions

The general problem could be addressed, however, if it were possible always to follow the principle of least privilege, which states that every subject should operate using the least set of privileges necessary to complete the job [12]. Several access control models designed with the purpose of enforcing that principle have been proposed [13]. With military applications in mind, designers have developed operating systems implementing *mandatory access control*, which bases confinement on data classification labels and personnel clearance.

A more recent approach is *domain-and-type enforcement* (DTE), in which an attribute called a domain is associated with each subject and another attribute called a type is associated with each object. A central matrix specifies whether a particular mode of access to objects of a type is granted or denied to subjects in a domain [3].

Another type of confinement mechanism is the Java security model [9], which is an example of language-based confinement of downloaded mobile code. Cryptographic methods for ensuring authenticity and integrity of programs are often suggested, but their main drawback is that they only solve the problem of confirming the author's identity and that the program has not been altered by someone else; they are of little help when the user does not trust the author.

Part II

Methods to improve system security

This page is intentionally left blank.

Paper E

The Remedy Dimension of Vulnerability Analysis

In *Proceedings of the 21st National Information Systems Security Conference*, pp. 91–98, Arlington, Virginia, Oct. 5–8, 1998.

This page is intentionally left blank.

The Remedy Dimension of Vulnerability Analysis

Ulf Lindqvist[1] Per Kaijser[2] Erland Jonsson[1]

[1]Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{ulfl, erland.jonsson}@ce.chalmers.se

[2]Siemens AG
DE-81730 München
Germany
Per.Kaijser@mchp.siemens.de

Abstract

This work is aimed at supporting system and information owners in their mission to apply a proper remedy when a security flaw is discovered during system operation. A broad analysis of the different aspects of flaw remediation has resulted in a structured taxonomy that will guide the system and information owners through the remedy identification process. The information produced in the process will help in making decisions about changes to the system or procedures. A selected vulnerability that was able to be removed using three different remedies is used as an example.

1 Introduction

When the discovery of a security flaw in a system has come to the knowledge of a party that risks suffering a direct loss if the vulnerability were to be exploited by an attacker, that party must decide what remedial action to take in order to remove the flaw from the system. The party in question is usually the organization that owns the system and/or the information stored and processed in the system. The work presented in this paper is aimed at supporting the owners in their mission to apply a proper remedy once a flaw is discovered, by providing them with a framework for remedy identification and analysis.

The traditional and most common situation in larger organizations is that the ownership of, or right to access, data and information stored and processed in a system coincides with the ownership of the system[3]. For smaller organizations, particularly for the many small and medium enterprises (SMEs), there is an increased interest in outsourcing, that is, letting a professional service provider own, manage and operate "your" system. The result is that information owners and system owners belong to different organizations. Still, both types of owners are vulnerable

[3]*Information owner* is normally referred to information with intellectual property rights (IPR). However, it can also be used to denote the individual or organization that is authorized to control access to a piece of information that might or might not be IPR protected. In the context of this paper, *information owner* covers both cases.

to security breaches. An information owner risks losing control of the information, and a system owner may be forced to pay damages or risks losing customers through a bad reputation.

To improve the security of IT systems, several guidelines and standards have been produced. These have been aimed at the different actors that have an effect on the security of the IT system, such as the manufacturers, procurers, managers, operators and users. For vendors and manufacturers, the functionality of the system and the development processes have been the target of standards (TCSEC [18], ITSEC [16], CC [6]) that specify criteria against which security evaluations can be made. These have also led to an increased interest in research on formal methods [1, 10, 15]. For procurers, baseline security documents have been created that give a minimum set of requirements on security features that an IT system should possess [20]. Managers, operators and users of an IT system need to follow certain rules in the form of security policies in order to minimize potential threats. For this purpose, guidelines and codes of practice have been specified [4, 8].

In spite of all these efforts, the number of security vulnerabilities can only be reduced, not eliminated. But what is more important: Only a minor part of the systems trusted with valuable information in trade, industry, public services and academia today are designed and implemented according to these criteria. Further, the ways in which hardware and software can be combined and interconnected are so complex that not even experts can fully understand how to avoid vulnerabilities. Reports about new vulnerabilities in computing systems are issued on almost a daily basis, for example in CERT advisories (such as [5]) posted on the Internet. It is true that security policies and recommendations for system and information owners play an important role, but they will not solve all weaknesses. To put it briefly:

- Vulnerabilities exist and will remain in all systems in operation.

Everyone should realize that, whatever precautions are taken, security flaws will be present in systems when delivered and in operation and that we need to form strategies for dealing with these flaws. However, this does not mean that we propose a penetrate-and-patch doctrine; it is still very important to try to eliminate as many security flaws as possible during early phases of system development, because the costs and risks associated with a repair increase dramatically later in the product life cycle. We want to emphasize that security should be considered throughout the entire system life cycle and that the efforts in different phases complement one another.

It is the owners of the information and the owners of the system who are primarily exposed to security risks. Therefore, our work aims at supporting the owners in this situation. The authors hope that the results of this work will also be beneficial to the security community in a wide sense, including international industrial consortia such as I-4[4] and ESF[5], incident response teams such as CERT and, of course, system and information owners.

[4]International Information Integrity Institute, a part of SRI Consulting which in turn is a subsidiary of SRI International. WWW: https://rome.isl.sri.com/i4/

[5]European Security Forum, Plumtree Court, London EC4A 4HT, England

In the following, Section 2 describes some earlier work in the field, while our analysis of the remedy dimension and our proposed taxonomy are presented in Section 3. Examples of remedies follow in Section 4, and some conclusions are drawn in Section 5.

2 Previous work

In our previous work on categorization[6] of intrusions, we made some general observations on the design of categorization schemes [14]. First, it is important to clearly state the attribute or view of the intrusion on which the categorization was based. We suggested the use of the term *dimension* for that view. Second, it is desirable to have mutually exclusive and collectively exhaustive categories, but this is often difficult, or even impossible, to fulfil. Third, the true value of such a taxonomy is that its formation and application enforces a structured analysis, which clarifies the matter and can generate new ideas.

The choice of the remedy attribute as a further dimension for categorization of vulnerabilities is natural and significant. It encompasses the whole life cycle of an IT system and focuses on the parties exposed to the risks that the vulnerabilities represent. Until now, little has been published on how to actually perform remedy planning and analysis, although several authors have observed the need for such activities. In an insightful paper, Kahn and Abrams stressed the importance of anticipating system security failures and planning for recovery and remediation [9]. Risk management, flaw remediation and evolutionary development is argued to provide more cost-effective and up-to-date security assurance than the TCSEC model of risk avoidance and static systems.

In the Common Criteria [6], there is a so called assurance family named *Life cycle support—Flaw remediation* (ALC_FLR). Earlier drafts of the criteria were studied by van Laenen [19], who points out the problem of re-evaluation when a change to the system has been made and also suggests two new requirements to be considered: *Mean time to remediation* and *Maximum time to remediation*. It should be noted, however, that this concerns only remedies provided by the developer.

3 Remedy analysis and taxonomy

In the terminology used in the field of dependable and fault-tolerant computing systems [12], the term *fault prevention* is used for methods that prevent faults from occurring or being introduced into a system. Actions that aim to reduce the presence of faults that already have been introduced fall under the category of *fault removal* and, especially, fault removal encountered during the operational phase of a system's life is called *corrective maintenance*. The present paper investigates corrective maintenance applied to faults that cause security failures or, in other words, *the remedy dimension* of security vulnerabilities.

[6]We prefer the term *categorization* to *classification*. The reason is that in the security field, the latter term is traditionally associated with a very specific dimension, namely, levels of confidentiality.

The remedy clearly depends on the nature of the vulnerability, but several new aspects must be carefully taken into account:

- What and who has caused the problem?
- Is there a possible way to remove the flaw?
- Will the changes introduce new vulnerabilities?
- Will the changes affect the quality of service?
- Will the changes actually remove the vulnerability?
- What will it cost to make the changes?
- What action should be taken—should any changes be made at all? If so, by whom?

The system owner can become aware of a vulnerability through internal experience as well as from external sources such as a product developer who provides a correction, an alert group such as CERT which may point out a vulnerability or from its own customers (the information owners). The information owner—the organization to which all users of the system belong—normally comes to know about a fault after having had practical experience with it, but may also be informed of it by others, for example the system owner.

First, the source of the vulnerability should be identified. This includes identification of the technical location of the fault in the system as well as identification of the organizational unit whose activities introduced the fault. The latter may for example be the producer of the product, the system owner or the information owner.

The next step is to turn to those believed to be able to provide a solution to the vulnerability. This is preceded by an analysis of possible locations of a remedy. The owner then informs the potential remedy providers about the problem, its location in the system or the process that is believed to cause it and, possibly, gives suggestions for how to overcome it. The result may be one or more proposed remedy actions that must be carefully analyzed with respect to their impact on the system and on system operation and use.

Before and after each of these steps, a decision must be made as to how to proceed. Is it worth continuing the remedy process? And, if it is considered worthwhile to continue and there are alternatives, which one(s) should be taken? All these decisions must be based on facts and be viewed in the light of economic constraints. To support and aid the system and information owners in their decisions, a four-stage remedy identification process is proposed. The properties to be identified and analyzed are:

- Fault location
- Remedy location
- Remedy provider
- Remedy impact

Each of these stages are described in detail below.

3.1 Identification of fault location

The point in the system structure, operation or use at which the fault causing the vulnerability is located is called the fault location. Since a vulnerability might consist of a combination of circumstances [13], it may not be possible to distinctively identify a single point as the location of the fault. Still, the analysis is a necessary starting point in the remedy process.

Our taxonomy on fault location is shown in Table 1. The taxonomy of computer program security flaws presented by Landwehr *et al.* [11] partly serves the same purpose, and our categorization can be viewed as a combination and extension of the dimensions they call *location* and *time of introduction*. The reader is urged to note that, although the same diagram is used for categorization of the remedy location in the following subsection, the remedy location does not always coincide with the fault location (see the example in Section 4).

Table 1. Taxonomy of fault location or remedy location.

Fault location or Remedy location	Product/ solution	Requirements
		Design
		Implementation
	Integration	Requirements
		Design
		Implementation
	Installation	External issues
		Internal issues
	Operation/ administration	Policy
		Monitoring and enforcement of policy
		Instructions
	Use	Policy
		Monitoring and enforcement of policy
		Instructions

We will describe some of the categories below, hoping that the names of the other categories will be self-explanatory. We consider a fault to be located in the *integration* if a component is vulnerable as part of one system but not of another. In the *installation* category, *external issues* are located outside the chosen system boundary (for example, physical protection) while *internal issues* are initial configuration parameters etc.

The second-level categories below the top-level categories *operation/administration* and *use* may also call for some clarification. A *policy* (or, more specifically, a *security policy*) is basically a set of rules stating what is allowed, what is not allowed and what must be done. *Monitoring and enforcement of policy* concern the management's efforts to make certain that the policy is respected and obeyed (a flaw may consist in the lack of enforcement of an existing and appropriate policy).

To help administrators and users to operate the system in a way consistent with the policy, *instructions* are required. For example, if the policy states that owners and users must take all reasonable action to prevent passwords from being revealed to an attacker, then the instructions should, for example, tell the system owner how to operate the system so that passwords are never sent in the clear via an untrusted network.

The result of this phase of the remedy identification process is the structural location of the cause of the fault. This will aid the owner in the next step of the process, namely, in determining where a remedy should be applied.

3.2 Identification of remedy location

Now the owner wishes to identify where in the system structure, operation or use a remedy should be applied. This step is based on the result of the fault location identification as well as on the type of flaw. The reason for this categorization is twofold: first, to be able to find the most appropriate remedy provider (see Section 3.3) and, second, to be able to estimate the remedy impact (see Section 3.4).

It should be noted that, from this stage and onward, several alternative proposed remedies to the same flaw may co-exist, each with its own location, provider and impact. The alternatives need not even be mutually exclusive, for example certain flaws might be of such a severe nature that an immediate remedy action is required until a more proper solution can be produced and applied. This stage in the process needs to be revisited as new proposals result from the owner's contacts with possible remedy providers.

The scheme for categorization of remedy location is identical to that of fault location, as shown in Table 1.

3.3 Identification of remedy provider

By remedy provider, we mean the party that needs to make the necessary changes in a component or process in order to remove the vulnerability. Identification of the remedy provider is naturally closely related to the remedy location, but is also related to the fault location. The organization behind the process in which the flaw was introduced is probably, but not necessarily, the best suited to provide information leading to a remedy. Furthermore, the originator of the fault has at least a moral, if not legal, responsibility to fix the problem.

The taxonomy of remedy provider is shown in Table 2. The categorization is based on the fact that the top-level categories often represent different organizations. *Product developers* are responsible for the production and maintenance (in terms of error correction and evolution) of the product. *Integrators* are responsible for the integration of products and solutions into a workable system. *Solution providers* are responsible for customer-specific solutions.

It should be noted that the *system* and *information owners* are also actively involved in the maintenance of the system, regardless of which party provides the actual remedy. Whereas a product developer often provides the technical solution for the removal of a fault, it is the system owner's administrators and operators that perform the update or ask the information owners to do it at their sites.

Table 2. Taxonomy of remedy provider.

Remedy provider	Product developers	Designers
		Implementors/ maintenance
	Integrators	
	Solution providers	
	System owners	Policy makers
		Administrators/ operators
	Information owners	Policy makers
		Administrators/ operators
		End-users

The identification of the remedy provider together with the result of the remedy location analysis helps the owner to identify the best way to remove the vulnerability. In the final step of the process, the results of the responses from the remedy providers (the suggested remedies) must be analyzed with respect to their impact on the system and the business.

3.4 Analysis of remedy impact

It is important to thoroughly analyze the impact of a suggested remedy on the system before applying it. If there are several different suggestions, the analysis could also help in choosing the optimum solution. This fourth step in the remedy process is based on our taxonomy of remedy impact as shown in Table 3. The reader should note that the category groups numbered 1 through 10 in Table 3 are not mutually exclusive. In fact, every remedy analyzed with respect to impact should be assigned to a single subcategory within each of these 10 different groups, as illustrated in Section 4.

The *primary technical effects* category concerns to what extent the remedy takes care of the vulnerability and what effects it has on the functionality of the component in which the changes were applied. If a particular instance of the vulnerability is removed but the basic flaw endures, we consider the vulnerability to be *partly eliminated*. On the other hand, a *provisionally eliminated* vulnerability means that the flaw itself is not repaired, but the situation in which it can be exploited is rendered impossible, for the time being (a typical example is the shutdown of a faulty service). The category of *secondary technical effects* concerns whether a *new vulnerability* is introduced, and how the *functionality* of unchanged (but dependent) parts is affected.

In addition to technical effects, there are also economic effects of a remedial action. The only such effects of interest to the decision-maker are of course the ones concerning their own organization. We have also separated the economic effects into primary and secondary, where the former are related to the immediate cost of

Table 3. Taxonomy of remedy impact.

Category				No.
Remedy impact	Primary technical effects	Target vulnerability eliminated	Provisionally	1
			Partly	
			Completely	
		Functionality (of changed parts)	Impaired	2
			Unchanged	
			Improved	
	Secondary technical effects	Severity of identified new vulnerability	High	3
			Medium	
			Low	
			None	
		Functionality (of unchanged parts)	Impaired	4
			Unchanged	
			Improved	
	Primary economic effects (related to performing the change)	Cost for internal resources	High	5
			Medium	
			Low	
		Cost for external resources	High	6
			Medium	
			Low	
			None	
	Secondary economic effects (after the change)	Cost in human resources	Increased	7
			Unchanged	
			Decreased	
		Processing time	Increased	8
			Unchanged	
			Decreased	
		Cost in computer resources	Increased	9
			Unchanged	
			Decreased	
	Time to remediation		Long	10
			Medium	
			Short	

performing the change, while the latter concern the long-term consequences after the change is made. The immediate cost consists of the internal cost from workload among the owner's own staff and the cost of paying for equipment, services or solutions from external sources (depending on the contract situation). An example of long-term impact is the case in which the change results in a certain additional working time for some administrator or user tasks. On the other hand, if the system services are faster and simpler after the change, a secondary economic effect would be decreased processing time.

The time the owner needs to wait for the remedy is of course a significant factor in the decision process, since the system and information are vulnerable until the remedy has been applied. The category of *time to remediation* with the rough subcategories *long, medium* and *short* is meant to be used in a relative rather than absolute sense.

4 A vulnerability and examples of remedies

In this section, we present a well-known vulnerability and some examples of remedies in order to illustrate and exemplify the taxonomy presented in Section 3. For the sake of brevity, the technical description of the flaw is here kept to a minimum; the interested reader will find more detailed descriptions in the references cited.

4.1 The Unix X terminal emulator logging vulnerability

The X Window System terminal program *xterm*, running with the effective user id of *root* (super-user) in some Unix variants, had a flawed logging facility that could be used to create an arbitrary new file or modify any existing file by appending an arbitrary set of data to it [2, 5, 13].

Our first step is to identify the vulnerability. It turns out that, in the procedure implementing the logging facility, *xterm* makes certain critical system calls with the privileges of *root* instead of with the privileges of the user invoking the program. In this case, the flaw must clearly be categorized as being located in the product design, since the program presumes *root* privileges but is not designed with the precaution needed for privileged programs.

A correction can either be provided by the product developers, product maintenance or even the system owner, giving rise to three alternative remedy actions for us to consider:

a) Remove the super-user privileges from *xterm*

b) Disable the logging facility of *xterm*

c) Rewrite *xterm* according to “the principle of least privilege”

We thus note that there is only one entry for fault location whereas there are three for both remedy location and remedy provider.

4.2 Remedy a: remove the super-user privileges

The quickest and easiest way to remedy the *xterm* flaw is to clear the set-user-id flag of the program, that is, to remove its super-user privileges. However, there is a reason why *xterm* was installed with those privileges: it needs to change the owner of the pseudo-terminal slave device, an action which requires *root* access[7]. When testing this remedy on a SunOS 4.1.2 system in a university computer security laboratory, we found the following:

- To the terminal user, *xterm* appears to function normally.
- The pseudo-terminal slave device, to which *xterm* connects, is still owned by *root*. In order for the device to be readable and writable for the user, it must be so for a group of users or even all users. Consequently, a new vulnerability is introduced, primarily threatening the user rather than the system owner.
- If the system logging file */etc/utmp* is writable only to *root* (which was the case in the test system, since a writable */etc/utmp* constitutes another vulnerability, see [13]) the terminal connection is not reported by some programs that list the users logged on to the system, for example *who*.

We can only consider the vulnerability to be provisionally eliminated by this remedy because, for example, an uninformed administrator who discovers the device and logging problems might assume that the privileges have been turned off by mistake or by accident. If the administrator turns the privileges back on, the system would again be vulnerable. Any economic effects of this simple remedy are negligible.

With the information at hand, we make the following categorization of this remedy:

Fault location: Product: Design

Remedy location: Installation: Internal issues

Remedy provider: System owners: Administrators/operators

Remedy impact: see Table 4.

4.3 Remedy b: disable the logging facility

The patches distributed by the developers in response to the CERT warning disable the logging facility. The function of the logging facility is to provide a simple means for the user to save the terminal output in a file, a service evidently impaired by this remedy. Another disadvantage of this approach is that it does not solve the basic problem, namely, that *xterm* unnecessarily makes all its system calls with *root* privileges, leaving the system vulnerable to a number of other, similar and yet undiscovered flaws (see the discussion on Remedy c below).

[7]There are Unix variants in which *xterm* works without special privileges. In such cases, this particular vulnerability never existed, and the remedy discussion is a non-issue.

Table 4. Impact analysis of the three suggested remedies.

Impact of remedy		a	b	c	No.
Primary technical effects	Target vulnerability eliminated	Provision-ally	Partly	Completely	1
	Functionality	Unchanged	Impaired	Unchanged	2
Secondary technical effects	Severity of identified new vulnerability	Medium	None	None	3
	Functionality	Impaired	Unchanged	Unchanged	4
Primary economic effects	Cost for internal resources	Low	Low	Low	5
	Cost for external resources	None	None	*Depends*	6
Secondary economic effects	Cost in human resources	Unchanged	Unchanged	Unchanged	7
	Processing time	Unchanged	Unchanged	Unchanged	8
	Cost in computer resources	Unchanged	Unchanged	Unchanged	9
Time to remediation		Short	Medium	Long	10

Fault location: Product: Design

Remedy location: Product: Implementation

Remedy provider: Product developers: Implementors/maintenance

Remedy impact: see Table 4.

4.4 Remedy c: rewrite the program

Regardless of whether the concern is reliability, safety or security, it has long been known that critical regions of software should be as small and simple as possible, since complex programs are error-prone. It is also well-known in the security community that no action should be performed with higher privileges than those absolutely necessary to complete the task. These two rules are known as the principles of “economy of mechanism” and “least privilege”, respectively [17]. The flawed version of *xterm* is a large program running with constant super-user privileges, although such powers are necessary only for a small fraction of its duties. Unfortunately, *xterm* is not the only Unix utility that violates both of the above principles; *sendmail* is another notorious example.

This remedy action suggests *xterm* to be rewritten in a defensive, security-conscious programming style, following “best practice” guidelines for privileged programs [3, 7]. In this case, not only the logging flaw would be eliminated, but also any similar security flaws in other parts of the code. This is a relatively expensive solution, however, and there is always a risk that new security flaws and other bugs are introduced when such a large piece of software is modified extensively.

Fault location: Product: Design

Remedy location: Product: Design

Remedy provider: Product developers: Designers

Remedy impact: see Table 4.

4.5 Discussion

In our example, a single fault was able to be removed using three different remedies with different impacts. Thanks to the taxonomy, the three remedies can be easily compared. It gives the system owner well-founded facts for making a decision, and we see that even if remedy c is preferred, either a or b could probably be accepted as a temporary solution.

How different owners would categorize a certain aspect of a given remedy may appear somewhat subjective. It should be remembered that each owner needs to find and validate remedies according to site-specific circumstances. Categorizations might therefore vary for different owners. Future development of categorization criteria could perhaps further help users of the taxonomy.

5 Conclusions

The purpose of the remedy identification process defined and described in this paper is to aid and support those exposed to the threats—the system and information owners—in how to proceed in their decision process, rather than to design a remedy for a given vulnerability. Our process consists of four phases; locating the fault, locating the remedy, identifying the provider of the remedy and analyzing the impact of the remedy. Each of these phases requires a taxonomy for easy categorization. The paper has described four taxonomies that are the core of the analysis in each of these steps. The remedy location and remedy provider phases can be iterative, since the result of each may require an update of the other. This is different from the first phase (fault location), which only serves as initial input, and the last phase (remedy impact), which analyzes the output from the two middle phases. The final result—the impact of each proposed remedy—is the desired outcome on which the owner can base a sound decision as to how to proceed.

References

[1] Marshall D Abrams and Marvin V Zelkowitz. Striving for correctness. *Computers & Security*, 14(8):719–738, 1995.

[2] Taimur Aslam, Ivan Krsul, and Eugene H Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Information Systems Security Conference*, pages 551–560, Baltimore, Maryland, October 22–25, 1996. National Institute of Standards and Technology/National Computer Security Center.

[3] Matt Bishop. How to write a setuid program. *;login: (The USENIX Association Newsletter)*, 12(1):5–11, January/February 1987.

[4] British Standards Institution. *Code of Practice for Information Security Management*, 1995. BS 7799.

[5] CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. *xterm Logging Vulnerability*, November 11, 1993. CERT Advisory CA-93:17.

[6] Common Criteria Implementation Board. *Common Criteria for Information Technology Security Evaluation*, May 1998. Version 2.0. See also ISO/IEC 15408.

[7] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, second edition, 1996.

[8] INFOSEC Business Advisory Group. *The IBAG Framework for Commercial IT Security*, September 1993. Version 2.0.

[9] Jay J Kahn and Marshall D Abrams. Contingency planning: What to do when bad things happen to good systems. In *Proceedings of the 18th National Information Systems Security Conference*, pages 470–479, Baltimore, Maryland, October 10–13, 1995. National Institute of Standards and Technology/National Computer Security Center.

[10] Carl E Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[11] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[12] Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, 1992.

[13] Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. Analysis of selected computer security intrusions: In search of the vulnerability. Technical Report 275, Department of Computer Engineering, Chalmers University of Technology,

Göteborg, Sweden, 1996. Presented at NORDSEC – Nordic Workshop on Secure Computer Systems, Göteborg, Sweden, November 7–8, 1996.

[14] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, California, May 4–7, 1997. IEEE Computer Society Press, Los Alamitos, California.

[15] Peter G Neumann. Architectures and formal representations for secure systems. Technical Report SRI-CSL-96-05, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1996.

[16] Office for Official Publications of the European Communities. *Information Technology Security Evaluation Criteria*, June 1991. Version 1.2.

[17] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[18] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[19] Filip van Laenen. Pedigree and credentials, remediation and legal aspects to gain assurance in IT products and systems. Master's thesis, Katholieke Universiteit Leuven, Belgium, and Norges Tekniske Høyskole, Norway, 1995.

[20] X/Open Company Ltd., UK. *X/Open CAE Specification: Baseline Security Services (XBSS)*, 1995. X/Open Document Number C529.

Paper F

An Approach to UNIX Security Logging

In *Proceedings of the 21st National Information Systems Security Conference,* pp. 62–75, Arlington, Virginia, Oct. 5–8, 1998.

This page is intentionally left blank.

An Approach to UNIX Security Logging

Stefan Axelsson Ulf Lindqvist Ulf Gustafson[*] Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
{sax, ulfl, erland.jonsson}@ce.chalmers.se

Abstract

Host-based intrusion detection and diagnosis systems rely on logged data. However, the logging mechanism may be complicated and time-consuming and the amount of logged data tends to be very large. To counter these problems we suggest a very simple and cheap logging method, lightweight logging. *It can be easily implemented on a UNIX system, particularly on the Solaris operating system from Sun Microsystems. It is based on logging every invocation of the* exec(2) *system call together with its arguments. We use data from realistic intrusion experiments to show the benefits of the proposed logging and in particular that this logging method consumes as little system resources as comparable methods, while still being more effective.*

1 Introduction

The main problem with collecting audit data for a log is not that it is difficult to collect enough data, but rather that it is altogether too easy to collect an overwhelming amount of it. The sheer volume of the audit data is the immediate reason that logging is often considered a costly security measure. The collection of a large amount of audit data places considerable strain on processing and storage facilities, not to mention the time that must be spent, either manually, or aided by computers, sifting through the logs in order to find any breaches of security. The trade-off is between logging too much, and being drowned in audit data, or logging too little to be able to ascertain whether indeed a breach has taken place [1, 7, 8].

Most UNIX installations do not run any form of security logging software, mainly because the security logging facilities are expensive in terms of disk storage, processing time, and the cost associated with analysing the audit trail, either manually or by special software. In this paper we suggest a minimal logging policy,

[*]Author's present address: Ericsson Mobile Data Design AB, S:t Sigfridsgatan 89, SE-412 66 Göteborg, Sweden, Ulf.Gustafson@erv.ericsson.se

lightweight logging, based on the one single system call *exec(2)*[1]. We use empirical data derived from practical intrusion experiments to compare the lightweight logging method with a few other simple methods. It is concluded that the *intrusion traceability* of the proposed logging method is superior to that of the comparable methods.

2 The purpose of logging

The main purpose of logging for security reasons is to be able to hold users of the system accountable for their actions [14]. Logging is one of two basic requirements for this, the other being identification/authentication. It is impossible to hold a user accountable for some action indicated in the logs if it can not be excluded that someone else has “masqueraded” as the user.

Even though less than perfect accountability may result from the mere existence of a log, the logging mechanism serves other useful purposes [9]:

- It makes it possible to review the patterns of use, of objects, of users, and of security mechanisms in the system and to evaluate the effectiveness of the latter.
- It allows the site security officer to discover repeated attempts by users of the system to bypass security mechanisms.
- It makes it possible for the site security officer to trail the use (or abuse) that may occur when a user assumes privileges greater than his or her normal ones. While this may not have come about as a result of a security violation, it is possible for the user to abuse his or her privileges in the new role.
- The knowledge that there is a mechanism that logs security relevant actions in the system acts as a deterrent to would-be intruders. Of course, for a security logging policy to be effective in a deterring capacity, it must be known to would-be intruders.
- The existence of a log makes “after the fact” damage assessment and damage control easier and more effective. This in turn raises user assurance that attempts to bypass security mechanisms will be recorded and discovered. Logs are a vital aid in this aspect of contingency resolution [6].

In UNIX environments in general, and in the systems under discussion in particular, some of the above mentioned aims cannot be fully realized. For instance, once a user has assumed super-user privileges in a UNIX system, he (or she) then typically has the power to turn off logging, alter existing logs, or subvert the running logging mechanism to make it provide a false record of events. Furthermore, UNIX systems typically do not use sufficiently strong methods of authentication to make it possible to hold a user accountable on the grounds of what appears in an

[1]We use *exec(2)* as a generic name denoting all kernel system calls implementing the traditional UNIX *exec* functionality. In most UNIX versions (including SunOS 4.x), only *execve(2V)* is a kernel system call, while other variants of *exec* are provided as library routines.

audit trail. In either case, the knowledge that a security violation has taken place is to be much preferred to the situation in which a breach of security has taken place, but gone unnoticed.

3 Lightweight logging

3.1 Definition

We strive for a logging policy that would allow us to detect and trace attacks against our system, i.e. that could be incorporated into an intrusion-detection system (IDS) and by its mere simplicity facilitate the postmortem intrusion-detection task. Our main purpose is to provide an audit trail from which the security officer can establish exactly what occurred, and how it occurred, rather than merely being able to detect that some sort of significant event has taken place. A logging policy should meet the following requirements:

1) The system should be transparent to the user, i.e. it should behave in the manner to which he has been accustomed.

2) Since system resources are always sparse, as little as possible should be consumed. This means minimizing the use of storage space, processing time, and time spent by the administrator.

3) While meeting the above requirements, sufficient data should be recorded to maximize our chances to detect and trace any, and all, intrusions.[2]

We have found that it would be possible to trace *most* of the intrusions presented in this paper by logging relevant information about each *exec(2)* system call made in the system. Since the number of *exec(2)* calls roughly corresponds to the number of commands issued by the user, the amount of audit data should be in the same order as that of *pacct*,[3] while recording more security relevant data than *pacct* does.

Unfortunately one cannot configure the SunOS 4.x BSM audit mechanism (see Section 4.2) to generate one record for every command executed, like *pacct.* If one wishes to record every invocation of the *exec(2)* system call, one must audit all the system calls in that audit class, in total 15 different system calls. This may produce more audit data than we care to store and process. Furthermore, the arguments to the *exec(2)* call are not recorded, and that fact reduces the quality of the audit data considerably.[4]

[2]Our intrusion data were collected on the premise that the attackers operated as insiders. In order to log data relevant to tracing intrusions from outsiders, a network security tool such as *Tcp wrapper* could be combined with our suggested logging mechanism. See [15] for a description of *Tcp wrapper.*

[3]See Section 4.2.2 for a more detailed presentation of *pacct*, the UNIX process accounting facility.

[4]Both these restrictions have been lifted in SunOS 5.x.

Table 1. Penetration scenario.

Step	Shell command	Comment
1	`$ ln -s /u/vulnerable-file -i`	Make link to a *setuid* root shell script.
2	`$ -i`	Invoke the shell script as -i.
3	`root#`	The user now has an interactive root shell.

3.2 Example

The example in Table 1 is a classic UNIX intrusion scenario that can be exploited to gain super-user privileges. This security flaw was present in SunOS 4.1.2, the version on which the first experiment was conducted. In order for the flaw to exist, there must be a shell script somewhere on the system that is *setuid* or *setgid* to someone, i.e. it is run with the privileges of its owner, or group, not its caller. The flaw is exploited by the intruder calling the shell script via a symbolic link, and this results in the intruder gaining access to an interactive command interpreter, henceforth called shell.

This flaw comes about as a result of a bug in the UNIX kernel. When the kernel executes the shell script, it first applies the *setuid* bit to the shell and then calls the shell with the filename of the shell script as the first argument. If this filename is "*-i*," the shell mistakes this for the command line switch to start in interactive mode. In later versions of SunOS, 5.x this problem has been corrected.[5]

To analyse what needs to be recorded in order to trace this intrusion, we look at the system calls made when exploiting this flaw. Steps 1) and 2) in Table 2 detail the system calls that are invoked when running *ln(1V)* and *sh(1)*. Both these commands are executed by a shell that performs the *fork(2V)*/*exec(2)* sequence, which is a prerequisite for all command execution in UNIX.

We outline our suggestion for what information to be included in the audit record in Table 3.

Perhaps the only field in Table 3 that merits further comment is the field "log UID". We propose that each user be assigned a unique identifier when he logs into the system. This identifier does not change for the duration of the session, even if the user's real UID changes, as a result of an invocation of the command *su(1V)* for instance. The existence of the "log UID" field makes it easier to trace the commands invoked by each user, although it is not strictly necessary. The same information may be distilled from complete knowledge about the branch on the process tree from the root (login) to the leaf (the current process). The log UID simplifies this task; we have borrowed the concept from C2 auditing [9, 14].

From the above audit records it becomes clear that user *5252* executed a *ln(1V)* command that made a soft link with the name `-i` to the shell script, and that the

[5]The filename is no longer passed as the argument to the shell. Instead, the shell is passed a filename on the form */dev/fd/X* where *X* refers to the file descriptor of the already open file. See [11, p. 69] for an introduction to the */dev/fd* interface.

Table 2. System calls.

Step	System calls invoked	Comment
1	fork() execve("/bin/ln", "ln", "/u/vul...", "-i") stat ("-i", 0x9048) = -1 ENOENT symlink ("/u/vulnerable-file", "-i") = 0 close (0) = 0 close (1) = 0 close (2) = 0 exit (0) = ?	Make link to a *setuid* root shell script. (ENOENT = No such file or directory.)
2	fork() execve("-i", "-i", ...) sigblock (0x1) = 0 sigvec (1, 0xf7fff94c, 0xf7fff940) = 0 sigvec (1, 0xf7fff8d4, 0) = 0 sigsetmask (0) = 0x1 sigblock (0x1) = 0 .	Invoke the shell script as -i. The shell starts with some calls to *sigblock*, *sigvec*, and *sigsetmask*. The system calls that are executed are then dependent on the input to the shell.
3	`root#`	

Table 3. The proposed system call logging.

Information recorded for *execve(2)*	Step 1 (ln) (example)	Step 2 (sh) (example)
a record creation time stamp	xxxx1	xxxx2
the real UID	5252	5252
log UID	YY	YY
effective UID	5252	0
real GID	11	11
effective GID	11	11
process ID	1278	1280
parent process ID	1277	1277
filename	/bin/ln	./-i
current working directory	/u/hack	/u/hack
root directory	/	/
return value	success	success
argument vector to *execve(2V)*	"-s", "/u/vulnerable-file", "-i"	"-i"

user then invoked the shell script via the link.

If we look in detail at the above, it becomes clear that we need only log the invocations of the *execve(2V)* system call made by this user to trace the intrusion. Since we log the argument vector (argv) to the *execve(2V)* call, we need not log the symlink call separately. As can be seen above, that information is recorded when we log the argument vector to the *ln(1V)* command. We have all the data necessary to trace this specific intrusion back to the user that performed it.

In essence, the proposed logging scheme, creates one audit record per command issued. This also holds true for regular process accounting with *pacct*, but there are several differences:

- By logging the start of execution of every command instead of the end of execution, we have a better chance of detecting an ongoing intrusion attempt. This is especially true if we consider long running commands that crack passwords or search the filesystem for example. Furthermore, the command that commences the intrusion is logged. This is far from certain if we delay logging until the command has completed execution, since this may already have turned off auditing etc.

- The most severe security intrusions in UNIX environments are often performed by tricking a setuid program into performing some illicit action. By logging both the real and effective UID every time a command is to be run, we can detect many such intrusions.

- Regular accounting logs the first eight characters of every finished command but, since programs can be copied and renamed, this is easy to circumvent. By logging the full path name of every command, together with all arguments, the proposed auditing policy is much more difficult to trick.

4 The logging during the data collection experiment

4.1 The experiment

During the years 1993–1996, we performed four intrusion experiments in UNIX systems [5, 10]. The original goal of these experiments was quantitative modelling of operational security, that is, we tried to find measures for security that would reflect the system's "ability to resist attacks". In order to do so, extensive logging and reporting were enforced and a great deal of data were generated. We believe that these data are also useful for the validation of the logging policy proposed in this paper.

During the experiments a number of students (13, 24, 32, and 42, respectively) were allowed to perform intrusions on a system in operational use for laboratory courses at the Department of Computer Engineering at Chalmers in Sweden. The system consisted of 24 SUN ELC disk-less workstations and a file server, all running SunOS 4.1.2 or SunOS 4.1.3_U1. The system was configured as delivered, with no special security enhancing features [3].

The attackers, who worked in pairs, were given an account on the system—thus, they were “insiders”—and were encouraged to perform as many intrusions as possible. Their activities were limited by a set of rules meant to avoid disturbing other users of the system and to ensure that the experiment was legal. Further details are found in the references cited above.

4.2 The logging

There were three main classes of accounting in the experiment system that were active:

Connect time accounting is performed by various programs that write records into */var/adm/wtmp*, and */etc/utmp* [12]. Programs such as *login(1)* update the *wtmp(5V)* and *utmp(5V)* files so that we can keep track of who was logged into the system and when he was logged in.

Process accounting is performed by the system kernel. Upon termination of a process, one record per process is written to a file, in this case */var/adm/pacct*. The main purpose of process accounting is to provide the operator of the system with command usage statistics on which to base service charges for use of the system [12].

Error and administrative logging is primarily performed by the *syslogd(8)* daemon [12]. Various system daemons, user programs, or the kernel log abnormal, noteworthy conditions via the *syslog(3)* function. These messages end up in the files */var/adm/messages* and */var/log/syslog* on the experiment system.

Another class of logging designed with security in mind is the SunOS BSM (Basic Security Module) logging sub system [12]. This logging facility is said by Sun Microsystems to conform to the requirements laid forth in TCSEC C2, even though the SunOS BSM has not been formally certified according to TCSEC. This logging mechanism was not active in the experiment system.

The system logging and accounting files in the experiment system thus consist of */var/adm/wtmp*, */etc/utmp*, */var/adm/pacct*, */var/log/syslog*, and */var/adm/messages*.

4.2.1 Connect time accounting Various system programs enter records in the */var/adm/wtmp* and */etc/utmp* files when users log into or out of the system. The purpose of the *utmp(5)* file is to provide information about users currently logged into the system, and the entry for the particular user is cleared when he logs out of the system. The *wtmp(5V)* file is never modified in this manner; instead, when the user logs out, another entry is made containing the time he left the system. The *wtmp(5V)* file thus contains a record of each user as he entered and exited the system.

The *wtmp(5V)* file also contains information indicating when the system was shut down or rebooted and when the *date(1V)* command was used to change the system time.

The *wtmp(5V)* records contain the following information:

- The name of the terminal on which the user logged in.
- The name of the user who logged in.
- The name of the remote host from which the user logged in, if any.
- The time the user logged into or out of the system.

4.2.2 Process accounting by pacct The process accounting system is, as mentioned before, designed to provide the operator of the system with command usage statistics on which to base service charges for use of the system. The *pacct* system is usually activated by the *accton(8)* command when booting the system. When active, the UNIX kernel appends an audit record to the end of the log file, typically */var/adm/pacct*, on the termination of every process. The audit record contains the following fields:

- Accounting flags; contains information indicating whether *execve(2V)* was ever accomplished and whether the process ever had super-user privileges.
- Exit status.
- Accounting user.
- Accounting group ID.
- Controlling terminal.
- Time of invocation.
- Time spent in user state.
- Time spent in system state.
- Total elapsed time.
- Average memory usage.
- Number of characters transferred.
- Blocks read or written.
- Accounting command name; only the last eight characters of the filename are recorded.

4.2.3 Error and administrative logging Beside the functions described above, many user and system programs use the logging facility provided by the *syslog* service. At system start-up, the logging daemon *syslogd(8)* is started, and processes can then communicate with *syslogd(8)* via the *syslog(3)* interface.

The messages sent to *syslog(3)* contain a priority argument encoded as a *facility* and a *level* to indicate which entity within the system generated the log entry and the severity of the event that triggered the entry. The *syslog* service is configured to act on the different *facilities* and *levels* by appending the message to the appropriate file, write the message on the system console, notify the system administrator, or send the message via the network to a *syslogd(8)* daemon on another host. In the experiment system, *syslog* was configured to append all messages to */var/adm/messages* and debug messages from *sendmail(8)* to */var/log/syslog*.

5 Evaluation of the intrusion data with respect to different logging methods

During our experiments we defined an intrusion as the successful performance of an action that the user was not normally allowed to perform. About 65 intrusions were made, most of them already known by the security community, e.g., by CERT.[6] However, while CERT only informs about what system vulnerability was used for a specific intrusion, our experiment yielded further data. The greatest advantage of our intrusion data is that we know *exactly how* the intrusion was performed, which obviously is of specific interest when discussing logging for intrusion-detection purposes. Therefore, we categorize our intrusions according to what kind of audit trail they leave. For each intrusion class, we discuss the possibility of detecting an attack with normal system accounting or monitoring, and by means of using the suggested lightweight logging method.

The intrusions are categorized in ten broad classes as presented in the rest of this section. For the sake of brevity, only one typical intrusion in each class is described in detail, while the others are outlined. The discussion is structured under the following headings:

System logging; the logging performed by the kernel and system processes such as *init(8)*, and *pacct(8)*.

Application program logging; the logging performed by *application programs*, such as *su(1V)* etc.

Monitoring resource utilization; some attacks result in abnormal load on CPU, disk, network, etc., and this can be monitored, and anomalous behaviour can be detected. In practice, many intrusions are detected because the users of the system report that it acts “funny” in this respect.

Lightweight logging; discussion of the validity of the recorded information related to the suggested lightweight logging policy (logging *execve(2V)*).

5.1 Class C1: Misuse of security-enhancing packages

There are several programs available that help the supervisor of a UNIX system to increase security by testing for known security problems. These programs can of course also be (ab)used by an attacker to learn about existing flaws in the attacked system.

[6]See *http://www.cert.org* for information about CERT, and the advisories they publish.

Crack The target system did not enforce password shadowing, so any user was able to read out the encrypted password values and mount a dictionary attack by obtaining and executing the publicly available password guessing program *Crack*.

System logging: The execution of these kinds of programs, which are well-known packages consisting of several subprograms, leaves distinct patterns of multiple entries in the *pacct* file that are easy enough to detect and trace, provided the commands are not renamed.

Application program logging: N/A.

Resource utilization: Possibly massive disk (network if NFS) and CPU utilization.

Lightweight logging: The program name is recorded and saved with its arguments. Each spawned subprogram in the collection will also be recorded together with its arguments. This approach gives more accurate information considering the patterns in the log file.

COPS Intended to be used by system administrators to find security problems in their UNIX installations, *COPS* is a publicly available package that can also be used by attackers. It consists of a set of programs, each of which tries to find and point out potential security vulnerabilities.

Password generation rules When assigning passwords to other students attending courses at the department, a program that randomly creates 7-character lower-case passwords is used. To make the passwords pronounceable and thus easier to memorize, the program makes every password contain 3 vowels and 4 consonants in a distinct pattern. Unfortunately, it turns out that this pattern severely limits the randomness of user passwords and makes exhaustive search feasible. The attackers compiled a dictionary that satisfied the password generation rules and then ran *Crack*.

Conclusion: Both system logging and our proposed logging policy are capable of detecting the use of the security-enhancing packages encountered during the experiment.

5.2 Class C2: Search for files with misconfigured permissions or *setuid* programs

A general search of the filesystem for files for which the attacker has write permission, or files that are *setuid* to some user is often a step performed by the security packages mentioned in C1. We list it as a separate class since many of our attackers performed such a search when first trying to breach security. These attacks also have in common that they are resource-intensive in terms of (network) disk traffic and may be detected because of this.

Search for files with public write permission The target in this attack was carelessly configured permissions on various files in the system, especially system files or user configuration files. Files with public write permissions can be modified by arbitrary users, compromising the integrity of the system. During the experiment we chose not to count general searching as a breach in itself; to regard the action as a breach, we demanded a detailed description of how to exploit the vulnerable file.

System logging: If the *find(1)* command is used, traces of that can be found in *pacct*. However, since the arguments to the *find(1)* command are not available, it is doubtful whether the security administrator can tell the difference between benign uses of *find(1)* and uses that are consistent with an ongoing intrusion attempt.

Application program logging: N/A.

Resource utilization: Possibly massive disk (network if NFS) and CPU utilization.

Lightweight logging: The arguments to *find(1)*, together with the command name, are logged, making it possible to discover what the attacker is searching for. If the attacker tries to hide the name by making soft or hard links to the find command, the links are also traceable.

Search for *setuid* files Wrongly configured *setuid* files may compromise overall system security, especially *setuid* shell scripts or *setuid* programs with built-in shell escapes. In this case we also demanded a detailed description of how to exploit the vulnerable file.

Conclusion: The suggested logging policy can detect usage of *find(1)* for purposes of searching for files as above. System logging, on the other hand, has a difficult time differentiating between suspect and legitimate uses of *find(1)*.

5.3 Class C3: Attacks during system initialization

As the experiment system was configured, it was possible to attack it by halting it during system initialization. The single-user root privileges thus obtained were able to be further exploited to become multi-user root.

Single user boot It was possible to boot the clients from the console to single-user mode. This was possible because */etc/ttytab* was not set up to secure console login. Hence, it was possible to modify an arbitrary filesystem on the client. (Although the clients were disk-less, their root directories were mounted via NFS from the file server.)

System logging: Through the records in */var/adm/wtmp*, */var/adm/pacct* and */var/adm/messages* it is possible to tell when the machine was rebooted after it has come up in multi-user mode again. However, the commands executed in single-user mode are not logged, since logging is not active in single-user mode.

Application program logging: N/A.

Resource utilization: limited.

Lightweight logging: It is possible to log the actions performed during single-user mode, but this is not normally done. The single-user mode is viewed as a transient administrative state, employed for administrative duties or system repair. As such, the resources necessary for running the logging mechanism may be unavailable. In our case, all but the root partition was mounted read only, making it difficult to store the log on disk.

Inserting a new account into the *LEADING_SLASHetc/passwd* file This is primarily a method to become multi-user root. After becoming single-user root, it is possible to insert a new account in the password file. It is then possible to log into that account when the client comes up in multi-user mode.

***Setuid* command interpreter/program in root filesystem** As single-user root, it is possible to make a copy of a command interpreter (shell), change the owner of the copy to an arbitrary user (for example root) and set its *setuid* flag. Then, when the host has entered multi-user mode, the attacker need only execute the copied shell to take the identity of its assigned owner. This method was primarily used to become multi-user root.

File server intrusion through *setuid* program Although the clients were all disk-less, all modifications on the clients root filesystems were also available to the users on the file server. In this way, it was possible to become another user by executing a *setuid* program as described in the preceding intrusion scenario.

Conclusion: Since the system is not fully operational during initialization, it is difficult for both methods to detect, let alone trace, any intrusion attempt. It would be technically difficult to design a logging mechanism that would function under such circumstances, since we are in effect giving away super-user privileges to anyone who happens to walk by. This turns this attack into an “outsider” attack, and we must monitor physical access to the computer room in order to have a chance of catching the intruder.

5.4 Class C4: Exploiting inadvertent read/write permissions of system files

As the experiment system was configured, many critical system files and directories were set up with inadvertently lax access permissions. This made it possible for the attackers to modify critical files to subvert system programs, or, in one case, to read data to which the attackers should not have had access.

YP configuration error NIS (see *yp(3R)*)was installed according to the manual, meaning that it was necessary to initialize */var/yp* before bringing the machine up to multi-user mode. Doing this, and neglecting that an appropriate *umask* is not activated by default in single-user mode, may result in dangerous file permission settings on files created under those circumstances. The NIS server configuration database, */var/yp*, in the target system had permission mode 777, that is, writable and readable for all.

/etc/utmp **writable** The default on the experiment system was for */etc/utmp* to be writable for all users. This makes it possible for an intruder to hide from appearing in the output of commands such as *who(1)* and *users(1)* but, more importantly, it allows a user to alter system files. This is accomplished by editing */etc/utmp* and then issuing a *setuid* command (*write(1)* for instance) that uses */etc/utmp* to find its output file.

System logging: *pacct* will show only that some user issued, for instance, a *write(1)* command that looks benign enough. There is certainly no way of differentiating this use of *write(1)* from ordinary legitimate uses of that command.

Application program logging: N/A.

Resource utilization: Limited.

Lightweight logging: Unfortunately, if the command the user uses to manipulate */etc/utmp* does not stand out in the audit trail, this kind of intrusion will be difficult to trace. Since we log all arguments to commands issued, our chances of catching the modifying command will have increased substantially in comparison with the chances that *pacct* has.

Crash the X-server When a client wishes to connect to the X-server on the local machine, the client looks for a UNIX domain socket at a predefined location in the filesystem (*/tmp/.X11-unix/X0* in the experiment system). This socket and its directory were world-writable in the experiment system and, as a result, the user could remove the socket and thus hang the X-server. By replacing the socket with a non-empty directory and forcing the X-server to restart, the directory is unlinked, and the files are left “hanging”. This would require the supervisor to manually fix the system on the next boot with *fsck(8)*.

Reading of “mounted” backup tapes In the experiment system, a backup tape was always present in the tape streamer awaiting that night’s backup run. Since the backup tapes were constantly reused (a fairly common policy in many installations) and the tape streamer’s device file had world-read permissions set, it was possible for the attackers to read the previous week’s backup tapes. This enabled them to read files that they would not normally be allowed to read.

Conclusion: Misuse of system files with erroneous permissions is often detectable by the proposed logging scheme. This misuse often manifests itself as a suspect argument to a user command, that is, the command accesses a file that it should not normally access. System logging has very slim chances of detecting this, since arguments are not logged.

5.5 Class C5: Intercepting data

Due both to configuration errors and the nature of certain UNIX system applications, it was possible for the attackers to intercept communication between users and the experiment system.

Snooping the X-server A publicly available program called *xkey* was used to listen to traffic to the X-server. This program tries to connect to the X-server on the target machine and, if the request is granted, makes it possible to intercept any keystrokes typed by the console user at the target machine.

Frame buffer grabber If a person can log into a SunOS 4.1.x workstation from a remote host, he or she may be able to read the contents of the console's video RAM memory. The frame buffer character special file, */dev/fb,* is per default writable and readable for everyone.

Ethernet snooping Many typical UNIX clients for remote login transmit authentication information in the clear via the network. It is thus possible for someone with access to the network (network topology and technology permitting) to eavesdrop on this traffic and learn passwords, etc. In most UNIX installations one must already have local super-user privileges to be able to perform this kind of attack, and this was the case in the experiment system when the attackers performed the intrusions.

System logging: *pacct* records the commands issued by every user, including root. If the super-user has run any of the more popular network listening tools, they should appear in the log entries of *pacct*, that is, unless the intruder has already turned off logging, which unfortunately is very likely.

Application program logging: N/A.

Resource utilization: Listening to all network traffic (by setting the network interface in "promiscuous mode") can load the local host heavily.

Lightweight logging: The same argument as for *pacct* above applies. However, since the arguments to, for instance *tcpdump*, are recorded, it should be easier to differ between legitimate and illegitimate uses of *tcpdump*.

Conclusion: The snooping that is described above is performed by special programs, most of which are not part of the system distribution. It is easy for the attacker to rename popular packages when installing them, thus foiling our logging effort. In the experiment, most attackers did not bother with this, and hence their intrusions were relatively easy to trace, since these are programs that normally should not be run.

5.6 Class C6: Trojan horses

Some attackers made unsuspecting users execute Trojan horses, that is, applications that purported to do something benign, but did something sinister in addition to that.

Trojan su It is possible for anyone to create a fake *su(1V)* program that when executed saves a copy of the entered password and then prints an error message "su: Sorry" as though the password were wrong. The program then erases itself from the filesystem.

System logging: Normally, when *su(1V)* is executed but the result is unsuccessful, a record starting with "*#su*" is found in *pacct*. The "*#*" indicates that the program was executed with root privileges.[7] Consequently, the Trojan horse *su*, which does not run with root privileges, should appear as a plain "*su*", which can easily be detected. We cannot judge whether this can be circumvented by a more carefully designed Trojan horse. With regard to tracing, the only thing *pacct* tells us is the user who executed the Trojan *su*, not the location or creator of the program.

Application program logging: N/A.

Resource utilization: limited.

Lightweight logging: Enough information to trace this intrusion is recorded since the intrusion method requires that the command is invoked as *su*. Information is recorded on which user executed the program and the full path of the program, and both real UID and *setuid* settings are logged. Since the full path of the program is logged, the chances of discovering who actually planted the fake *su* program increases.

Trojan e-mail attachment This is somewhat of a social engineering attack. One group of attackers sent an e-mail message that contained what was announced to be a picture of explicit nature. However, the picture could not be viewed by normal software already installed on the system, but only with the supplied software, which was also attached to the e-mail. That software was a Trojan horse that hijacked the viewer's account before it correctly displayed the picture.

Conclusion: System logging records only the last part of the path to the command. Because of this it is impossible to differentiate between the running of legitimate commands and their Trojan counterpart. The suggested logging policy detects these attacks, since it will show the execution of a normal system command with a suspect path, or the running of a command that has been introduced into the system in a suspicious way.

5.7 Class C7: Forged mail and news

Owing to the design of the mail and news servers on the experiment system, it was fairly easy to send a message that purported to be from someone else.

Faking email On many UNIX systems, it is possible to fake the sender of an e-mail message, making it appear to originate from another user or even from a non-existing user. This is done by connecting to the mail port through TCP/IP and interacting directly with the *sendmail(8)* daemon.

System logging: If the *telnet(1C)* command has been used, we can find a record of that in *pacct* on the sending machine and connect that to the records in *syslog(3)* (described below) through the time stamps.

[7] However, the documentation is unclear as to the exact circumstances under which the "#" is inserted by *pacct(2)*. We have found that it is a fairly unreliable indicator as to what privileges were actually acquired by the executing program.

Application program logging: If the message has been sent from one of our workstations, we can find tracks in */var/log/syslog* left by *sendmail(8)* via *syslogd(8)* on that machine, but only with the faked sender identity.

Resource utilization: limited.

Lightweight logging: Clearly, not enough information is recorded to trace the sending of a fake e-mail. Partial information may be available depending on the way in which way the *telnet(1C)* command is invoked or whether the command *mconnect(8)* is used.

Forged news As a consequence of the way in which the remote USENET News server protocol was designed, it is fairly easy to forge a USENET News article to make it appear as though it originated from another user on the system. All authentication must be performed in the news-client software, but nothing prevents a user from connecting to a remote News server by hand, that is, by using *telnet(1C)*.

Conclusion: All attackers that forged mail and news did indeed use the *telnet(1C)* command, passing the parameters on the command line, and were thus easy to detect by the suggested logging, even if the actual mail or news article would be difficult to trace. However, it is trivial to invoke the *telnet(1C)* command without any command line arguments and foil both methods of logging.

5.8 Class C8: Subverting *setuid* root applications into reading or writing system files

Setuid root applications on UNIX systems are allowed to read or write to any file by default. It is imperative that such applications check user supplied arguments carefully, lest they be tricked into doing something that the user would not normally be allowed to do. However, many such applications contain flaws that allow an attacker to perform such unauthorized reading or writing of critical files.

Xterm logfile bug, version 1 The *xterm(1)* client has an option for the entire session to be logged to a file. Furthermore, *xterm(1)* is *setuid* root for it to be able to change the owner of its device file to that of the current user. A bug makes it possible for any user to specify an existing file as the logfile to *xterm(1)* and have *xterm(1)* append data supplied by the attacker to that file.

System logging: From the *pacct* file, all we can see is that someone has run *xterm(1)*, a so common occurrence that we can safely say that it is impossible to trace an intrusion this way.

Application program logging: N/A

Resource utilization: Limited.

Lightweight logging: Since we log all the arguments to *xterm(1)* as it is being run, we catch both the invocation of the logfile mechanism and the telltale argument `-e echo "roott::0:1::/bin/sh"` or a similar one, which is the hallmark of this intrusion.

Xterm logfile bug, version 2 A variation of the preceding exploit, where the attacker can create a file and have the output from *xterm(1)* inserted in that file, provided that the file does not already exist.

Change files through mail alias The UNIX operating system maintains a global mail aliases database used by the *sendmail(8)* program to reroute electronic mail. One standard alias delivered with some versions of UNIX is *decode*. By allowing this alias, it is possible for anyone to modify some directories in the system.

Finger daemon The *finger(1)* utility in the experiment system is *setuid* to root, and it makes an insufficient access check when returning information about a user. It is possible to make finger display the contents of any file via the use of a strategic symbolic link from *.plan* in the user's home directory, to the target file.

Ex/vi-preserve changes file The venerable UNIX text editors *ex(1)* and *vi(1)* have a feature by which in the event that the computer crashes or the user is unexpectedly logged out, the system preserves the file the user was last editing. The utility that accomplishes this, *expreserve(8)*, is *setuid* to root, and has a weakness by which the attacker can replace and change the owner of any file on the system.

***/dev/audio* denial of service** Owing to a kernel bug, it is possible to crash the machine by sending a file via *rcp(1C)* to */dev/audio* on a remote machine.

Interactive *setuid* shell This problem in SunOS 4.1.x has been fixed in more recent operating systems. An attacker simply invokes a *setuid* (or *setgid* or both) shell script through a symbolic link, which immediately results in an interactive shell with the effective UID (and/or GID) of the owner of the shell script. This is detailed in Section 3.2.

Conclusion: The methods used to trick the *setuid* program inevitably either supply suspect arguments to the command or modify the filesystem in advance to running the *setuid* command. Depending on the specific circumstances, our logging proposal has a high chance of detecting and tracing the intrusion, since we either log the suspect arguments themselves or log the commands that poison the filesystem. System logging does not manage to accomplish either.

5.9 Class C9: Buffer overrun

As mentioned above, a *setuid* program must be careful in checking its arguments. There exists a class of security flaws where the attacker subverts the *setuid* application by filling an internal (argument) buffer so that it overflows into the *setuid* program's execution context. In this way it is possible for the attacker to force the *setuid* program to execute arbitrary instructions. We have encountered only one such attack. We include it here because of the severity of this type of attack and because it is widespread and commonly encountered in the field.

Buffer overrun in rdist The *rdist(1)* utility has a fixed length buffer that can be filled and thus made to overflow onto the stack of *rdist(1)*, which is *setuid* root. The attackers did not manage to exploit this other than to crash *rdist(1)*, but we include it here in any case, as it is an interesting class of intrusions.

System logging: As usual, *pacct* does not manage to leave any conclusive traces in the log files. The invocation of the program with the overflow condition would probably not show up, since the name of the finished program is recorded at the end of execution when the original program image has typically already been overlaid with that of a *setuid* shell.

Application program logging: N/A.

Resource utilization: Limited.

Lightweight logging: Since we log the arguments to the command, *rdist(1)* in this case, we immediately see that something is wrong. We in fact record in the log the entire program that *rdist(1)* is lured into overflowing onto its stack! (We may not always be so fortunate; some variations of these exploits keep the actual overflow code in an environment variable, which we will not log.) Since we see both the original *exec(2)* of *rdist(1)*, followed by the *exec(2)* of a root shell, without the intervening *fork(2V)*, we have conclusive proof that the system has been subverted.

Conclusion: See the discussion concerning the above exploit, since it is typical of these kinds of exploits.

5.10 Class C10: Execution of prepacked exploit scripts

It is possible for the novice attacker to download prepacked programs or command scripts that exploit a known security flaw. Such packages are in wide circulation and provide the attacker with an easy entry into the system.

Loadmodule *setuid* root script We have included a specific example of a breach that involves a *setuid* script since this *setuid* script is included in the SunOS distribution tapes. Hence, this security flaw is widespread and, as such exploit scripts have been developed, and widely circulated. This particular exploit script (*load.root*) was found by many experimenters who used it successfully, some without any deeper understanding of the security flaw involved. The exploit script in question uses the environment variable *IFS* to trick *loadmodule(8)* into producing an interactive shell with super-user privileges.

System logging: Since *pacct* does not record the arguments to the commands that the user executes, it is difficult to establish that the *load.root* exploit script has indeed been run. However, as the exploit script executes a number of commands in a predefined sequence, it should, at least in theory, be possible to ascertain that the script has been run with some level of certainty. This is generally not possible without detailed knowledge about the script. As mentioned previously, the “#” marker to indicate that super-user privileges has been used does not appear in this case.

Application program logging: N/A.

Resource utilization: Limited.

Lightweight logging: Since we can determine what commands were run and with what arguments, it becomes much easier to determine that a breach has occurred. In particular, the fact that the user has executed a script that in turn invokes *exec("/bin/sh", "sh", "-i")* with root privileges gives the game away. However, the flaw is exploited by the introduction of an environment variable, something which we do not record because this would lead to the storage of too much data. A strong indication that some sort of *IFS* manipulation has taken place, however, is the fact that the audit trail shows that a user has executed a command named *bin* as part of a shell script.

Mailrace (sendmail) The original mail handling client */bin/mail(1)* is *setuid* root in SunOS. If a recipient of a mail message lacks a mail box, the program creates one before it appends the mail message to it. Unfortunately, there exists a race condition between the creation of the mail box and the opening of it for writing. Exploit scripts have been published that exploit this race condition. These operate by replacing the newly created mail box with a symbolic link to any file which, as a result, will be overwritten or created with the contents specified by the attacker.

Conclusion: If the script is known beforehand, its use can be established with normal system logging. The suggested logging makes it a good deal more likely that a script that has not been previously examined can be traced and analysed to determine its effects.

6 Summary and discussion of results

After studying the above security breaches it becomes clear that the main system logging mechanism, *pacct*, suffers from three major shortcomings:

1) It does not commit the executed command to the audit trail until it has finished executing. This misses crucial long running commands. Commands, the process image of which is overlaid by a call to *exec(2)*, does not appear in the audit trail, and the command that crashes or compromises the system is at risk of not being included in the audit trail.

2) *Pacct* does not record the arguments to issued commands. This more often than not turns the log material into a useless alphabet soup from a security perspective, since it is impossible to see what executed commands were set to act upon in terms of files etc. In most of the cases above, the arguments to the commands are what set legitimate uses of the commands apart from illegitimate ones.

3) The mechanism that is supposed to trace uses of super-user privileges is unreliable at best and downright erroneous at worst. It can thus not be trusted to provide worthwhile information about the use of super-user privileges.

Table 4. Summary of logging mechanism evaluation.

Class (number of intrusions in class)	**Light-weight logging**	**Tradi-tional logging**
C1: Misuse of security enhancing packages (3)	X	X
C2: Search for files with misconfigured permissions or *setuid* programs (2)	X	
C3: Attacks during system initialization (4)		
C4: Exploiting inadvertent read/write permissions of system files (4)	X	
C5: Intercepting data (3)		
C6: Trojan horses (2)	X	X
C7: Forged mail and news (2)		
C8: Subverting *setuid* root applications into reading or writing system files (7)	X	
C9: Buffer overrun (1)	X	
C10: Execution of prepacked exploit scripts (2)	X	X
Number of classes:	7	3
Number of intrusions (30):	21	7

In Section 5 we have compared lightweight logging to three other logging methods: *system logging*, *application program logging*, and *monitoring resource utilization*. In order to make our comparison, we group all these logging methods under the heading of *traditional* logging, since we believe that they would most often be employed together. The results are summarised in Table 4, which contains a total of 30 different attacks in 10 classes. We see that lightweight logging detects 21 intrusions in 7 classes, whereas traditional logging only covers 7 intrusions in 3 classes. The "coverage"—loosely defined as related to the number of missed intrusions—is for lightweight logging, about a factor 2.5 better than for traditional logging, e.g., 9 missed intrusions compared to 23. This is despite the fact that traditional logging collects audit data from more sources. However, neither method succeeded in detecting any of the attacks in the three classes C3, C5, and C7 (altogether 9).

From this comparison it becomes clear that by correcting the three shortcomings in the original *pacct* mechanism, mentioned above, our proposed policy manages to trace an overwhelming portion of the intrusions, namely those that fall into the following three categories:

1) The user has run commands he should not run; the log shows that someone with a log-UID that does not correspond to a known supervisor has executed commands with super-user privileges.

2) The user has run commands with suspect arguments and, by doing so, has managed to trick a system application into doing something illicit.

3) The user has run a suspicious-looking sequence of commands. This indicates that the user has run some exploit script or some security enhancing package that he should not have run.

Furthermore, as a result of this, our logging policy produces an audit trail from which more detailed information can be extracted, namely exactly *how* an intrusion was performed and not only that is *was indeed* performed. Thus, by also logging more security-relevant information pertaining to who executed the command, and the general environment in which it was executed, we have a sufficiently complete record of events on which we could base further action, provided that the audit trail is protected from manipulation. This can be accomplished by logging to a dedicated network loghost, or to a write-only media, as detailed in [4, Chapter 10].

7 Conclusion

We have shown that the lightweight logging method is more effective in tracing intrusions than comparable methods and that it traces an overwhelming majority of intrusions encountered during our experiments. It can very easily be implemented using the SunOS BSM module in newer versions of the SunOS operating system [13]. Since it does not consume much resources in terms of processing power and storage capacity, it can be left running on all machines in an installation. Thus, it can be used as a “poor man’s logging”.

References

[1] James P Anderson. Computer security threat monitoring and surveillance. Technical report, James P Anderson Co., Box 42, Fort Washington, PA 19034, USA, April 15, 1980. In [2].

[2] Matt Bishop, editor. *History of Computer Security Project CD-ROM*. Number 1. Department of Computer Science, University of California at Davis, Davis, CA 95616-8562, USA, October 1998. Available from *http://seclab.cs.ucdavis.edu/projects/history*.

[3] Sarah Brocklehurst, Bev Littlewood, Tomas Olovsson, and Erland Jonsson. On measurement of operational security. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS ’94)*, pages 257–266, Gaithersburg, Maryland, June 27–July 1, 1994.

[4] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O’Reilly & Associates, second edition, 1996.

[5] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[6] Jay J Kahn and Marshall D Abrams. Contingency planning: What to do when bad things happen to good systems. In *Proceedings of the 18th National Information Systems Security Conference*, pages 470–479, Baltimore, Maryland, October 10–13, 1995. National Institute of Standards and Technology/National Computer Security Center.

[7] Teresa F Lunt. A survey of intrusion detection techniques. *Computers & Security*, 12(4):405–418, June 1993.

[8] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[9] National Computer Security Center, Fort George G. Meade, MD 20755-6000, USA. *A Guide to Understanding Audit in Trusted Systems*, June 1, 1988. NCSC-TG-001, Version-2.

[10] Tomas Olovsson, Erland Jonsson, Sarah Brocklehurst, and Bev Littlewood. Towards operational measures of computer security: Experimentation and modelling. In Brian Randell et al., editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, chapter VIII. Springer, Berlin, 1995.

[11] W Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[12] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA. *System and Network Administration*, March 27, 1990. Part No: 800-4764-10, Revision A.

[13] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA. *SunSHIELD Basic Security Module Guide*, November 1995.

[14] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[15] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the 3rd USENIX UNIX Security Symposium*, pages 85–92, Baltimore, Maryland, September 14–16, 1992. USENIX Association.

Paper G

Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST)

In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 146–161, Oakland, California, May 9–12, 1999.

This page is intentionally left blank.

Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST)*

Ulf Lindqvist
Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
ulfl@ce.chalmers.se

Phillip A. Porras
Computer Science Laboratory
SRI International
Menlo Park, California
porras@csl.sri.com

Abstract

This paper describes an expert system development toolset called the Production-Based Expert System Toolset (P-BEST) and how it is employed in the development of a modern generic signature-analysis engine for computer and network misuse detection. For more than a decade, earlier versions of P-BEST have been used in intrusion detection research and in the development of some of the most well-known intrusion detection systems, but this is the first time the principles and language of P-BEST are described to a wide audience. We present rule sets for detecting subversion methods against which there are few defenses—specifically, SYN flooding and buffer overruns—and provide performance measurements. Together, these examples and measurements indicate that P-BEST-based expert systems are well suited for real-time misuse detection in contemporary computing environments. In addition, the simplicity of the P-BEST language and its close integration with the C programming language makes it easy to use while it is still very powerful and flexible.

1 Introduction

Intrusion detection components analyze system and user operations in computer and network systems in search of activity considered undesirable from a security perspective. Data sources for intrusion detection may include audit trails produced by an operating system, or network traffic flowing between systems, or application logs, or data collected from system probes (e.g., file system alteration monitors).

*The work presented in this paper is currently funded by the Information Technology Office, Defense Advanced Research Projects Agency (DARPA/ITO), under contract number F30602-96-C-0294. U. Lindqvist participated in this work while visiting SRI International as an international fellow and his work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), under project number P10435.

The collected data may be stored for batch-mode analysis or immediately analyzed in real-time.

For the most part, the various strategies for intrusion detection are not unique to the field, but are rather derived from applications established by other fields: knowledge-based expert systems, pattern recognition algorithms, statistical profiling techniques, neural networks, Bayesian statistics, information retrieval algorithms, state-transition models, Petri-net techniques, and so forth. Among the more widely used strategies proposed early within the intrusion detection community are signature-based analyses.

Intuitively, we describe a signature-based intrusion-detection component as an algorithm with which we specify the characteristics of malicious behavior and then monitor an event stream for activity that maps to the target behavior. Various signature-based systems have been developed, ranging from simple (but efficient) pattern-matching systems to more sophisticated algorithms that employ more general directed reasoning systems such as rule-based expert systems. In this paper, we describe in detail the principles and language of one forward-chaining rule-based expert system construction toolset called P-BEST (Production-Based Expert System Toolset), which has been continually applied to intrusion detection applications for more than a decade, but never before widely presented in this level of detail.

By using a general expert system, we can describe the behavior of our signature-based intrusion-detection component within an established theoretical framework. This choice also facilitates the evolution of the component, because new rules can be added without changing existing rules and without creating any undesired dependency. Traditional reasons for not choosing an expert system are related to low performance, difficult integration with other program components, and language complexity. However, in this paper we show that P-BEST is sufficiently fast for real-time detection of currently widely used attack methods—SYN flooding and buffer overruns—against which systems usually have no defense mechanisms. We also show that P-BEST provides exceptional interoperability with native operating system libraries, and is easily integrated into a larger software framework for distributed anomaly and misuse detection. We also argue that while the production rule language is powerful, it remains easy to use for beginners.

2 Monitoring misuse through expert systems

Expert systems provide strategies and mechanisms for processing facts regarding the state of a given environment, and deriving logical inferences from these facts. With respect to intrusion detection, a fact maps to an event that is recorded and evaluated by the expert system. This process of fact evaluation leading to the assertion of a new derived fact or conclusion is referred to as *modus ponens,* which states that given $(p \Rightarrow q)$ and p we deduce q. Systems that iteratively apply modus ponens under a bottom-up reasoning strategy (from evidence evaluation to conclusion) are referred to as *forward-chaining* systems. Forward-chaining expert systems are well-suited for reasoning about activity within an event stream. A forward-chaining rule-based system is data-driven: each fact asserted may satisfy the conditions under which new facts or conclusions are derived. Alternatively, *backward-chaining* systems employ the reverse strategy; starting from a proposed hypothesis they proceed to collect supportive evidence. Backward-chaining systems are typically applied to problems of diagnosis, whereas forward-chaining strategies dominate systems involving prognosis, monitoring, and control applications.

Using a forward-chaining rule-based system, one may establish a chain of rules, or *rule set*, with which a series of asserted facts may lead the system to deduce that a targeted multistep scenario has occurred. Within an intrusion detection system, event records are asserted as facts and evaluated against penetration rule sets. As individual rules are evaluated against facts and satisfied, the individual event records provide a trail of reasoning that allows the user to analyze the evidence of malicious activity in isolation from the full event stream. In this section, we will discuss the basic elements of forward-chaining rule-based systems, and provide an overview of the P-BEST expert system and its language.

2.1 Components of forward-chaining systems

The underlying strategy of a forward-chaining reasoning system involves the atomic evaluation of each fact presented to the system against conditional expressions that, when satisfied by the arguments of a fact, establish new derived facts or conclusions. In this context, a *fact* is a statement that is asserted into the system and whose validity is accepted (for example, "smoke is present"). Facts are often implemented as attributes and values that represent the state of the environment to which the expert system is applied. A *rule* is an inference formula of the form $\phi_1,\ldots,\phi_n$ *infer* ψ. Inference formulae can be alternatively expressed as *production rules*, such as `IF ... THEN ...`. Production rules are the basic elements through which an expert system is programmed to interpret and discover meaning from environmental signals that it receives, as in

`IF` *smoke is present* `THEN` *fire is near*.

A production rule consists of two parts, the *antecedent* (or conditional part, left-hand side) and the *consequent* (or right-hand side) as shown in Figure 1. When the *conditions* (predicate expressions) in the antecedent are satisfied, the rule is *activated*. The logical component through which an expert system evaluates a fact against the production rules is referred to as the *inference engine*. As an antecedent is found to be satisfied by the attributes of a fact, the consequent of the rule is asserted to hold, and the rule is said to have *fired*. Expert systems might additionally allow the inference engine to initiate action within the consequent, for example:

`IF` *fire is near* `THEN` **initiate** *sprinkler*.

Abstractly, the assertion of action, such as the initiation of a response, based on a fact derived from an inference engine is placed within the purview of a *decision engine*, though in practice inference and response may be merged.

The collection of facts available to the system at any point in time is called the *factbase* (or working memory) of the system. The collection of rules is called the *knowledge base* (or production memory). Although separation of data (facts)

$$\boxed{\begin{array}{l} \texttt{IF} \\ \quad\left.\begin{array}{l} condition_1 \\ condition_2 \\ \quad\vdots \end{array}\right\} \text{antecedent} \\ \texttt{THEN} \\ \quad\left.\begin{array}{l} derived\_fact_1 \\ derived\_fact_2 \\ \quad\vdots \end{array}\right\} \text{consequent} \end{array}}$$

Figure 1. Production rule structure.

from knowledge (rules) is an important abstraction within rule-based expert systems, some texts use the terms more loosely and consider the factbase to be part of the knowledge base. Another important abstraction is the separation of knowledge from the inference engine. In practice, an inference engine, also known as an expert system *shell*, provides several advantages over a one-of-a-kind system written in a procedural language. In particular, a knowledge-independent shell can be used to develop expert systems for many different knowledge domains. The knowledge in the expert system can also be incrementally extended by adding new rules, as opposed to implementing large portions of the decision process all at once. Next, we present the principles and language of P-BEST, a construction toolset for building customized inference engines, and discuss its applicability to intrusion detection.

2.2 An overview of P-BEST

The Production-Based Expert System Toolset (P-BEST) was originally written by Alan Whitehurst, and employed in the Multics Intrusion Detection and Alerting System (MIDAS) [18], which performed misuse detection on the National Computer Security Center's Internet-connected mainframe, Dockmaster. P-BEST was later enhanced at SRI by Whitehurst, and later by Fred Gilham, and was employed in an early version of the Intrusion Detection Expert Systems (IDES) [14], and later Next-Generation IDES (NIDES) [1]. See Section 3 for details on the application of P-BEST on these systems.

The P-BEST toolset consists of a rule translator, a library of runtime routines, and a set of garbage collection routines. When using P-BEST, rules and facts are written in the P-BEST production rule specification language. The rule translator, *pbcc*, is then used to translate the specification into a C language expert system program. This expert system can then be compiled into either of two forms: a stand-alone self-contained executable program or a set of library routines that implement the core P-BEST inference engine, and which can be linked to a larger software framework. P-BEST has several features that make it well-suited for the type of application described in this paper:

- The P-BEST language is small and relatively intuitive to use and extend.
- It is easily applied to a variety of problem domains. P-BEST provides a general-purpose forward-chaining inference engine that can be targeted to

a specific application domain. P-BEST does not inherently depend on the structure of the input data stream or the inference objectives of the application that employs it.

- By using translation instead of interpretation of rules, P-BEST can be used to build expert systems for performance-demanding applications. A pre-compiled expert system, rather than an expert system interpreter, provides a significant advantage in performing real-time event analysis.

- Pre-compilation also allows P-BEST components to be integrated well into larger program frameworks, and is easily called from, and can call out to, other C libraries. Arbitrary C functions can be called from the antecedent or consequent of any P-BEST rule. Thus, it is possible to write powerful rules without adding unnecessary complexity to the P-BEST language.

2.3 The P-BEST language

P-BEST provides a production rule language from which users may specify the inference formula for reasoning and acting upon facts asserted into its factbase from external sources or derived from the satisfaction of other production rules. This section provides a brief overview of the principle elements of this language, with common examples of its usage. The language overview provides the reader with a primer for understanding several examples of intrusion detection rules later in this paper.

In P-BEST, the structure of a fact is specified by the user through a template definition referred to as a pattern type or *ptype*. For example, to define a ptype named *event* that consists of the four fields *event_type* (an integer), *return_code* (an integer), *username* (a string), and *hostname* (a string), we define the fact template as in Figure 2. Facts from such a ptype definition could be constructed through the monitoring of audit records and asserted into the factbase for evaluation against the available production rules.

```
ptype[event event_type:int,
            return_code:int,
            username:string,
            hostname:string]
```

Figure 2. An example of a ptype declaration.

Fact evaluation is performed by the P-BEST inference engine, where the attributes of the fact are mapped against the predicate expression(s) of each rule antecedent. For example, we may want to determine whether the asserted fact represents an unsuccessful login attempt, which we shall refer to as *e*. To express this criterion using a mathematical notation style, we can form the statement in Equation 1.

$$\Big(\exists e\Big)\Big((e \in S) \wedge \mathrm{event}(e) \wedge (e_{event\_type} = login) \wedge (e_{return\_code} = bad\_password)\Big) \quad (1)$$

Here, *S* represents the set of all facts known to the P-BEST factbase, and within which a production rule antecedent postulates the existence of a fact *e* that satisfies specific properties. In the P-BEST language, the statement in Equation 1 placed in the antecedent of a rule would be written as in Figure 3.

```
[+e:event|event_type == login,
          return_code == BAD_PASSWORD]
```

Figure 3. An example of fact matching.

The term `e:event` allows one to assign an *alias* `e` to one fact (of possibly several) that satisfies the antecedent for the duration of the rule. The plus (`+`) sign after the opening bracket represents an existential quantifier that allows the rule to check for any fact that satisfies the conditions of the antecedent. Alternatively, a minus (`-`) sign searches for cases where no fact in the factbase satisfies the conditions of the antecedent. For example,

```
[-event|username == "GoodGuy"]
```

evaluates to true if there is no event in the factbase that has been asserted on behalf of “GoodGuy.”

The plus and minus tests have corresponding assert and delete actions that can appear in the consequent of a rule. For example, to assert a new fact of ptype `bad_login` and give its fields initial values, we can write

```
[+bad_login|username = e.username, hostname = e.hostname]
```

To be deleted from the factbase, a fact must be matched and given an alias in the antecedent before it can be deleted in the consequent. This is illustrated in the example of a complete rule named `Bad_Login` in Figure 4.

```
rule[Bad_Login(#10;*):
   [+e:event| event_type == login,
              return_code == BAD_PASSWORD]
==>
   [+bad_login| username = e.username,
                hostname = e.hostname]
   [-|e]
   [!|printf("Bad login for user %s from \
      host %s\n", e.username, e.hostname)]
 ]
```

Figure 4. An example of a rule declaration.

The `Bad_Login` rule in Figure 4 also demonstrates how the evaluation of an asserted fact can be used to derive subsequent facts that may themselves drive new inferences. That is, in the above rule, should a login event be encountered with a return code of `BAD_PASSWORD`, the rule creates a new fact of ptype `bad_login`, which saves the username and hostname of the event; the rule also destroys the event fact `e` from the factbase. Using a mathematical notation, we can represent this state transition in our factbase from S to a desired new state S' as in Equation 2 (this excludes lines 8 and 9 in Figure 4).

$$\frac{\Big(\exists e\Big)\Big((e \in S) \wedge \text{event}(e) \wedge (e_{event\_type} = login) \wedge (e_{return\_code} = bad\_password)\Big)}{\vdash \Big(S' = S - \{e\} \bigcup \{\text{bad\_login}(b) \mid (b_{username} = e_{username}) \wedge (b_{hostname} = e_{hostname})\}\Big)} \tag{2}$$

Within parentheses after the rule name (line 1), there is a semicolon-separated list of options. The option `#10` means that this rule is given a ranking (priority) of 10. Priorities allow one to specify well-defined orders in the sequences for rule evaluation, and are primarily used for rules required to be evaluated first for initialization purposes, or that must be evaluated last to perform garbage collection. The star option (`*`) indicates that the rule is repeatable, that is, the rule is allowed to fire repeatedly even if no other rule is fired in between. Thus, a key function of the consequent is to alter the state of the factbase such that the antecedent is not satisfied indefinitely (e.g., the consequent may mark or remove a fact). The arrow delimiter (`==>`) separates the antecedent and the consequent (line 4).

The `[!|...]` clause (line 8) within the consequent illustrates how the P-BEST inference engine may call out to native C functions should action be warranted when the antecedent is evaluated to true. Both inference and action can be taken directly within the P-BEST inference engine. P-BEST recognizes most of the standard library C functions, which may be invoked directly via the `[!|...]` clause, and which may refer to ptype attributes directly. User-defined C functions and auxiliary variables may also be invoked and referenced, respectively. To do this, we must declare our intentions to reference C variables and functions using the P-BEST external type declaration mechanism *xtype*. For example, the following external declarations will allow P-BEST to recognize a user-defined C function called *native_probe()* returning an integer and an integer variable *end_of_stream* as follows:

```
xtype [native_probe: intfunc]
xtype [end_of_stream: int]
```

We can then employ our native C routine and variable directly in a P-BEST production rule, as illustrated in Figure 5. The antecedent `[?|...]` clause (line 3) is a query clause used to evaluate conditional requirements. This rule will check to see whether the `end_of_stream` variable has been set to `1`, and if not, it will set the variable to the return code of the function `native_probe()` (line 5), which is invoked in the consequent. This *native_probe()* could, for example, provide an interface to the host operating system that allows the expert system to retrieve application records, which it may then assert as facts in the factbase. The rule also gives an example (line 6) of how a field in an existing fact can be modified; in this case, the field `rec_cnt` of the fact `counter`, aliased in the antecedent, is incremented by 1.

To further improve the performance of the expert system, rules can be disabled and enabled dynamically through actions in the consequents of rules. A rule can even disable itself, which means that it can fire once, at most, unless enabled again by another rule. To disable a rule, we can put the following action in a consequent:

```
[-#rulename]
```

```
rule[get_native_record(-99;*):
   [+c:counter]
   [?|'end_of_stream != 1]
==>
   [!|'end_of_stream = native_probe()]
   [/c|rec_cnt += 1]
]
```

Figure 5. Example usage of external C types.

To enable a rule, we can change the minus sign in the above statement to a plus sign. In addition, a rule can be declared as disabled from start by adding a single minus sign to the list of options after the rule name, for example:

```
rule[rulename(#10;*;-):
```

Using these features, we can build preconditional requirements that can enable or disable whole portions of the knowledge base, depending on the current state of the environment being monitored. For example, rules pertaining to the analysis of a service *A* can be dynamically added or removed from the knowledge base by the expert system itself, depending on whether service *A* is currently enabled or disabled within the analysis target. Another example is when the analysis is extended with previously disabled rules due to an increased level of suspicion reported by the basic rule sets.

Another powerful feature of P-BEST is the ability of rules to uniquely mark and unmark facts, and to test for these marks. This can be used when we want to give several groups of mutually exclusive rules the chance to examine a fact before it is deleted from the factbase. Each rule will evaluate the fact, and if the antecedent is satisfied, the consequent of the rule will mark the fact. This will allow the rule to avoid re-firing, while not having to remove the fact completely from the factbase. When all such rules have evaluated (and if necessary marked) the fact, the fact can then be removed by a lower-priority fact-removal rule that is run last. For example, to match an event that is not marked with `CHECKED`, we can put the following test in the antecedent of our rule:

```
[+e:event^CHECKED]
```

To mark a matched event fact `e` with `CHECKED`, we can add the following action to the consequent:

```
[$|e:CHECKED]
```

Alternatively, to unmark a fact we simply use a caret (^) instead of the dollar sign ($):

```
[^|e:CHECKED]
```

Finally, we can use the dollar sign to check for a marked fact, as follows:

```
[+e:event$CHECKED]
```

2.4 P-BEST language simplicity and usability tested in student experiment

Although the P-BEST language has proven itself suitable for intrusion detection systems, it is in fact also a general language for building rule-based expert systems in many different applications. The close integration with C makes it unnecessary to include more than the basic operations in the P-BEST language itself, because any needed operation can be designed as a C function and called from the antecedent or consequent of a P-BEST rule. Thus, the P-BEST language can be kept small and simple, resulting in a very low learning threshold for beginners.

In addition to its use in intrusion detection system development, P-BEST has recently for the first time been used for laboratory exercises in a university course in applied computer security at Chalmers. In addition to the educational goals of these exercises, we wanted to learn what amount of instruction is required for beginners when applying P-BEST to intrusion detection analysis and thereby see whether the experiment would support or contradict our hypothesis that the P-BEST is easy to use for beginners.

The assignment was to build a system that could be used to automatically detect attacks against a file transfer (FTP) server. For evaluation of their resulting system, the students were given a very large data file (3 megabytes of text) containing recorded network data representing actual FTP transactions. A small number of real and synthetic intrusions were mixed with a large number of normal transactions, and the students were to use their system to find those intrusions. It was supposed to be a pedagogic effect that the file was too large to be easily examined by hand, because this is the very reason for having automatic intrusion detection tools. It was also required by the students to include in their lab reports a discussion of their experiences of using the tool.

There were 87 students who participated in the assignment, and with a few exceptions they worked in pairs, making a total of 46 groups. The estimated maximum working time was two lab sessions of four hours each, plus another eight hours of homework to prepare the lab sessions and to complete the report. Out of the 46 groups, 25 had built a system that gave the completely correct answer. An additional 8 groups would most likely have got the correct result if they had not all misinterpreted a vaguely formulated part of the instructions. Only a handful of groups failed to hand in a report before the given deadline. Most students reported that they found the exercise interesting and some even took the time to give detailed suggestions of improvements to the tool. As we had expected, being used to writing programs in a procedural style, they had some initial difficulties in declarative programming. In summary, we claim that the student experiment shows that P-BEST has a low learning threshold for beginners and is thereby suitable both for building user-customizable intrusion detection systems as well as for student exercises in computer security courses.

3 Integration of P-BEST into IDS components

For more than 10 years, P-BEST has been successfully integrated into several intrusion detection systems (IDSs) that represent the state of the art for their time. The application of P-BEST to intrusion detection began in the mainframe world of Multics and lands in present time with the highly distributed, scalable, and network-oriented EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) environment. It is not only the IDSs that have changed over time; P-BEST itself has been continuously improved as the requirements and its operational environment have changed. However, performance and language simplicity are issues that have had top priority from the beginning, and are no less important today.

3.1 P-BEST in MIDAS

P-BEST was developed at SRI International and first deployed as the core of MIDAS, which provided real-time intrusion and misuse detection for the National Computer Security Center's networked mainframe, Dockmaster, a Honeywell DPS-8/70 running Multics [18]. Audit data preprocessing and command monitoring was performed on the Dockmaster, and the data was sent to a separate Symbolics Lisp machine where the expert system and the user interface were running.

MIDAS used both static and dynamic knowledge for detecting intrusive user behavior. The static knowledge was represented in so-called immediate attack heuristics written as P-BEST rules that would trigger on events that were considered anomalous regardless of previous system activity. In terms of dynamic knowledge, MIDAS recorded user and system statistics in a database that would represent normal behavior. It is interesting to note that it was in fact another set of P-BEST rules—the user anomaly heuristics and the system state heuristics—that used threshold values derived from the statistics database to distinguish anomalous user and system behavior from normal activity. Thus, the P-BEST inference engine was the sole analysis component in MIDAS.

3.2 P-BEST in IDES and NIDES

In 1983, SRI International began research on statistical techniques for audit-trail reduction and analysis [6]. This research led to the development of a prototype IDES, capable of providing real-time detection of security violations on single-target host systems. Originally, IDES only used statistical anomaly detection [5, 12], but later a component for misuse detection based on static knowledge was added, using P-BEST [14]. The two components were fed the same audit records, but performed their inferences and reporting independently.

Next, SRI began a comprehensive effort to enhance, optimize, and re-engineer the earlier IDES prototype into a production-quality intrusion-detection system with the name Next-Generation Intrusion Detection Expert System (NIDES). Just like its predecessor, NIDES has both a statistical anomaly detection component and a rule-based misuse detection component [1]. Again, P-BEST was the expert system shell of choice for the rule-based component, but P-BEST was first extensively revised. Among other things, the revision gave P-BEST a new syntax and a very tight coupling to the C programming language. While the early version of P-BEST used in MIDAS and IDES compiled rules into Lisp object code, the new version produced C source code. NIDES collects host audit trail data from different host systems and converts it to the NIDES audit record format. The current version of NIDES has a default rulebase of 39 rule sets (69 total production rules) but also allows the user to write his or her own rules (that, for example, are specific to the user's environment or policy) and has a mechanism for dynamically adding new rules at runtime.

3.3 P-BEST in the EMERALD eXpert

The EMERALD environment is a distributed scalable tool suite for tracking malicious activity through and across large networks [16]. EMERALD employs a building-block architectural strategy using independent distributed surveillance *monitors* that can analyze and respond to malicious activity on local targets, and can interoperate to form an analysis hierarchy. The generic EMERALD monitor architecture is designed to enable the flexible introduction and deletion of analysis engines from the monitor boundary as necessary. In its dual-analysis configuration, an EMERALD monitor instantiation combines signature analysis with statistical profiling to provide complementary forms of analysis over the operation of network services and infrastructure. In general, a monitor may include additional analysis engines that can implement other forms of event analysis, or a monitor may consist of only a single resolver implementing a response policy based on intrusion summaries produced by other EMERALD monitors. Monitors also incorporate a versatile application programmers' interface (API) that enhances their ability to interoperate with the analysis target, and with other third-party intrusion detection tools.

Underlying the deployment of an EMERALD monitor is the selection of a target-specific event stream. The event stream may be derived from a variety of sources, including audit data, network datagrams, SNMP traffic, application logs, and analysis results from other intrusion detection instrumentation. The event stream is parsed, filtered, and formatted by the target-specific event-collection methods provided by the monitor's pluggable configuration library, referred to as the *resource object*. Event records are then forwarded to the monitor's analysis engine(s) for processing.

The EMERALD *eXpert* (pronounced E-expert) is a generic signature-analysis engine based on the expert system shell P-BEST. The eXpert resource object has two parts, one of which consists of the configuration files for the EMERALD API that define the transports used for message passing (e.g., files or network connections), the message templates, and so forth, for the particular analysis target. The other part of the resource object is a P-BEST source file containing the fact type (ptype) declarations and rules. In the ptype declarations, the user must specify to what message field (if any) the ptype field corresponds.

Under EMERALD's eXpert architecture, special-purpose rule sets are encapsulated within resource objects that are then instantiated with an EMERALD monitor, and which can then be distributed to an appropriate observation point in the computing environment. This enables a spectrum of configurations from light weight distributed eXpert signature engines to heavy-duty centralized host-layer eXpert engines, such as those constructed for use in NIDES and MIDAS. In a given environment, P-BEST-based monitors may be independently distributed to analyze the activity of multiple network services (e.g., FTP, SMTP, HTTP) or network elements (e.g., a router or firewall). As each EMERALD eXpert is deployed to its target, it is instantiated with an appropriate resource object (e.g., an FTP resource object for FTP monitoring), while the eXpert code base remains independent of the analysis target.

EMERALD also introduces a target-independent code generation utility that allows one to automatically produce the library interfaces necessary to integrate a P-BEST expert system into the EMERALD monitor infrastructure. This utility effectively relieves the creator of a resource object from dealing with the internal operation of the eXpert code-base, even when redirecting the eXpert to a completely new event stream. This automated generation utility both enhances the rapid integration of eXpert to new analysis targets, and simplifies the process of augmenting the rule base with new heuristics. The basic operation of an eXpert analysis engine is as follows:

1. On startup, eXpert is initialized and its interface routine waits for messages on one or several transports, as specified in the configuration files of the resource object.

2. When an event record is received in the form of an EMERALD message, the message is matched against an interface data structure associated with the ptype definition in the eXpert's P-BEST fact base.

3. The message content is transferred to the interface data structure, which in turn is used to assert a fact into the expert system factbase.

4. The eXpert interface component hands over control to the expert system inference engine.

5. If a rule is fired, in which the consequent specifies that an alert shall be generated, the alert is propagated back to the analysis engine's interface component, which in turn composes and sends the alert on to the EMERALD resolver. The resolver operates as the monitor's decision engine, and can invoke local responses based on the alert or propagate the alert on to subscribers of the monitor's results (including administrative display interfaces).

6. When there are no more rules that can fire, the expert system returns control to the interface routine that again starts waiting for incoming messages.

In the following section, we discuss examples of how eXpert can be used to analyze very different types of event streams.

4 eXpert rule development examples

Throughout its usage, P-BEST inference engines have implemented a variety of intrusion detection rule sets for detecting and responding to numerous forms of malicious activity. Next, we describe the application of P-BEST in reasoning about attacks represented in two data streams: Solaris 2.5.+ audit trails, and TCP/IP packet streams. The examples illustrate the declarative style of the language, and how event streams can be represented and analyzed.

4.1 Examples of BSM audit trail analysis

The first example of an event stream to be analyzed is the audit trail produced by the Solaris Basic Security Module (BSM) from Sun Microsystems [19]. The audit records are normally saved in a file, but we have developed a BSM collection unit that receives audit records from the OS kernel in real time, formats and sends each record as an EMERALD message to the target monitor for analysis.

For all the rules that analyze BSM data, there is a ptype called `bsm_event` into which the relevant fields from incoming messages are mapped. There is also a rule that has highest priority and copies the time of every incoming `bsm_event` fact into a new `time` fact, and finally a rule with lowest priority that removes the `bsm_event` fact after all the other rules have had a chance to look at it. For the sake of brevity, these ptype definitions and administrative rules are omitted from the examples.

4.1.1 Failed authentication attempts As an example of the declarative programming paradigm that P-BEST supports, we present a set of rules that are designed to detect a number of failed authentication attempts within a certain time window. The example illustrates how facts are created in rule consequents to keep state information between incoming events, and how the rule designer can make sure that facts are removed from the factbase when they are no longer needed.

Let us assume that we want to raise an alert if `x` user authentication failures occur within `y` seconds for a monitored target. A user authentication failure is defined as the case when either an invalid username or an invalid password is given to one of the programs *login*, *telnet*, *rlogin*, *rshd*, or *su*. To accomplish this, we may employ the rule set presented in Table 1, which is described as follows:

• `A1, A2`: For every incoming event that is a user authentication failure, save the event information in a `bad_login` fact and increment the counter for current bad logins (`current_bl_cntr`) by 1. The reason for having two rules is to separate the case where the username is invalid (`A1`) from the case where the username is valid but the password is invalid (`A2`). In the latter case, we want to include the username in the information we save and therefore need a rule consequent that is different from the former case where there is no username reported in the audit record.

• `A3`: When the `current_bl_cntr` counter has the value `x`, send an alert and create a `max_bl_reached` fact to indicate that the authentication failure threshold was reached.

• `A4`: If there exists a `max_bl_reached` fact, then loop through all saved `bad_login` facts. For every `bad_login` fact, print the information contained in the fact to a log file and delete the fact from the factbase.

• `A5`: If there exists a `max_bl_reached` fact but no `bad_login` facts (i.e., they were all printed and deleted by rule `A4`), then delete the `max_bl_reached` fact from the factbase.

• `A6`: If there exists a `bad_login` fact, but no `max_bl_reached` fact, and the difference between the `bad_login` timestamp and the current event timestamp is more than `y` seconds, then delete the `bad_login` fact from the factbase and decrement the `current_bl_cntr` by 1.

4.1.2 Buffer overrun attacks

Buffer overrun attacks are a common way for attackers to gain super-user privileges after first breaking into an unprivileged user account. Typically, a privileged (*setuid* to *root*) program is called with an extremely long and carefully crafted argument that overflows memory buffers and alters the program execution [3]. In principle, it would require a fair amount of programming skills and patience to exploit a buffer overrun vulnerability, but ready-to-use exploit programs that can be downloaded from Internet sites give immediate super-user access when executed. Here, we present an example of a simple heuristic P-BEST rule that detects the behavior of most of the exploit programs. For example, it has been tested against buffer overrun exploits that are based on subverting Solaris 2.5 *eject, fdformat, ffbconfig* and *ufsrestore*.[1]

The heuristic rule is based on the following observations of the audit trail characteristics of common buffer overrun exploits:

- We can detect the attack by analyzing a single *exec* system call audit record, as suggested in [2].
- To determine that the *exec* call concerns a *setuid* program (otherwise, it would not be a target for attack), we simply match only the audit records for which the *effective user id* and *real user id* fields are different.
- The argument passed to the *exec* call is relatively long (because it must overflow a buffer and contain executable code), making the length of the entire audit record significantly exceed the length of almost all normal *setuid exec* calls.
- By necessity of the applicable hardware (Sun and Intel), the *exec* argument contains binary opcodes in the range of ascii control characters. While such a property may not necessarily hold on all possible hardware platforms, this heuristic works exceptionally well for our purposes.

The P-BEST rule that uses the observations above to detect buffer overrun attacks is shown in Figure 6. This simple heuristic rule is not a fool-proof way to detect all possible buffer overrun attacks, but it is remarkably efficient in terms of coverage

[1]There are numerous additional buffer overrun attacks that employ the identical attack strategy as the four attacks discussed here. All should be subject to detection by this rule.

Table 1. Rule set for detection of failed authentication attempts.

```
rule[A1(*):
   [+e:bsm_eventˆA12]
   [?|e.header_event_type   == ’AUE_login  ||
      e.header_event_type   == ’AUE_telnet ||
      e.header_event_type   == ’AUE_rlogin ||
      e.header_event_type   == ’AUE_rshd   ||
      e.header_event_type   == ’AUE_su]
   [?|e.return_return_value == ’INVALID_USER]
   [+cc: current_bl_cntr]
   [-max_bl_reached]
==>
   [+bad_login |
      timestamp    = e.header_time,
      audit_seq_no = e.msequenceNumber,
      username     = "invalid username",
      command      = e.header_command,
      etype        = e.header_event_type,
      hostname     = e.subject_hostname,
      portID       = e.subject_port_id,
      processID    = e.subject_pid,
      textList     = e.textList]
   [/cc| value += 1]
   [$|e:A12]
]
```

```
rule[A2(*):
   [+e:bsm_eventˆA12]
   [?|e.header_event_type   == ’AUE_login  ||
      e.header_event_type   == ’AUE_telnet ||
      e.header_event_type   == ’AUE_rlogin ||
      e.header_event_type   == ’AUE_rshd   ||
      e.header_event_type   == ’AUE_su]
   [?|e.return_return_value == ’INVALID_PWD]
   [+cc: current_bl_cntr]
   [-max_bl_reached]
==>
   [+bad_login |
      timestamp    = e.header_time,
      audit_seq_no = e.msequenceNumber,
      username     = e.subject_runame,
      command      = e.header_command,
      etype        = e.header_event_type,
      hostname     = e.subject_hostname,
      portID       = e.subject_port_id,
      processID    = e.subject_pid,
      textList     = e.textList]
   [/cc| value += 1]
   [$|e:A12]
]
```

continues on next page

continued from previous page

```
rule[A3(*):
   [-max_bl_reached]
   [+cc:current_bl_cntr | value == 'x]
   [+ts:time^A3]
==>
   [!|printf("ALERT: Max Bad Logins \n")]
   [+max_bl_reached | value = 1]
   [$|ts:A3]
   [!|EXpertReport('eXpertMessagePointerString,
      1042, "description", 'pTypeString,
      "MAX LOGIN ALERT",
      "ruleName", 'pTypeString, "A3", "")]
]
```

```
rule[A4(*):
   [+max_bl_reached]
   [+bc:bad_login]
   [+cc:current_bl_cntr]
==>
   [!|printf("(%s): %s from %s on %s port %d, \
     PID = %d, time = %d, seq no = %d \n",
     bc.textlist, bc.command, bc.username,
     bc.hostname, bc.portID, bc.processID,
     bc.timestamp, bc.audit_seq_no)]
   [/cc|value -= 1]
   [-|bc]
]
```

```
rule[A5(*):
   [+mx:max_bl_reached]
   [-bad_login]
==>
   [-|mx]
]
```

```
rule[A6(*):
   [+ts:time^A6]
   [-max_bl_reached]
   [+bc:bad_login]
   [+cc:current_bl_cntr]
   [?|(ts.sec - bc.timestamp) > 'y]
==>
   [/cc|value -=1 ]
   [-|bc]
   [$|ts:A6]
]
```

and correctness; it detects most common attacks and has not produced any false positives when tested on a collection of over 35 million audit records in which the location of buffer overflow attacks was known *a priori*.

```
rule[BSM_LONG_SUID_EXEC(*):
   [+e:bsm_event]
   [?|e.header_event_type == 'AUE_EXEC ||
      e.header_event_type == 'AUE_EXECVE]
   [?|e.subject_euid != e.subject_ruid ]
   [?|contains (e.exec_args, "^\\") == 1]
   [?|e.header_size > 'NORMAL_LENGTH]
==>
   [!|printf("ALERT: Buffer overrun attack \
      on command %s\n", e.header_command)]
]
```

Figure 6. A heuristic rule for detecting common buffer overrun attacks.

To determine a suitable value for the `NORMAL_LENGTH` threshold parameter, we have analyzed in the order of 4 million audit records representing normal system usage (of which over 29 thousand were *exec* events) in addition to audit records representing common buffer overrun attacks. This analysis gave the following results:

- All the attacks we tested produce an *exec* audit record with a record length of at least 500 bytes.
- Only 0.15 per cent of the normal *exec* audit records were longer than 400 bytes.

Consequently, by setting the threshold to 400 and adding the conditions for *setuid* and control characters, false positives are effectively eliminated while exploits of the described type are detected.

4.2 Network-based traffic analysis

In addition to its extensive application to the area of audit trail analysis, P-BEST is now being applied to the analysis of network traffic streams. This work includes the analysis of TCP/IP packet streams for low-level TCP and IP layer attacks (i.e., attacks that target vulnerabilities at the transport layer and below) as well as higher-layer attacks involving vulnerabilities of application-layer (or network service-layer) protocols, such as FTP, SMTP, and HTTP.

4.2.1 Attack description: SYN flood attack The SYN flood attack is a denial-of-service attack that prevents the target machine from accepting new connections to a given IP port [17]. Briefly, the attack exploits a resource exhaustion vulnerability in the way operating systems handle TCP/IP connections. A TCP/IP connection is established through a three-step handshake, in which the client sends a SYN packet, followed by the server responding with a SYN-ACK packet, which is then acknowledged by the client with an ACK packet. Of course, by no means is there an expectation that all TCP/IP handshakes run to completion. When the SYN packet is received, the server allocates an entry in a finite queue of pending connections. We refer to this stage as a *half-open* connection. The queue entry will either be released when the final ACK is received by the server, or the server will proceed to timeout the incomplete handshake and release the entry.

An attacker can exploit the TCP/IP connection logic by initiating a series of SYN packet connection requests to a server, but not completing the handshakes with an ACK packet. Internally, the server's queue of pending connections for the port will eventually be exhausted and will not be released until the timeout periods for the unfinished connections expire. As a result, subsequent connection requests to the server that occur while the connection queue is full will be dropped, effectively denying access to the server by other legitimate clients.

4.2.2 Event stream format The requirements for detecting the occurrence of a SYN flooding attack against a host are rather minimal. From the perspective of TCP/IP traffic monitoring, the analysis engine need only monitor SYN-ACK and ACK packet exchanges to identify incomplete TCP/IP handshakes. In this example, the traffic monitor is placed on a segment of the network capable of observing traffic to and from the analysis target (the host being monitored). All SYN-ACK packets sent from—and ACK packets sent to—the analysis target are recorded, and the following event record is derived:

```
Connection Event Format:
 <Event_Type> <Timestamp> <Seq_ID> <Client_ID>
```

The Event_Type field is simply a binary flag, which indicates whether the packet has its SYN and ACK flags enabled (which we can denote with 0), or only the ACK flag enabled (denoted by 1). The timestamp is a numeric encoding of the time at which the packet is observed from the monitor. The sequence ID represents the TCP Sequence ID field, which is used to associate client requests with server replies. Last, the Client_ID can be used to identify the client who initiated the connection. The Client_ID is not critical for detection, and in all likelihood will not be reliable (i.e., attackers will manufacture IP packets with bogus IP source addresses). Nevertheless, we may choose to capture such information as the IP address and port number of the client packet for reporting purposes only.

4.2.3 P-BEST fact type definitions Table 2 illustrates the ptype definitions of three example facts that are specified for use in performing the TCP SYN flooding analysis. The first ptype, `conn_event`, is used to assert the connection event described in the connection event record format discussed above. As connection events are captured by the network monitor, their fields can be mapped (one to one) to the fields of the `conn_event` ptype, and the `conn_event` ptype is then asserted into the factbase of the SYN flood eXpert. The `open_conn` ptype is used to construct facts regarding half-open connections that are pending completion of the TCP/IP handshake. Note, although we use the shorthand name `open_conn`, the fact actually represents the assertion that a TCP *half-opened* connection has been observed. The fields of the `open_conn` contain the TCP sequence ID of the pending connection, a `client_ID` string (as discussed above), the timestamp as

Table 2. Facts for TCP SYN flood detection.

1	`ptype[conn_event`	`ptype[open_conn`	`ptype[bad_conn`
2	`e_type:integer,`	`expired:integer,`	`count:integer]`
3	`sec:integer,`	`sec:integer,`	
4	`seq_id:integer,`	`seq_id:integer,`	
5	`client_ID:string]`	`client_ID:string]`	

copied from the connection event, and an expired flag used for garbage collection by the production rules. Last, the bad connection fact, `bad_conn`, maintains a running count of the number of bad connection requests detected through the observations of SYN-ACK and ACK packages between the analysis target and external clients.

4.2.4 Example P-BEST rules for SYN flood detection The following illustrates one inference strategy that P-BEST can employ for deducing a TCP SYN flooding attack, using the fact definitions defined above. In addition, a few constants are referenced from the rule set, and are defined as follows:

- `max_bad_conns:` Number of bad connections tolerated before SYN flood alert.
- `expire_time:` Amount of time to wait on ACK before a connection is declared a bad connection.
- `bad_conn_life:` Number of seconds that a bad connection fact will live before being released.

Abstractly, the rules attempt to identify half-open TCP connections that expire beyond a user-defined waiting period. As we assert half-open connection facts into our factbase, we must include logic to recognize both when the connections are successfully completed and when half-open connection expire beyond the user-defined waiting period, from which we deduce the occurrence of a bad connection. SYN flood attacks will result in excessive bursts of bad connections, which we monitor with rules that maintain a running count of bad connections over a sliding window of time. When the number of bad connections exceeds our maximum tolerance for bad connections within our sliding time window, we raise an alert to denote the burst of noncompleted connection requests. The following is a brief summary of the rule set shown in Table 3.

- `create_open_conn`: determines whether the event connection represents a SYN-ACK packet (from the monitor target). If so, the rule asserts a new fact into the factbase called `open_conn`, which records the TCP sequence number, the timestamp at which this half-opened connection was first observed, an expired flag to indicate when the half-open connection exceeds a time threshold, and the `client_ID`.
- `destroy_open_conn`: removes an open connection fact when the corresponding ACK packet is received from the client.
- `ignore_spurious_acks`: removes events involving ACK packets that are not associated with a specific SYN-ACK pending connection. In practice, such packets are normal.

Table 3. Rule set for detection of TCP SYN flood attacks.

```
rule[create_open_conn(*):
   [+ev:conn_event|e_type == 0]
==>
   [+open_conn |seq_id = ev.seq_id,
                sec = ev.sec,
                expired = 0,
                client_ID = ev.client_ID]
   [-|ev]
]
```

```
rule[destroy_open_conn(*):
   [+ev:conn_event|e_type == 1]
   [+oc:open_conn|seq_id == (ev.seq_id - 1),
                  expired == 0]
==>
   [-|oc] [-|ev]
]
```

```
rule[ignore_spurious_acks(*):
   [+ev:conn_event|e_type == 1]
   [-open_conn|seq_id == (ev.seq_id - 1)]
==>
   [-|ev]
]
```

```
 rule[first_bad_conn(*):
   [+ts:time]
   [-bad_conn]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > 'expire_time]
==>
   [+bad_conn|count = 1]
   [/oc| expired = 1]
]
```

continues on next page

continued from previous page

```
rule[add_to_bad_cons(*):
   [+ts:time]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > ’expire_time]
   [+bc:bad_conn|count < ’max_bad_conns]
==>
   [/bc|count += 1]
   [/oc|expired = 1]
]
```

```
rule[max_open_cons(*):
   [+ts:time]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > ’expire_time]
   [+bc:bad_conn|count == ’max_bad_conns]
==>
   [!|syn_alert("SYN Attack: Last Host %s.\
      SeqID = %d. Time = %d",
       oc.client_ID, oc.seq_id, oc.sec)]
   [/bc|count = 0]
   [/oc|expired = 1]
]

```

```
rule[free_bad_open_cons(*):
   [+ts:time]
   [+bc:bad_conn]
   [+oc:open_conn|expired == 1]
   [?|(ts.sec - oc.sec) > ’bad_conn_life]
==>
   [-|oc]
   [/bc|count -= 1]
]
```

```
rule[del_alerted_cons
   [+oc:open_conn|expired == 1]
   [+bad_conn|count == 0]
==>
   [-|oc]
]
```

Minor corrections were made to this table after the original publication.

• `first_bad_conn`: This and the following rule manage a running count of the set of bad connections observed by the inference engine. They are driven by time facts (line 24) which are used to monitor whether there exists a half-open connection that has exceeded the `expire_time` limit. This rule is applied once, to the first `open_conn` fact encountered that is older than `expire_time`. Its consequent creates the `bad_conn` fact, which initializes the bad connection counter upon the first encountered expired connection. Note that the antecedent line 25 evaluates to false once the `bad_conn` fact has been initialized. In addition, the rule marks the `open_conn` fact as expired (line 30), which is consulted by `free_bad_open_cons` when performing garbage collection.

• `add_to_bad_cons`: is applied while the total number of `bad_conn` facts is less than the maximum tolerated. If an `open_conn` fact timestamp exceeds the expiration time and the fact has not been counted earlier, then the `bad_conn` count is incremented, and the expired flag for the `open_conn` fact is set.

• `max_open_cons`: is applied when the maximum number of `bad_conn` facts is encountered during a burst of `bad_conn_life` time units. If a `bad_conn` count reaches the maximum tolerated `bad_conn` facts, the consequent initiates a SYN flood alert, and resets the bad connection count.

• `free_bad_open_cons`: limits the amount of time that a bad open connection is counted against the system. The `bad_conn_life` variable provides a user-defined length of time with which a bad connection is considered relevant to the bad connection count. This variable effectively represents the burst duration for accumulating bad connections. Once an open connection exceeds the `bad_conn_life`, then it is removed and the bad connection count is reduced.

• `del_alerted_cons`: deletes the half-open connections that have caused an alert.

5 Performance

There are a variety of factors that influence the amount of time required to process records through a P-BEST-based signature analysis engine. In this section, we briefly discuss some of these factors and summarize several performance measurements in analyzing both Solaris audit records and TCP packets through an EMERALD eXpert P-BEST engine. These measurements are intended to reflect the pure processing time required by the eXpert in receiving events, translating and asserting the events into the eXpert fact base, processing the events through the inference engine, and handling alert reporting.

The measurements exclude the processing time added to the system for event generation; that is, it excludes the impact to system resources in audit record generation or the capturing and filtering of TCP packets. It is difficult to estimate the daily expected volumes of audit and network traffic across a computing environment, in that such statistics are directly dependent on the structure of the computing environment, network topology, and behavior and size of the user community. Furthermore, the EMERALD architectural model lends itself well to the separation of the event generation and collection components from the analytical engines, which could in fact operate in parallel on separate hosts.

Table 4. Performance of sample BSM and TCP analysis engines.

	24 hrs BSM 43 users 365 MB total 1.1 million recs	120 hrs BSM 44 users 1.41 GB total 4.2 million recs	24 hrs IP 496 connects 331 MB total 83,002 recs	120 hrs IP 1,343 connects 1.3 GB total 352,445 recs
1 rule set 2 rules buffer overrun	4:10 min:sec	15:41 min:sec	—	—
16 rule sets 28 rules various intrusions	8:09 min:sec	30:53 min:sec	—	—
1 rule set 12 rules TCP SYN flood	—	—	1:33 min:sec	3:02 min:sec

The performance measurements were collected on a FreeBSD 2.2.6 host computer system using a Pentium II 333 Mhz processor with 128 MB RAM. In addition to the processing capabilities of the host platform, there are several factors that significantly influence the overall performance of the analysis engine. For example, the average record size and total event stream size dictate the amount of I/O overhead required. As each event is asserted by the rule base, the antecedent evaluation also impacts performance: the sheer number of rules to evaluate, as well as the complexity of each antecedent evaluation, significantly influence event processing throughput. Consequent activation is also a consideration, as is the management of derived facts that are asserted during the analysis.

Table 4 presents a summary of three analyses performed on 1 and 5 day collections of Solaris 2.5.1 audit records and TCP packet streams. The audit and TCP data sets were collected by MIT Lincoln Laboratories, and made available for the DARPA Intrusion Detection Evaluation Program. The BSM audit logs analyzed here represent the simulated usage of a server with 43 users over one 24 hour period and 44 users over a 5 day work week, with minimal filtering. While it is difficult to generalize what such loads imply for other computing environments, the data set is representative of the volume and type of audit activity observed during a prolonged study of several Air Force local area networks.

The first row in Table 4 summarizes the performance of an EMERALD eXpert implementing the buffer overflow rule presented in Section 4.1, which is roughly able to apply this rule to 24 hours of audit data (over one million audit records) in 4 minutes, and 120 hours of audit data (4.2 million audit records) in under 16 minutes. In the second row, we present an eXpert with a more extensive collection of 28 rules. These rules implement 16 sets of Solaris BSM intrusion detection heuristics, including threshold analyses, immediate attack recognition, process subversion detection, and illegal file access recognition. While the knowledge base of this second eXpert represent an increase of fourteen fold over the 2-rule eXpert system in row one, it introduces only a two fold increase in the overall processing time of the 1 and 5 day data sets. In this computing environment, the 16 rule sets can process the full five day data set in just over 30 minutes; this represents a small fraction of the overall audit generation time.

The third and fourth columns of Table 4 present an analysis of TCP/IP traffic through a gateway that provides service between an internal domain of 4 servers and 20 workstations, and an external untrusted network. Row three of Table 4 summarizes the performance of the TCP SYN flood detection rules presented in Section 4.2 (with a few additional administrative rules). Here, a server was selected for analysis, and all TCP packets sent to and from it were monitored for 24 and 120 hours, during which 496 and 1,343 connections were observed over 24 and 120 hours, respectively. The SYN Flood eXpert monitored only those TCP packets targeted for the host of interest in which the SYN or ACK flags were enabled. The filtering out of unnecessary packets is critical to managing the performance of a real-time signature analysis engine, and in the SYN flooding case, the criteria for analysis excludes all packets that are not directly involved in the TCP handshake. In our simulated analysis, the SYN Flood eXpert is capable of performing the 24 hour packet analysis in 1.5 minutes, and the 120 hour analysis in 3 minutes.

6 Related work

P-BEST has evolved over a substantial lineage of intrusion detection projects, which include MIDAS, IDES, NIDES, and now the EMERALD eXpert. It represents a very early example of the application of a forward-chaining rule-based expert system to the problem of misuse detection in computer system activity logs. However, P-BEST is by no means the only system to have applied rule-based expert system techniques to detecting misuse in computing environments.

Several other systems have been developed that also center around the use of forward-chaining inference logic, and have applied a variety of techniques for representing the underlying heuristics used to represent misuse. The ASAX (Advanced Security and Audit Trail Analysis on UniX) project [9], produced a highly specialized rule-based programming language called RUSSEL (Rule Based Sequence Evaluation Language), which provides a combination of procedural and rule-based programming constructs to reason about activity in Unix audit trails.

The University of California at Santa Barbara proposed the use of state transition diagrams to model the sequence of operations and state changes that occur during the execution of a penetration [15]. This technique was prototyped for SunOS 4.1.3+ and Solaris audit trails in a tool called the Unix State Transition Analysis Tool (USTAT) [10]. While it did not represent its knowledge base using production rules, USTAT was architected as a classic expert system, with an inference engine, knowledge base, fact base, and separate decision engine. Another system, called IDIOT (Intrusion Detection In Our Time), took a similar graphical approach to the analysis of signature operations, but used Colored Petri-nets to model its analysis of the patterns of execution represented in an event stream [13].

Wisdom and Sense [20] and NADIR [11], both from Los Alamos National Laboratory, are further examples of intrusion detection systems that employed rule-based analyses to identify known malicious activity. In the case of W&S, the anomaly detection component was also implemented as a rule-base. The signature analysis component was combined into the same rule-base to represent site-specific policies, expert penetration rules and other administrative data. NADIR's expert rule-base consists of penetration rules that are developed by interviewing and working with security personnel.

Last, it is important to recognize a continuing growth in the number of commercial products that provide forms of signature analysis for various computing environments. Given the proprietary nature of these systems, it is difficult to understand which have chosen hard-coded narrow solutions to their problem sets, and which have chosen more broad techniques that may be portable beyond their current customers' needs.

7 Limitations

In Section 2 we attempted to summarize how and why forward reasoning systems provide a good foundation for modeling known abusive activity represented in an event stream. There are, of course, limitations that are fair to point out with respect to this general method. In our own system, antecedent evaluation is absolute, and less capable in environments where uncertainty, incompleteness, or inaccuracies exist within the event stream content. Other reasoning systems can provide some options for handling belief and uncertainty within the analysis framework [8]. In the presence of incomplete data, backward reasoning systems can operate in a diagnosis mode to seek out collaborative evidence of problems, and furthermore provide quantitative probabilities based on "evidence to date" that a certain problem is the culprit responsible for the presence of given symptoms. Such reasoning capabilities could be valuable if applied well to the intrusion detection domain.

In addition to event stream inadequacies, heuristics presuppose the existence of detailed insight into that which constitutes abusive system activity. The problem of recognizing and responding to *unknown* malicious phenomena is extremely difficult, and not directly addressed under signature analysis. Only in the cases where it is possible to look for certain *results*—rather than explicit action sequences leading to those results—does signature analysis have a chance to detect new attack methods. For example, if an anonymous user causes the deletion of a file from our FTP server, we can detect this result without knowing exactly how the attack was carried out.

Other techniques that attempt to understand *normal* system operation and to provide quick recognition of anomalous activity have been proposed; statistical profiling [5], neural networks [4], and sequence analysis [7]. The intent of these systems is to maximize the points at which anomalous activity corresponds to malicious activity, which as a general property does not always hold. In addition, attempting to maximize such systems' sensitivity to malicious activity also tends to increase their sensitivity to inane anomalies.

8 Future work

In parallel with our current academic experiments with P-BEST, we are developing an Internet-accessible P-BEST translation service, which will allow users to develop and compile rule sets into self-contained expert systems. Linkage modules will be provided to allow users to feed the expert system Solaris 2.5.+ audit records and TCP/IP packets in batch and real time. Users will be provided an HTML interface from which ptype definitions and production rules can be submitted to the P-BEST translation service. The translation service will attempt to compile an expert system based on the ptypes and rules; if successful, the user will receive a URL link from which the expert system can be downloaded and tested in the user's own environment. If errors are identified in the rules or ptype declarations, a summary of the errors will be returned to the user for revision. In addition, a simple reporting utility will be provided to convert alerts generated by the expert system to HTML or email notifications. We will make the following components available to other universities interested in conducting classroom experiments involving signature-based intrusion detection:

- P-BEST expert system generation service available via HTML-based interface
- Solaris audit and TCP/IP batch and real-time event collection interface modules
- HTML and email alert reporting interface module
- Language manuals and supporting documentation (including exploit detection exercises) developed in support of our current university classroom experiment

For more information on the Internet-accessible P-BEST translation service for academic experimentation, the reader may refer to the following URL:
`http://www.csl.sri.com/emerald/`

9 Conclusion

We have presented the operation of a production-based expert system toolset, and its application to the problem of computer and network signature-based intrusion detection. P-BEST has had considerable exposure to the intrusion detection problem domain over the past decade, under the MIDAS, IDES, and NIDES projects, and now within the EMERALD eXpert. P-BEST has been employed on a Symbolics processor for handling Multics audit records, SunOS 4.1.+, Solaris 2.5.+, FreeBSD, and Linux for real-time audit trail analysis, accounting log analysis, and TCP/IP packet analysis.

We presented details of the P-BEST production rule specification language, and illustrated its use with example rule sets for detecting misuse in Solaris 2.5.+ audit trails and TCP/IP packet streams. We also discussed the performance of P-BEST inference engines in analyzing millions of events, which illustrates that P-BEST has been—and continues to be—useful in live monitoring of computer and network operations.

In addition, work is in progress to move P-BEST into academic environments, where it will be made openly available as an instructional tool for illustrating signature-based intrusion detection. P-BEST is currently being used for laboratory exercises in one university course on applied computer security, where students are guided through its usage and assigned rule development tasks for analyzing given intrusions. We have demonstrated that the P-BEST language is not too complex for beginners to employ, and is efficient for supporting the iterative development of increasingly complex inference logic for automated reasoning about misuse in computer and network operations.

Acknowledgments

The authors are grateful to Alan Whitehurst, the original author of P-BEST, and to Fred Gilham who has made later enhancements and documentation and has answered many questions from us. Several people have participated in discussions and contributed with ideas and suggestions; we thank Peter Neumann, Keith Skinner, and Martin Fong at SRI and Erland Jonsson and Stefan Axelsson at Chalmers. We also thank the Chalmers students who participated in the P-BEST lab exercise.

References

[1] D Anderson, T Frivold, and A Valdes. Next-generation intrusion-detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1995.

[2] Stefan Axelsson, Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. An approach to UNIX security logging. In *Proceedings of the 21st National Information Systems Security Conference*, pages 62–75, Arlington, Virginia, October 5–8, 1998. National Institute of Standards and Technology/National Computer Security Center.

[3] D Bruschi, E Rosti, and R Banfi. A tool for pro-active defense against the buffer overrun attack. In Jean-Jacques Quisquater et al., editors, *Computer Security – Proceedings of ESORICS 98*, volume 1485 of *LNCS*, pages 17–31, Louvain-la-Neuve, Belgium, September 16–18, 1998. Springer-Verlag.

[4] Hervé Debar, Monique Becker, and Didier Siboni. A neural network component for an intrusion detection system. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 240–250, Oakland, California, May 4–6, 1992. IEEE Computer Society Press, Los Alamitos, California.

[5] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[6] Dorothy E Denning and Peter G Neumann. Requirements and model for IDES—a real-time intrusion detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, 1985.

[7] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, California, May 6–8, 1996. IEEE Computer Society Press, Los Alamitos, California.

[8] Thomas D Garvey and Teresa F Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, pages 372–385, Washington, D.C., October 1–4, 1991. National Institute of Standards and Technology/National Computer Security Center.

[9] Jani Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In Yves Deswarte et al., editors, *Computer Security – Proceedings of ESORICS 92*, volume 648 of *LNCS*, pages 435–450, Toulouse, France, November 23–25, 1992. Springer-Verlag.

[10] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[11] Kathleen A Jackson, David H DuBois, and Cathy A Stallings. An expert system application for network intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, pages 215–225, Washington, D.C., October 1–4, 1991. National Institute of Standards and Technology/National Computer Security Center.

[12] Harold S Javitz and Alfonso Valdes. The SRI IDES statistical anomaly detector. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 316–326, Oakland, California, May 20–22, 1991. IEEE Computer Society Press, Los Alamitos, California.

[13] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1995.

[14] Teresa F Lunt, R Jagannathan, Rosanna Lee, Alan Whitehurst, and Sherry Listgarten. Knowledge-based intrusion detection. In *Proceedings of the Annual AI Systems in Government Conference*, pages 102–107, Washington, D.C., March 27–31, 1989. IEEE Computer Society Press, Los Alamitos, California.

[15] Phillip A Porras and Richard A Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, pages 220–229, San Antonio, Texas, November 30–December 4, 1992. IEEE Computer Society Press, Los Alamitos, California.

[16] Phillip A Porras and Peter G Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, October 7–10 1997. National Institute of Standards and Technology/National Computer Security Center.

[17] Christoph L Schuba, Ivan V Krsul, Markus G Kuhn, Eugene H Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223, Oakland, California, May 4–7, 1997. IEEE Computer Society Press, Los Alamitos, California.

[18] Michael M Sebring, Eric Shellhouse, Mary E Hanna, and R Alan Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, pages 74–81, Baltimore, Maryland, October 17–20, 1988. National Institute of Standards and Technology/National Computer Security Center.

[19] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA. *SunSHIELD Basic Security Module Guide*, November 1995.

[20] H S Vaccaro and G E Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, California, May 1–3, 1989. IEEE Computer Society Press, Los Alamitos, California.

This page is intentionally left blank.

Paper H

Designing IDLE: The Intrusion Data Library Enterprise

Abstract presented at RAID ’98 (First International Workshop on the Recent Advances in Intrusion Detection), Louvain-la-Neuve, Belgium, Sept. 14–16, 1998.

This page is intentionally left blank.

Designing IDLE: The Intrusion Data Library Enterprise

Ulf Lindqvist[1,2] Douglas Moran* Phillip A. Porras[2] Mabry Tyson[3]

[1]Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden
ulfl@ce.chalmers.se

[2]Computer Science Laboratory
[3]Artificial Intelligence Center
SRI International
Menlo Park, California
{ulf, porras}@csl.sri.com
tyson@ai.sri.com

Abstract

High quality, timely information on intrusions is crucial in the development, testing, tuning, and updating of intrusion detection systems (IDSs) and intrusion recovery systems. We present the Intrusion Data Library Enterprise (IDLE), a design and initial compilation of an extensible library of intrusion data that is efficiently parseable in both human-readable and platform-independent machine-readable forms. The IDLE format will be made available as a resource specifically for the intrusion detection community. IDLE will provide IDS developers and users with accurate field data for testing and tuning and, as new intrusion types are discovered, it will enable tools to automatically update rule sets and parameters.

1 Background and motivation

As developers of tools for intrusion detection and diagnosis, we have identified a need for data on intrusions. Typically, we would like to have a collection of reliable, detailed records that include exploit instructions and data on vulnerable system configurations to make the intrusions repeatable in a lab environment. We would also like to be able to add various types of information, such as what indicators of the intrusion we are able to observe in the system. Legitimate concerns about distributing information on intrusion schemes has hampered the growth of such databases. Ironically, this has led to the current situation where security professionals find their best sources for intrusion data on underground Internet sites.

Our experience from detailed intrusion analysis indicates that a vast amount of information is needed about the particular intrusion sample to make correct statements about an intrusion type in terms of prerequisites, impact, traces, difficulty,

*Author's present address: Recourse Technologies, Inc., 2450 El Camino Real, Palo Alto, CA 94306-1706, USA, dmoran@recourse.net remedies etc. Also, because systems are continually patched to block known intrusions, it can be difficult to recreate a vulnerable system configuration for each intrusion sample when detailed vulnerability information is missing.

Typically, intrusion handling systems need to be updated when new types of intrusions are discovered. A major problem facing intrusion detection system (IDS) developers is that the intrusion databases utilized provide little leverage for automating extensions to their systems. In many systems it is assumed that the users will collect intrusion descriptions from various sources and write new rules or change parameters as new intrusions are discovered. This could be compared to modern virus detection systems, which often have automatic network-based update functionality.

As a solution to this problem, we present the Intrusion Data Library Enterprise (IDLE) which is an effort to create a standard format for describing vulnerability and exploit information for IDS purposes.

2 Related work

Most of the previously presented databases for system vulnerabilities and/or exploits fall into one of two different categories, each with their own shortcomings, respectively:

- Databases that are kept internal to an organization.

  Problems: People outside the owner organization do not know about the databases, their contents, or their structure. Sharing is discouraged or even prohibited, motivated by liability issues or the simple fact that organizations do not want to reveal how their systems could be attacked.

- Publicly available databases from “underground” sources.

  Problems: Data on such sites is often incomplete, of varying reliability, and certainly not intended for intrusion detection.

To our knowledge, the only previously published work on vulnerability database structure that talks about storing signatures for intrusion detection is that by performed by Krsul at Purdue University [5]. However, the main purpose of Krsul’s database is the same as that of the often-cited Landwehr taxonomy [6], namely, vulnerability analysis and prevention rather than intrusion detection.

Vulnerability prevention was also the original goal of the vulnerability database project managed by Matt Bishop at University of California at Davis, but it has come to include intrusion signatures as well [2]. We currently seek to coordinate the efforts in IDLE with the Davis project.

A threat description language specifically targeted for computer viruses was suggested by Brunnstein *et al.* [3]. A main difference between viruses and general intrusions is that it is often more difficult to describe an exact signature in the general case.

At Chalmers, we have previously worked on categorization of many aspects of vulnerabilities and intrusions, including cause [7, 11], method and result [8], risk [9], traces [1], and remedy [10]. That work will serve as a foundation for the IDLE structure.

3 IDLE: A standard format for data sharing

In IDLE, we attempt to create a standard format that will facilitate rapid distribution of information among IDS developers and related groups in order to achieve "critical mass" in the coverage. Although the breadth of such exchange and the access controls are outside the scope of this work, we would like to point out that IDLE supports fine-grained data filtering that can be used to implement various policies for data sharing and sanitization.

3.1 IDLE design highlights

The emphasis of our design is to include information that ensures automated repeatability, detection, and diagnosis of each intrusion sample. What makes IDLE different from current intrusion databases is that it is designed to serve the IDS community by coordinating detailed information on vulnerable configurations and exploit instructions with documented observable dynamic and static traces (indicators) of the intrusion type. The IDLE trace information is structured in a form that will support an IDS downloading a new description and extracting the information needed to automatically generate new rules (signatures, parameters, etc.) to identify the new intrusion.

IDLE is designed to store technical data about intrusion *types*, where the primary distinction between two types is that their observable traces differ in a significant way. Another distinction between types is information that concern the vulnerable configurations, for example, operating system, applications etc. However, IDLE is not meant to be a database of intrusion *incidents* where evidence concerning attack cases should be stored. Such a database would have other concerns and goals [4], but could use IDLE for the technical description of the types of intrusions occurring in the recorded incidents.

Support for partial information is a core part of the design of IDLE. Information about an intrusion type will typically be initially incomplete, and different groups may tend to populate only chosen subsets of a record. Incremental population is especially important in the observables, because different developers monitor system activity from different perspectives: network traffic, audit logs, application logs, filesystem traces, etc. IDLE must also be easily extensible to support the aspects relevant to new and different target platforms, tools, and IDSs. In summary, there are three aspects of incremental population:

- New entries (intrusion types) in the repository
- New data for certain fields of an existing entry
- New types of fields for new and old entries

It should also be noted that published records may be incomplete due to sharing policies requiring certain parts of the database to be suppressed.

3.2 Implementation

We have chosen to use the Extensible Markup Language (XML) [13], proposed successor to HTML, for the intrusion database. There is an intense ongoing development of tools for authoring, displaying, browsing and handling XML documents, and we expect substantial leverage from this activity. XML provides a number of features such as platform-independence, naturally hierarchical structure, customizable field display filtering, the possibility to mix human-readable free-text fields and machine-readable fields in the same record, and easy addition of new types of fields. Those features enable IDLE to evolve both in terms of the content of individual records and of the structure of the library. This makes us confident that for IDLE, XML is a far better choice than an existing proprietary database format.

We will now give some examples of what the first draft structure of IDLE looks like, both in terms of element hierarchy, XML code examples and conversion to HTML for presentation. We have chosen to document an intrusion in which a so-called buffer overflow vulnerability in some versions of the Solaris operating system is exploited to gain administrator privileges.

In Figure 1, the hierarchical document structure for the intrusion record is shown from the top level. Lower level elements are expanded in Figures 2, 3 and 4. An example of what the actual XML code can look like is shown in Figure 5 which corresponds to the CONFIRM element in Figure 2. A presentation specification language candidate for XML is the Extensible Style Language (XSL), which in itself is an application of XML. We have written XSL style files for conversion of our XML documents to HTML for viewing with standard Web browsers. An example of XLS code is shown in Figure 6 and the resulting browser view is shown in Figure 7.

3.3 Current status

Currently, we are populating the database with example entries to evaluate and refine the first draft structure. The objective is to be able to freeze a first Document Type Definition (DTD) against which new entries can be formally validated by an XML parser. The design of a DTD, together with an evaluation of suitable tools for editing and database management is under way [12].

For presentation of the database contents, automatic generation of HTML controlled by mapping rules written in the Extensible Style Language (XSL) has been implemented. This conversion illustrates how data stored in XML easily can be filtered, split, hyperlinked and interpreted.

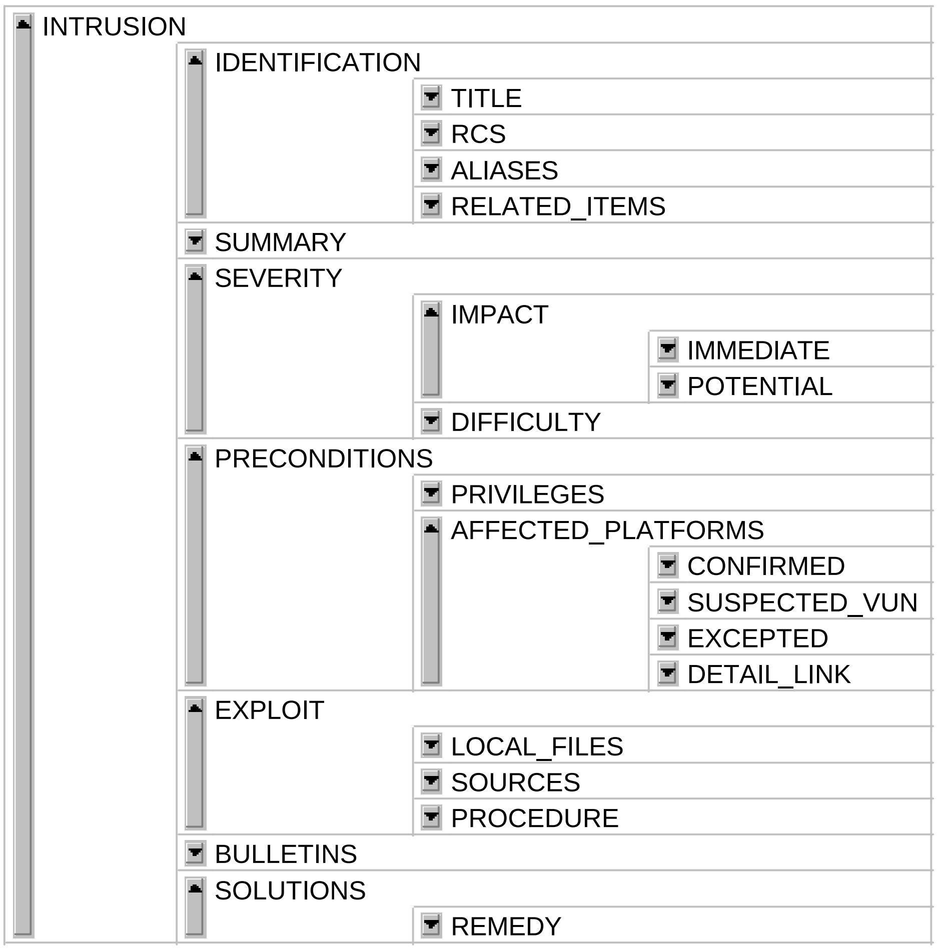


Figure 1. Top level view of the document structure.

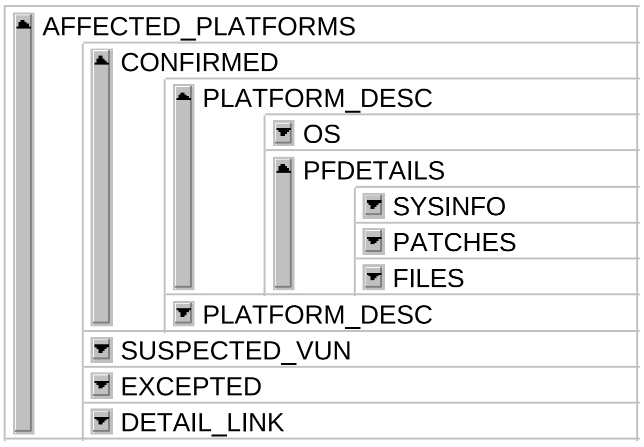
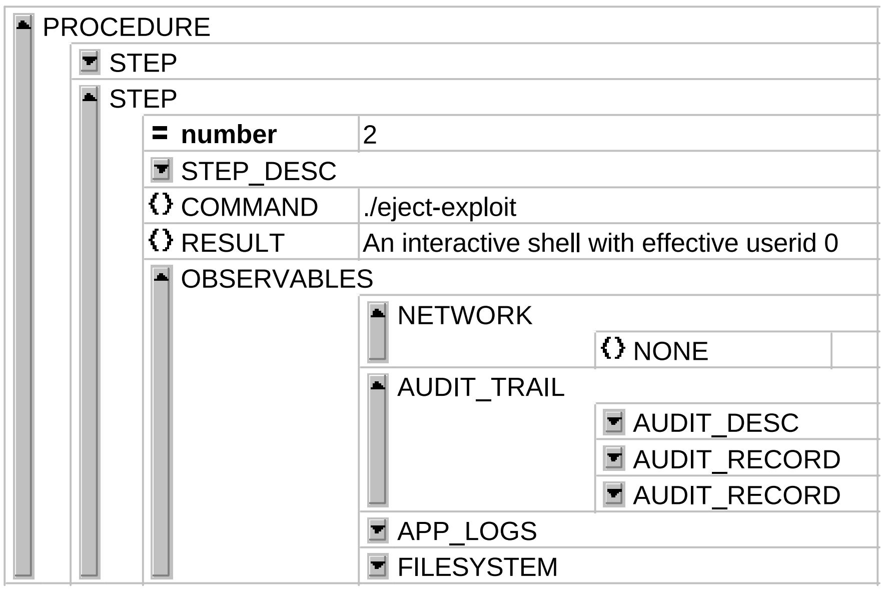
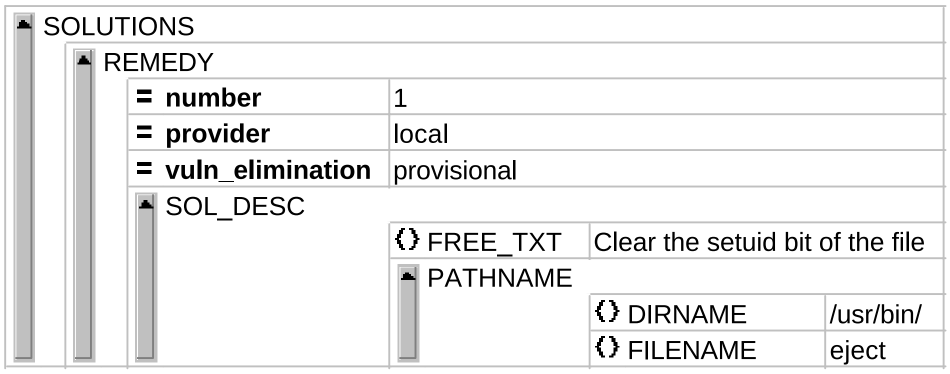


Figure 2. Detailed view of the CONFIRMED element.



```
PROCEDURE
    STEP
    STEP
        = number          2
          STEP_DESC
        {} COMMAND        ./eject-exploit
        {} RESULT         An interactive shell with effective userid 0
          OBSERVABLES
                          NETWORK
                                      {} NONE
                          AUDIT_TRAIL
                                         AUDIT_DESC
                                         AUDIT_RECORD
                                         AUDIT_RECORD
                          APP_LOGS
                          FILESYSTEM
```

Figure 3. Detailed view of PROCEDURE and OBSERVABLES.



```
SOLUTIONS
    REMEDY
        = number             1
        = provider           local
        = vuln_elimination   provisional
          SOL_DESC
                             {} FREE_TXT   Clear the setuid bit of the file
                                PATHNAME
                                           {} DIRNAME    /usr/bin/
                                           {} FILENAME   eject
```

Figure 4. Detailed view of SOLUTIONS and REMEDY.

```xml
<CONFIRMED>
  <PLATFORM_DESC>
    <OS>
      <OSNAME>Solaris</OSNAME>
      <OSREV>2.5</OSREV>
    </OS>
    <PFDETAILS>
      <SYSINFO command="uname -a">SunOS...</SYSINFO>
      <PATCHES command="showrev -p">No patches are installed
      </PATCHES>
      <FILES>
        <FSENTRY>
          <PATHNAME>
            <DIRNAME>/usr/bin/</DIRNAME>
            <FILENAME>eject</FILENAME>
          </PATHNAME>
          <CHECKSUM algorithm="MD5">35d...</CHECKSUM>
          <FILESIZE unit="bytes">9680</FILESIZE>
          <FILEPERMISSIONS>-r-sr-xr-x</FILEPERMISSIONS>
          <FILEOWNER uid="0">root</FILEOWNER>
          <FILEGROUP gid="2">bin</FILEGROUP>
        </FSENTRY>
      </FILES>
    </PFDETAILS>
  </PLATFORM_DESC>
  <PLATFORM_DESC>
    <OS>
      <OSNAME>Solaris</OSNAME>
      <OSREV>2.5.1</OSREV>
    </OS>
    <PFDETAILS>
      <SYSINFO command="uname -a">SunOS...</SYSINFO>
      <PATCHES command="showrev -p">No patches are installed
      </PATCHES>
      <FILES>
        <FSENTRY>
          <PATHNAME>
            <DIRNAME>/usr/bin/</DIRNAME>
            <FILENAME>eject</FILENAME>
          </PATHNAME>
          <CHECKSUM algorithm="MD5">206...</CHECKSUM>
          <FILESIZE unit="bytes">9676</FILESIZE>
          <FILEPERMISSIONS>-r-sr-xr-x</FILEPERMISSIONS>
          <FILEOWNER uid="0">root</FILEOWNER>
          <FILEGROUP gid="2">bin</FILEGROUP>
        </FSENTRY>
      </FILES>
    </PFDETAILS>
  </PLATFORM_DESC>
</CONFIRMED>
```

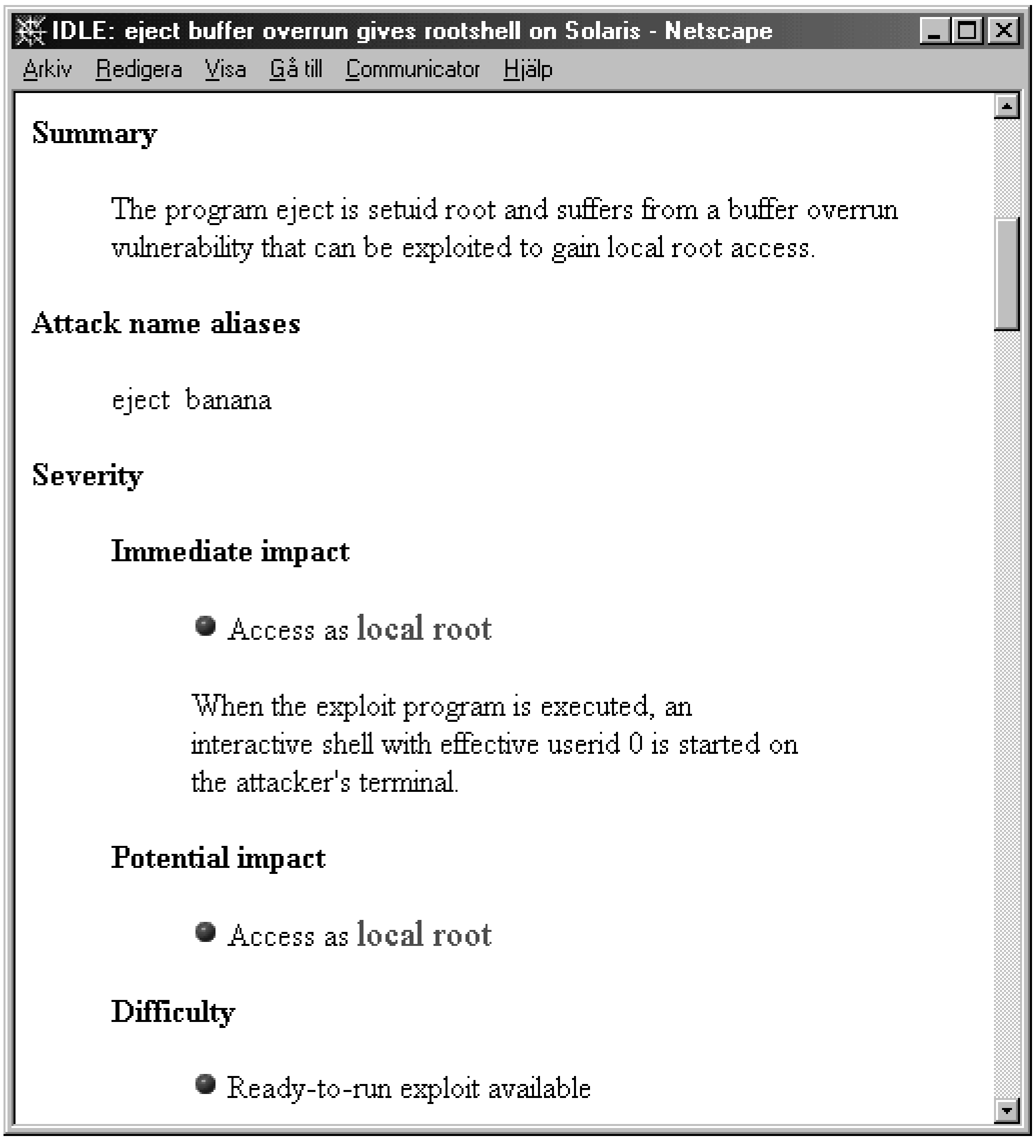
Figure 5. An example of the CONFIRMED element in XML.

```
<rule>
<target-element type="SUMMARY"/>
  <H4>Summary</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>

<rule>
<target-element type="ALIASES"/>
  <H4>Attack name aliases</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>

<rule>
<element type="ALIASES">
  <target-element
    type="ATTACKNAME"/>
</element>
  <children/><TT> </TT>
</rule>

<rule>
<target-element
  type="SEVERITY"/>
```

```
  <H4>Severity</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>

<rule>
<element type="IMPACT">
  <target-element
    type="IMMEDIATE"/>
</element>
  <H4>Immediate impact</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>

<rule>
<element type="IMPACT">
  <target-element
    type="POTENTIAL"/>
</element>
  <H4>Potential impact</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>
```

```
<!-- We want to match the LOCALROOT element
     one step below IMPACT -->
<rule>
<element type="IMPACT">
  <element>
    <target-element type="LOCALROOT"/>
  </element>
</element>
  <P><IMG SRC="pictures/reddot.gif"/>Access
       as <FONT SIZE="+1" COLOR="red">local root</FONT></P>
</rule>

<rule>
<target-element type="DIFFICULTY"/>
  <H4>Difficulty</H4>
  <BLOCKQUOTE>
  <children/>
  </BLOCKQUOTE>
</rule>

<rule>
<element type="DIFFICULTY">
  <target-element type="READYEXPLOIT"/>
</element>
  <P><IMG SRC="pictures/reddot.gif"/>Ready-to-run
                            exploit available</P>
</rule>
```

Figure 6. An example of XSL code for conversion of XML to HTML.


IDLE: eject buffer overrun gives rootshell on Solaris - Netscape
Arkiv Redigera Visa Gå till Communicator Hjälp

Summary

The program eject is setuid root and suffers from a buffer overrun vulnerability that can be exploited to gain local root access.

Attack name aliases

eject banana

Severity

Immediate impact

- Access as local root

When the exploit program is executed, an interactive shell with effective userid 0 is started on the attacker's terminal.

Potential impact

- Access as local root

Difficulty

- Ready-to-run exploit available

Figure 7. The resulting HTML file as viewed with the Netscape browser.

4 Open issues and future work

IDLE is based on results the authors' previous research on intrusion analysis, detection and diagnosis. However, there are still issues in the IDLE design that will require further research, for example:

- How should observable traces be represented to efficiently serve different needs and tools?

- How can we make sure that the format cannot be utilized by an automatic attack tool?

There are also other problems of a political rather than technical kind:

- Should there be a central IDLE data repository? If so:
  - Who can be trusted to handle such a repository?
  - Who has the resources to maintain the repository?
  - What organization is willing to expose itself to the risks of publishing high-quality exploit information, considering the possible legal and public relations problems?
  - What restrictions on access to the data should there be?

We hope that the benefits for the intrusion detection community of having a resource such as IDLE will motivate collaborative efforts to solve the remaining problems.

5 Conclusion

We have presented IDLE which is an effort to create a common format for describing intrusion data for the IDS community, something that is clearly needed and in great demand. We have shown how we can use XML to get built-in support for partial information and incremental population while allowing both human-readable and machine-readable fields to be stored in a platform-independent format. Because there are difficult political problems with a repository, we are currently concentrated on developing the format rather than populating the database.

References

[1] Stefan Axelsson, Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. An approach to UNIX security logging. In *Proceedings of the 21st National Information Systems Security Conference*, pages 62–75, Arlington, Virginia, October 5–8, 1998. National Institute of Standards and Technology/National Computer Security Center.

[2] Matt Bishop. The UC Davis vulnerabilities project, August 26, 1998. *http://seclab.cs.ucdavis.edu/projects/vulnerabilities/*.

[3] Klaus Brunnstein, Simone Fischer-Hübner, and Morton Swimmer. Classification of computer anomalies. In *Proceedings of the 13th National Computer Security Conference*, pages 374–383, Washington, D.C., October 1–4, 1990. National Institute of Standards and Technology/National Computer Security Center.

[4] John D Howard. *An Analysis of Security Incidents On The Internet 1989–1995*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 7, 1997.

[5] Ivan V Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, West Lafayette, Indiana, May 1998.

[6] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[7] Ulf Lindqvist, Ulf Gustafson, and Erland Jonsson. Analysis of selected computer security intrusions: In search of the vulnerability. Technical Report 275, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1996. Presented at NORDSEC – Nordic Workshop on Secure Computer Systems, Göteborg, Sweden, November 7–8, 1996.

[8] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, California, May 4–7, 1997. IEEE Computer Society Press, Los Alamitos, California.

[9] Ulf Lindqvist and Erland Jonsson. A map of security risks associated with using COTS. *Computer*, 31(6):60–66, June 1998.

[10] Ulf Lindqvist, Per Kaijser, and Erland Jonsson. The remedy dimension of vulnerability analysis. In *Proceedings of the 21st National Information Systems Security Conference*, pages 91–98, Arlington, Virginia, October 5–8, 1998. National Institute of Standards and Technology/National Computer Security Center.

[11] Ulf Lindqvist, Tomas Olovsson, and Erland Jonsson. An analysis of a secure system based on trusted components. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 213–223, Gaithersburg, Maryland, June 17–21, 1996. IEEE, Piscataway, New Jersey.

[12] Conny Stefors. Describing computer security intrusions with XML: Document structure and tools. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999. In preparation.

[13] The World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, February 10, 1998. *http://www.w3.org/TR/REC-xml*.

This is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.

SIR WINSTON LEONARD SPENCER CHURCHILL (1874–1965)