Intrusion Tolerant Software Architectures*

Victoria Stavridou, Bruno Dutertre, R. A. Riemenschneider, Hassen Saïdi
System Design Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
{victoria,bruno,rar,saidi}@sdl.sri.com

Abstract

The complexity of the software systems built today virtually guarantees the existence of security vulnerabilities. When the existence of specific vulnerabilities becomes known — typically as a result of detecting a successful attack — intrusion prevention techniques such as firewalls and anti-virus software seek to prevent future attackers from exploiting these vulnerabilities. However, vulnerabilities cannot be totally eliminated, their existence is not always known and preventing mechanisms cannot always be built. Intrusion tolerance *is a new concept, a new design paradigm, and potentially a new capability for dealing with residual security vulnerabilities. In this article we describe our initial exploration of the hypothesis that intrusion tolerance is best designed and enforced at the software architecture level.*

1. Introduction

Our national computing infrastructure is remarkably vulnerable to malicious attacks. It typically takes only an afternoon for a red team to break into the most heavily instrumented and protected experimental systems at DARPA's Technology Integration Center (TIC). The red teams invariably successfully penetrate the systems in spite of firewalls and the most advanced intrusion detection systems in the world, mostly unobserved, sometimes with the intrusion detection monitors watching helplessly from the sidelines. Vulnerabilities abound and chances are that malicious attackers will be able to exploit them just as well as our red teams. We need to provide our systems with built-in protection to assure that the successful attacks that are certain to occur will not completely disable the system and its ability to provide continued service.

A great deal of research has been devoted to techniques for designing and building systems that will not be compromised by intrusion attempts. Some techniques contribute to vulnerability avoidance (e.g., formal and semiformal design methods, software development process control). Others contribute to discovering and eliminating vulnerabilities that find their way into the design (e.g., testing, model checking). Yet others contribute to intrusion blocking (e.g., firewalls, anti-virus software). Nonetheless, inevitably, "intrusions happen". Intrusion tolerance is the last line of defense.

We believe that the fault tolerance research community has a powerful paradigm that can transform the way we defend against malicious attacks by moving the emphasis from detecting that one of a possibly infinite number of attacks is in progress to detecting, diagnosing, and recovering from one of a finite number of deviations in expected system behavior that occurs as a result of a successful attack. Our approach admits the inevitability of penetrations and focuses on building dependable detection, diagnosis, and recovery mechanisms.

Intrusion tolerance is the ability of a system to continue providing (possibly degraded but) adequate service after a penetration. This means that the system is designed so that the damage caused by the intruder is contained and, perhaps, automatically repaired. The premise of intrusion tolerance is that it is preferable to continue system operation after an intrusion, rather than shutting the system down and completly losing the service it provides. The objective of the work described here is the creation of technology that enables the development of systems which are intrusion tolerant *by design and construction*. We view intrusion toler-

* Our research on intrusion tolerant software architectures is supported by DARPA-funded SPAWAR contract N66001-00-C-8001. The prior work on software architecture hierarchy description and evolution was supported by DARPA-funded AFRL contract F30602-97-C-0040. The prior work on the use of architectures to guarantee security properties was supported by DARPA-funded AFRL contract F30602-95-C-0277. Current complementary research on the use of architecture hierarchies to maintain nonfunctional properties of evolving systems at run time is supported by DARPA-funded AFRL contract F30602-00-C-0199.

ance achievement as a proactive design activity, as opposed to a passive, "extra" feature to be added after the system has been developed. Intrusion tolerant systems are *inherently survivable*.

Our research is aimed at elaborating and providing a sound theoretical foundation for intrusion tolerance techniques. The key idea is, having detected an intrusion, to have the system act to minimize its effect. First and foremost, intrusion tolerance seeks to maintain at least minimally acceptable performance. Second, it attempts to contain, isolate, and eventually expel the intruder. Third, it logs the details, so that the vulnerability that enabled the intrusion can be subsequently analyzed, and, if possible, eliminated.

Our approach to realizing intrusion tolerance is based on the premise that *a system's architecture can ensure intrusion tolerance* and was inspired by our recent DARPA Information Survivability Program sponsored research that led to the development of provably secure software architectures. A well-designed architecture can guarantee that a system is multilevel secure if its components have much simpler security properties, in that it is possible to formally prove that, if the components have the simpler security properties and they are assembled in accordance with the architectural description, the resulting system is multilevel secure. Whether components have these properties can be determined, to a high degree of assurance, by a combination of analysis (in the case of components being developed specifically for that system) and testing (of "Commercial Off-the-Shelf" (COTS) and legacy components). Analogously, we expect that an intrusion tolerant architecture can guarantee that a system is intrusion tolerant if its components satisfy simpler properties whose presence can be directly verified. Thus, we say that such systems are guaranteed to be intrusion tolerant by design and construction. Our goal is to enable the creation and evolution of intrusion tolerant, distributed, dynamically reconfigurable software architecture hierarchies. We are validating the technology by conducting two proof-of-concept applications: one based on the Genoa crisis management system architecture and one based on SRI's Enclaves™ secure virtual private network architecture.

2. The fundamentals of intrusion tolerance

Despite use of best state-of-practice software engineering techniques, complex systems typically contain numerous vulnerabilities. We believe that designing a system with maximal security assurance requires planned activities structured in an *information vulnerability lifecycle* involving the following stages (see Figure 1):

- *Vulnerability avoidance*, where the objective is to avoid introducing vulnerabilities in the first place by using good development practices
- *Vulnerability removal*, where the objective is to remove vulnerabilities that were introduced in the system despite the avoidance efforts
- *Vulnerability blocking*, where the objective is to stop existing, known vulnerabilities, whose removal is not practical, from being exploited by attackers to gain unauthorized access

Intrusion tolerance is the ultimate line-of-defense and assumes that — despite the avoidance, removal, and blocking efforts — unknown and/or unmitigated vulnerabilities remain in the system. We thus assume that attackers will be able to penetrate the system. The objective of intrusion tolerance is, while maintaining acceptable (but possibly degraded) system service,

- to contain and, ideally, expel the intruder,
- to minimize the impact of the penetration on the integrity of the system,
- to report the vulnerability so that it can either be removed or blocked,
- to collaborate with any available intrusion detection sensors to produce more sensitive diagnosis,
- to provide measured response, and
- eventually, to return the system to full operation.

Some of these intrusion tolerance objectives require intrusion detection and identification. Others require mere recognition that an intrusion has, or may have, occurred. And others can be realized using engineering approaches that do not require intrusion recognition of any sort.

3. Intrusion tolerance compared

Understanding how intrusion tolerance compares with other established information assurance capabilities is needed in order to make optimal assignment of responsibilities to capabilities during system design.

Intrusion prevention for example is a reactive capability. It works by statically deploying firewalls, anti-virus software and other defenses in order to stop the exploitation of known classes of vulnerabilities. Intrusion detection is also a reactive defense capability. It builds mechanisms that detect intrusions and raise alerts. Intrusion tolerance, on the other hand, is a proactive defense capability. It has a design and engineering focus and heavily borrows concepts from the software and safety engineering communities. Unlike prevention and detection, it can potentially be

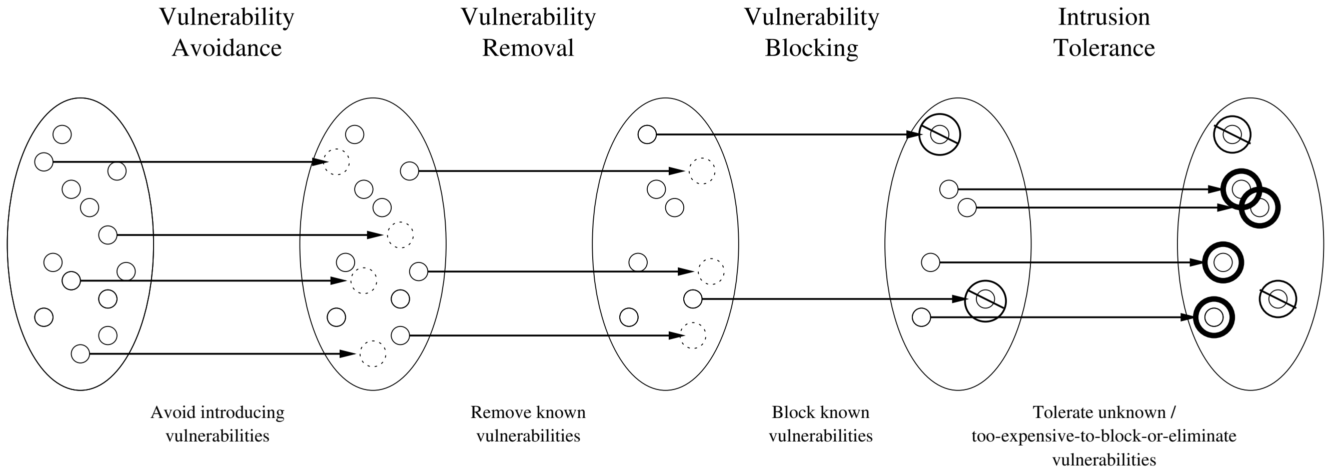

Figure 1. Information Vulnerability Lifecycle

effective against unknown classes of vulnerabilities. Since it is not predicated on recognizing specific attacks, but it rather aims to diagnose deviations from expected behavior, intrusion tolerance can be useful in situations involving previously unknown vulnerabilities as well as novel attacks.

Intrusion tolerance is similar to fault tolerance in that both disciplines aim to have the system continue providing acceptable service in the presence of anomalies. Some of the key fault tolerance concepts and approaches can be used as a basis for developing corresponding intrusion tolerance solutions. *Redundancy* — can be achieved through replication, diversity, and reconfiguration — is one such notion. *Independence* of the redundant components, in order to avoid common mode failures, is another. However, a fundamental premise of fault tolerance research is that the errors that may give rise to failures occur randomly. Attacks on systems on the other hand are intentional, systematic, and repeatable. The difference is crucial. Naïve replication, for example, provides no intrusion tolerance at all, since the intruder can systematically compromise all replicas with the same attack. Neither does simple reconfiguration, which can cause the system to become unstable and continually reconfigure itself in response to numerous similar attacks instead of providing service. Hence standard fault tolerance *techniques* and *algorithms* cannot be directly applied to obtain intrusion tolerance and new solutions are needed. On the other hand, the substantial fault tolerance body of work already in existence can provide useful insights when constructing a conceptual framework within which intrusion tolerance notions and approaches can be articulated, developed, and experimented with.

4. Intrusion tolerance policies and mechanisms

4.1. Policies

Different systems — and even different subsystems of a single system — require different levels of intrusion tolerance. We advocate a risk-based approach that balances protection with component criticality and development cost. The objective is to achieve an optimal intrusion risk level that supports the information assurance needs of the application.

Our approach is inspired by safety engineering, and centers around the definition of Intrusion Tolerance Levels (ITLs). The concept of the ITL is used as an indicator of the required level of protection against intrusion-induced system failures. As a starting point, ITLs will range from I1 to I4, where ITL I4 requires the highest level of assurance. Each system function will be allocated an ITL at the early design phases, and this will be inherited by the components that implement the function. Appropriate design techniques will be associated with each ITL and will guide the development process. In essence the ITLs represent acceptable intrusion tolerance, which may range from preserving confidentiality, integrity and availability for I4 to preserving availability alone for I1.

4.2. Mechanisms

In order to enforce the intrusion tolerance policy we need a variety of mechanisms, including

- *Intrusion detection and location mechanisms*: IDS sensor correlation, Byzantine agreement with authentication algorithms, trigger deployment

- *Intrusion containment mechanisms*: secure compartments, architecture reconfiguration to isolate compromised components (e.g., IDIP/Schnackenberg)

- *Intrusion mitigation mechanisms*: architecture reconfiguration to minimize stolen resources and disable inappropriate information flows

- *Intrusion recovery mechanisms*: extended/distributed recovery regions, architecture reconfiguration (e.g., roll back, roll forward) to eliminate intrusion impact

Mechanisms for redundancy management, tradeoffs, and independence are open issues.

5. Key issues

The key questions we investigating are

- *What is an intrusion tolerant architecture?* We are in the process of describing abstract intrusion tolerance properties at the architectural level. Both these emergent properties (such as secure composition, node data integrity and stability) and the mechanisms that can be used to ensure them (such as non-interference, tampering diagnosis based on proof-carrying integrity marks, and bounded reconfiguration) are being characterized. Our hypothesis is that, just as in our previous work on multi-level security (MLS) properties [12], by matching mechanisms to properties, we will be able to decompose the emergent intrusion tolerance properties into much simpler, directly verifiable, architectural properties.

- *How can intrusion tolerance levels be defined?* Not every system requires the same level of intrusion tolerance. A definition of levels of intrusion tolerance, based on level of acceptable risk and balanced protection, is under development.

- *How can intrusion tolerant architectures be constructed?* The creation and evolution of intrusion tolerant architectures requires a substantial library of refinement patterns that are known to preserve the key intrusion tolerance properties. This, in turn, requires the development of easy-to-apply lightweight techniques for verifying that the patterns preserve intrusion tolerance, and the definition of architectural styles that simplify the task of describing intrusion tolerant architectures. (Proof-carrying architectures [17], where security proofs are produced as a byproduct of the design process, are an example of this sort of lightweight formal method.)

- *What tools can support development of intrusion tolerant architectures?* In particular, what enhancements to out existing SADL and Teal toolkits will provide greater automated support for the specification and analysis of intrusion tolerance properties as well as the construction of intrusion tolerant software components.

- *How can our ideas be applied in development of real systems?* In order to address this question, two experiments are being performed to validate the applicability of these ideas to practical system development.

6. Security in Theory and Practice

Since intrusion tolerance is new concept we need to establish a sound theoretical foundation as a prerequisite to producing techniques for developing and reasoning about intrusion tolerant artifacts. Relating theoretical foundations of security to practical problems has historically been a challenge, because of a dichotomy in approach between the research community and the commercial world. The research community has emphasized mathematical modeling of security properties, pruducing models that have proven difficult to relate to actual systems. Because of this, commercially available systems are (mostly) not formally linked to security theory and it is thus difficult to determine whether they have the desired properties. We propose to bridge this gap by adopting a rigorous approach of relating models to systems. The key idea is replacing the abstract mathematical models by structured architectural models that still support formal analysis, but are both easier to analyze and easier to relate to actual systems.

Software architectures have emerged [19] as the leading conceptual tool for organizing software development processes and products. The architecture of a software system defines that system in terms of computational components and interactions among those components. This high level of abstraction helps bridge the gap between requirements and implementations, and thus enables the forging of concrete links between security models and secure systems. Although architectural structures are abstract in relation to details of the actual computations of the elements, those structures provide a natural framework for understanding emergent, system-level concerns such as intrusion tolerance (as well as other properties including global rates of flow, patterns of communication, and scalability, among others). Thus the software architecture is a particularly suitable level of abstraction for studying intrusion tolerance.

There are many, often subtle, interactions between security, fault tolerance, and intrusion tolerance. Consider, for example, the manipulation of system configuration and exploitation of covert channels arising from redundancy management to effect integrity or confidentiality violations. Unraveling the intricate co-dependence of intrusion tolerance and security requires analysis methods and tools, which is viable only when a formal approach is taken. Furthermore, if we are to avoid having to deal with these interactions at the implementation level, we need to have some way of connecting the architecture to the implementation in a provably correct way.

We are interested in producing useful intrusion tolerance artifacts and usable methods for developing such artifacts. Formal methods have traditionally been seen as impractical and difficult to use. We advocate the integration of *lightweight formal methods* — that is, formal metods that operate "behind the scenes" and that make relatively small demands on system resources – into design time and run time analysis tools, with interfaces and apparent functionality similar to existing tools, which therefore can easily be inserted into the software development and maintenance process. Software architectures support this notion particularly well. Whereas the complexity of reasoning at the implementation level is virtually always prohibitive, in virtue of the formal link between the architecture and the implementation, software architectures allow verification to be performed on the actual artifact[1] as opposed to some abstract representation bearing a tentative relation to it. In addition, the user deals only with architectures and implementations, and thus the formal analysis is transparent. And since architectures support the reuse of design paradigms — secure and intrusion tolerant ones included — they also amortize the verification and reasoning investment across many methodologies and products.

Application of lightweight formal methods to software architectures is especially well suited to the common problem of showing that systems constructed from a combination of new code and legacy code or COTS applications satisfy system requirements. In the architectural approach, new components are treated as "white boxes" in the architecture: their relevant properties are determined analytically, by abstraction, from lower level specifications of their internals. Reused components are treated as "black boxes": the properties that they must have if the system is to satisfy the (non-functional) requirements can be determined by analyzing the abstract architecture, and confidence that the black box components have the required properties can be obtained via unit testing and run time system monitoring. Software architectures provide the building blocks of our approach. In order to manipulate them to produce intrusion tolerance methods and artifacts we have adopted a rigorous approach based on formal methods. Our formal methods fielding strategy is to *design a lot, specify some, and prove just a little*, using lightweight formal methods.

[1] The verification consists of a proof that an abstraction has the desired property plus a proof that the refinement steps linking the abstraction to the actual implementation preserve the desired property. Thus, the verification shows the actual implementation has the property, without requiring reasoning about the actual implementation directly.

7. Software Architectures

Software architectures describe the *components* from which systems are built, *connections* between those components, and *constraints* on observable component and connector behavior and on how components and connectors can be assembled into architectures. Re-applicable patterns that guide composition of components, and constraints on the application of these patterns, have also been defined.

A software architecture description can characterize either a single architecture, or a family of architectures that share common structure. For example, the X/Open Organization has informally defined a standard for distributed transaction processing (DTP). This standard describes a family of architectures that can be constructed from resource managers, an application that accesses them, and a transaction manager. Resource managers include such diverse components as database management systems and print spoolers. Even the number of resource managers is left undetermined by the standard. But all resource managers interact with the transaction manager in accordance with a single protocol. This protocol, together with the protocol for communication between the transaction manager and the application, is the essence of the standard.

Software architectures support composition and evolution at the design level. They characterize families of applications with shared properties and they provide design-level abstractions, patterns and styles [12].

8. Architecture Description Languages

Historically, architecture descriptions have employed a variety of informal and semi-formal, often graphical, notations. These notations typically have no precise semantics, and so do not provide a suitable basis for detailed formal analysis of architectural properties. Even if a semantics were provided, the expressive power of these notations is quite limited.

Recent interest in trustworthy combination of components sparked the development of architectural description languages (ADLs) modeled on programming languages. While these languages often provide graphical representations for input and output, they are also capable of expressing complex constraints on architectures that cannot be naturally represented in a diagram. Moreover, just as in the case of programming languages, ADLs can be provided with a precise formal semantics, which allows formal analysis of the properties of architectures. Examples of ADLs include

- SRI's SADL [13]
- CMU's Wright [1]
- Stanford's Rapide [9]
- Honeywell's MetaH [22]
- CMU/IC's Darwin [10]
- UCI's C2 [20].

Of special note is the Acme ADL interchange language [3], which serves as a least common denominator for ADLs. Each of these ADLs differs from the others in expressive power. For example, our own ADL, called SADL, supports the definition of specific architectures, parameterized families of architectures, and architectural styles (collections of constraints that provide the semantics for the "boxes and arrows"). Our Teal "metaADL" supports definition of mappings between architectures, architectural refinement patterns, and architectural hierarchies.[2] SADL/Teal development is supported by a toolset specifically designed to support definition and analysis of architecture descriptions at a variety of levels of abstraction, in a variety of styles. A simple point-and-click style of GUI allows a system engineer who is not an expert in formal architecture definition to build architectural descriptions and hierarchies in the same intuitive fashion as popular CASE tools. However, internally, SADL represents architectural descriptions as logical theories, and Teal represents patterns as structure-preserving mappings on theories. The combination can thus provide powerful tools for automatically analyzing and maintaining evolving architectural descriptions. On the other hand, toolsets for some other languages, such as C2, provide much more built-in support for defining architectures in a particular style.

9. Architecture Hierarchies

SRI research on software architectures — funded in the past by DARPA ITO under the Evolutionary Design of Complex Systems (EDCS) program and currently by DARPA ITO under the Dynamic Assembly for Software Adaptability, Integrity, and Assurance (DASADA) program — is focussed on bridging the gap between requirements and implementation. Requirements can be directly represented by constraints on an abstract architectural description. This abstract description is the initial term in a series of increasingly concrete descriptions, called an *architecture hierarchy*. Each description in the series is linked to its predecessor by an implementation mapping that can be shown to preserve satisfaction of the requirements. The final term in the series is a description of the architecture described in terms of concrete constructs that can be directly implemented. So verification of the implementation mappings guarantees that the as-implemented architecture satisfies the requirements. Stated another way, a security vulnerability that does not exist in the abstract architecture cannot be introduced into any more concrete architecture in the hierarchy. Breaking the problem of verifying the concrete architecture into manageable chunks makes it solvable using lightweight formal methods.

Architecture hierarchies serve another important function: a hierarchy records a rationalized derivation of the implementation-level architecture from the abstract architecture. The hierarchy may actually have been obtained by abstraction from a concrete description, or some mixture of abstraction and refinement. But, once it has been defined, it can always be thought of as a refinement hierarchy. Hierarchies provide a suitable starting point for exploring the space of alternative implementations. Each implementation mapping can be thought of as capturing a design decision. At any description in the hierarchy, a new branch that leads to an alternative implementation of the abstract architecture can be grafted on by supplying an alternative implementation mapping. So a hierarchy provides a convenient starting point for searching the space of possible implementations of the abstract architecture.

For similar reasons, hierarchies provide a solid foundation for automated rearchitecting support. *Rearchitecting* is the process of modifying an architecture to accommodate changes in requirements or changes in the availability of components and connectors. The key to making rearchitecting more efficient is having access to the design decisions made in implementing the abstract architecture. Case studies suggest that design decisions recorded in the hierarchy frequently can be automatically reapplied after a change has been made to a description in the hierarchy. As a result, a modified implementation-level architecture can be generated largely automatically. This "derivation replay" technology, currently under development at SRI, offers the potential of greatly reducing the maintenance-time burden on the system architect.

10. Secure Software Architectures

Our earlier, DARPA ITO Information Survivability (IS) program-sponsored work on describing secure software architecture hierarchies [12, 17] resulted in a firm foundation on which to erect intrusion tolerant architecture technology. It illustrates the feasibility of guaranteeing a security property – albeit one much simpler than the intrusion tolerance

[2]Most of the capabilities of SADL and Teal were available in a single ADL, also called 'SADL' [13]. The primary difference is that Teal supports architecture descriptions in a number of ADLs, including both (the new) SADL and Acme.

properties under consideration here – though use of an appropriate architecture.

Abstractly, a simple multi-level secure (MLS) access control policy can be represented as a constraint on the system's data flow architecture: *If datum D is passed to component C, then the classification of D must be less than or equal to the clearance of C.* We defined an architecture hierarchy that links a data flow level description of an MLS X/Open's Distributed Transaction Processing (DTP) architecture to an implementation-level description of that architecture. The refinement steps of the hierarchy have been proven to preserve the MLS policy. Therefore, the implementation of the abstract architecture has been shown to be secure — a result that would be difficult to directly verify, given the complexity of the concrete description.

The comparatively light burden of proving that the implementation steps preserve the security policy was further reduced by using pre-verified patterns in developing the hierarchy. In earlier work, we introduced a strong notion of refinement step *correctness* [12] and have developed a stock of refinement patterns that that have been shown to always produce correct implementations, in that strong sense. The essence of our approach is to insist that patterns not only preserve all positive facts about the architecture, but also all negative facts. MLS security is a negative fact, in that it consists of the absence of undesirable data flow paths. If paths that violate the MLS policy have not been included in an architectural description, and a refinement of that description preserves negative facts, then paths that violate the MLS policy cannot have been introduced by the refinement step. Hence, verification of patterns in the library already provided many of the proofs that implementation mappings preserved the MLS policy.

The secure DTP architecture hierarchy illustrates the use of the "white box/black box" strategy for integrating legacy and COTS components. In the description of the standard, the application program is treated as a black box. Its internals are never specified, although its interface is refined. The security of the overall system depends on the application maintaining the MLS policy internally. On the other hand, the transaction manager is broken down into subcomponents during the design process, and its multilevel security is proved from much simpler assumptions about the properties of those subcomponents.

11. Applying Software Architecture Concepts to Intrusion Tolerance

The process of developing secure architectures is similar to, but simpler than, the process of developing intrusion tolerant architectures. Therefore, this earlier work will serve as a guide for the proposed research. The key enabling task is identifying the emergent intrusion tolerance properties that we want a system to exhibit. Such properties will include

- *Non-interference*, that is, the inability of the operation of one component (e.g., a compromised component) to affect the operation of other components
- *Post-intrusion data integrity*, that is, the ability to check and rectify data integrity violations
- *Intrusion containment*, that is, the ability to restrict the sphere of influence of an intruder
- *Intrusion mitigation*, that is, the ability to redirect information flow through uncompromised channels
- *Post-intrusion stability*, that is, the ability to thwart denial-of-service attacks which result from exploitation of the intrusion tolerance mechanisms (for example, by using the same vulnerability repeatedly to cause continuous reconfiguration)
- *Intrusion recovery*, that is the ability to restore data and channel integrity and block the vulnerability that led to the intrusion

In order to check that architectures at various levels in the hierarchy are intrusion tolerant, we need to map these emergent properties to smaller scale, architecture-level requirements. In the case of secure architectures, for example, the overall MLS property was reduced to checking that no inappropriate information flows were introduced during refinement of an abstract MLS architecture. We will achieve this decomposition of the emergent properties into checkable, localized constraints by considering the mechanisms that might be used to implement them. In the case of post-intrusion data integrity, for instance, this might be achieved by considering intrusion diagnosis methods such as Byzantine agreement using authentication algorithms based on proof-carrying integrity marks. Intrusion containment on the other hand, might be provided by reconfiguring the system to cut links to compromised components and by swapping in uncompromised, behaviorally equivalent but structurally diverse components.

Having defined the intrusion tolerance properties and their architecture level counterparts, we will develop the methodology for building architectures and components that are intrusion tolerant by design and construction. The first step will develop a language for specifying intrusion tolerance constraints at an abstract level. These constraints will state the *confidentiality, integrity, and availability* properties that the system must exhibit after an intrusion has occurred. Unlike the case of MLS, the specification of intrusion tolerance properties will not be entirely independent of the specification of system functionality. Nonetheless, intrusion tolerance is, prima facie, an architectural property. The system tolerates an intrusion if the system can perform its functions despite the fact that the intrusion may have compromised certain components and connectors. The details of a component's functionality are only relevant to determining whether compromise of that component can be tolerated. Therefore, the function performed by the component need not be included as part of the description of intrusion tolerance requirements. All that matters is which other components, if any, can substitute services for the compromised component, and which services performed by the component are essential to the system. It follows that a well-designed architecture can guarantee the desired intrusion tolerance properties. The second step will develop an architecture hierarchy that links an abstract architectural description that obviously has desired intrusion tolerance properties to an implementation-level architecture description. The final step is to prove that the implementation mappings of the hierarchy preserve intrusion tolerance. The result of this three-step process will be a collection of specialized architectural methods that can be used to describe an architecture with verified intrusion tolerance properties that can be directly implemented. The methods will include

- Architectural styles supporting intrusion tolerance
- Refinement patterns to capture design rationale and guide the development process
- Intrusion tolerance-preserving transformations, and
- Intrusion tolerance-specific correctness criteria.

Architectures derived using the methodology are guaranteed to be *intrusion tolerant by design and construction*. This is achieved using lightweight formal methods and without post hoc verification of complex software artifacts.

12. Intrusion-Tolerance Ontology

Our first step is to identify basic concepts relevant for defining what it means for an architecture to be intrusion tolerant. Our approach is based on applying dependability and fault-tolerance concepts to the particular case of intrusions. An architectural description of a distributed system defines the system's main components and their interfaces, and specifies how the components are connected and communicate. An architecture description also defines the system's boundary and its interface with the environment. The architecture is then the description of the overall system's organization and the specification of how the system is built out of its components. To discuss intrusion tolerance, we need to represent not only the overall structure of the system but also its externally-observable behavior. Different computational models can be used for this purpose. We chose a simple model based on labeled transition systems. If the visible behavior of each component of a system is given by a transition system then we can obtain the overall system's visible behavior by computing the composition of the component systems under appropriate synchronization. An architecture model may specify some constraints on the behavior of each component — say, that a value consumed at one port will result in a value being produced at another port prior to consumption of another value — which translates into constraints on the acceptable transition systems that can be attached to each component. An architectural description typically assigns a type to the components (e.g., server, client, pipe, filter, store) that implies a specific behavior. The architecture also specifies synchronization constraints that determine how the global system behavior is obtained from the behavior of its components. From this point of view, an architecture description can be seen as a design, in the sense of Lee and Anderson[8]. Given a set of transition systems, each modeling the visible behavior of a component, the architecture defines a global transition system that models the behavior of the system. Once we are given an architectural description and an associated computational model, we can say that the system is *intrusion tolerant* if compromise of a limited number of components does not lead to a global system failure. Although intrusion tolerance can be defined with respect to failures in general, we focus on two particular classes of security-related failures, namely, the loss of integrity and of availability. The following definitions show how the concepts of intrusion, vulnerabilities, attacks, and failures can be precisely characterized.

12.1. Requirements and Failures

The starting point is to assume that a system has to satisfy a set of security requirements that include both integrity requirements and availability requirements. *Integrity* is generally defined as the absence of improper (or illicit) alteration of information. We can then assume that integrity requirements can be formally specified as state invariants: we are given a set I of "good states", and a system that is in a state in I must never transition to a state that is not in I. Availability requirements correspond to readiness for service [7]. In typical distributed applications, the system can be said to be available if requests or actions from the system's user are followed by responses (other than error messages) within an appropriate time interval. As integrity requirements, such availability properties can be formulated in terms of states that should be avoided. Depending on the time limit, we can consider two types of availability properties: *eventual response* and *bounded-time response*. In both cases, we are given a set of "bad states", B, in which the system is not responsive to some or all of its users. Eventual response requires that the system should not remain forever in this set of bad states. Bounded time response specifies for how long the system is allowed to remain in this set of bad states. A *failure* can then be defined as the violation of one or more of these security requirements: an integrity failure occurs when the system reaches a state that does not belong to I; an availability failure occurs if the system stays for too long in state in B.

12.2. Vulnerabilities and Attacks

When discussing intrusions, it is essential to distinguish between accidental faults and deliberate actions from a hostile attacker. Both types of faults can lead to failures, but countermeasures that protect from accidental faults might be ineffective against deliberate attacks. This distinction is especially useful when considering availability requirements. In many cases, violation of an availability requirement cannot be ruled out because any system has limited resources, and performance degradation (including delayed responses or absence of response to some request) unavoidably occur if the system is overloaded. Such a situation must be distinguished from deliberate denial-of-service attacks that attempt to maintain the system in non-responsive states. If the system is in a state q and has not already failed, then a failure may occur if there is a system trajectory or execution that starts in q and leads to a bad state (in the case of integrity failures) or maintains the system in bad states (in the case of availability failures). We call each such possible execution a *vulnerability* of the system. Vulnerabilities are due to design faults. The existence of vulnerabilities does not guarantee that failures will necessarily occur. The actual trajectory followed by the system depends on input from the system's users and possibly on internal choices. Failures can occur by accident. The users may inadvertently produce inputs that globally lead the system to a bad state. On the other hand, an attack is a deliberate attempt by a user (or set of users) to force the system into following a bad trajectory, by producing calculated input at the right time. We can think an attack as a *strategy* that determines the attacker's next input based on what he has observed so far. Whether or not an attack might succeed can depend to a large extent on the environment, including what the innocent users — everyone except the attacker — are doing. This leads to a gradation of attacks. The strongest attacks are those that lead to a sure failure whatever innocent users are doing. Other attacks may lead to failure only under some assumptions about the likely behavior of users or the system.

12.3. Intrusions and Compromised Components

In our architectural framework, we consider intrusions to be component failures. In general, a component *fails* if its behavior does not match what the rest of the system expects, in other words, if the constraints on externally-visible system behavior that are built into the architecture are violated.[3] We define an *intrusion* as an error[4] that results from a successful attack on a system component. In some cases, the intrusion can immediately lead to a system failure. For example, if the compromised component is critical to implementing system services (i.e., the component is a single point of failure). In other cases, evaluating the impact of the intrusion requires examining how compromised components might behave, and possibly affect the rest of the system. As in general fault tolerance, different types of behavior after failure can be considered.

- An attack can render the attacked component unavailable (e.g., crash or a denial-of-service attack). This is similar to a *stopping* or *crash failure*; the component is non-responsive to the rest of the system.
- An attack can result in the attacker controlling how the compromised component behaves. This is similar to a *Byzantine failure*; the component behaves in an almost arbitrary manner. Component behavior may not be completely arbitrary as mechanisms such as encryption may limit what the attacker can do (assuming the encryption algorithms are strong enough). Other limits may exist. For example, the attacker may try to keep the intrusion undetected, which may limit what he can do.
- Other types of attacks may be worth considering. For example, a common form of attacks (e.g., on web servers) attempts to corrupt the content of data-storage components (e.g., a file system). After such an attack, the compromised component does not behave properly, since it returns incorrect data (i.e., data that violates specified architectural constraints on the component's output) in response to requests, but its behavior is not as arbitrary as in the case of Byzantine failures. In particular, the component still behaves in a consistent fashion: the same requests will get the same answers.

12.4. Intrusion Tolerance

Once components are compromised, the system is in an error state, that is, a state that can potentially lead to a failure. The whole purpose of intrusion tolerance is to prevent such errors from resulting in system failures. Obviously, intrusion tolerance cannot be absolute but must be defined with respect to a *fault model*, that is, the specification of

[3] As noted above, these constraints are typically much weaker than a full functional specification of the component. Very simple constraints are adequate to identify non-responsive components and grossly corrupted data.

[4] In the dependability terminology, an *error* is defined as that part of the system state that is liable to lead to a subsequent failure [7].

the nature and number of compromises that the system is required to tolerate.

Clearly, intrusion tolerance requires redundancy and diversity. Redundancy prevents the compromise of a single component from leading to an immediate system failure. Diversity is required to reduce the risk of common-mode failures; we must make sure that the same attack cannot compromise several components. Another aspect of intrusion tolerance concerns is the recovery from intrusions, which requires detecting that a component has been compromised, isolating the component to avoid further damage, repairing the component, and restoring the system. In general, many techniques designed for fault tolerance are applicable in this context. System reconfiguration plays an essential role. As a common consequence of intrusions, the attacker may increase his capability to mount new attacks on the rest of the system. Access to a component may give the attacker access to valuable information such as cryptographic keys or passwords, allow the attacker to observe traffic that is not visible from outside the system, and enable new attacks that are not possible from outside. Modeling attacks that can be launched from a compromised component can rely on the game-theoretic principles discussed previously. An important part of intrusion tolerance is to make sure that the resources and information that can be gained from compromised components do not enable easy attacks on other component targets.

13. Case Studies

13.1. Enclaves

Enclaves^{TM} [4] is a lightweight software framework for building secure group applications that can be deployed in insecure networks such as the Internet. Enclaves is implemented as middleware that provides secure communication and group-management services to a group-oriented application. The group is organized around a central group leader that starts and ends the application, and is responsible for all group-management activities, including user authentication, generation and distribution of cryptographic keys, and distribution of group-membership information. Regular group members establish a bi-directional communication channel with the leader when they join the application.

The current Enclaves implementation is based on the assumption that the leader and all group members (past and present) are trustworthy. However, many group applications (e.g., in electronic commerce), involve a set of users who collaborate on a common task without fully trusting each other. Even if all members are trusted a priori, intrusions at a user's host machine can compromise security. Similarly, compromising the leader can lead to violation to confidentiality, integrity, and availability requirements. In other words, the current Enclaves implementation is not intrusion tolerant. We are currently examining a new architecture and a new set of cryptographic protocols that increase the resilience of Enclaves to intrusions or failures of the group participants (i.e., the members and the leader).

As a first step, we designed a new set of protocols for implementing group-management services in Enclaves. These protocols still rely on a central leader but have been shown to tolerate an arbitrary number of misbehaving members. These protocols ensure proper user authentication: if user A and the leader are trustworthy, and the leader accepts A as a group member, then A actually requested to join the group. Once A is in the group, the protocols ensure proper distribution of group-management messages: All the group-management messages accepted by A were actually sent by the leader; they are accepted by A in the same order as they were sent by the leader; no group-management message accepted by A is a duplicate. These properties hold under the assumptions that attackers or compromised members cannot break the encryption mechanisms used.

These more robust protocols tolerate misbehaving members and are resilient to outsider attacks, but the central group leader remains a single point of failure. We are currently designing a more scalable and intrusion-tolerant Enclaves that does not rely on a single leader, but uses a distributed set of servers to implement the same services as the leader currently does. Intrusion tolerance will rely on servers monitoring each other and dynamically reconfiguring the system as member or server failures are detected.

13.2. Genoa

Project Genoa is a DARPA ISO program that is developing a prototype decision-support environment for agencies involved in crisis prediction, analysis, and management (`http://www.darpa.mil/iso/project_genoa/Project_Genoa_White_paper.html`). The Genoa system consists of a collection of clusters, each of which includes a set of tools for gathering information from heterogeneous sources and for building models and structured arguments for assessing and analyzing crisis situations. Genoa also includes visualization, search, and editing tools, and tools for organizing collaboration between analysts. Each cluster is autonomous and typically encompasses a single organization, but separate clusters can be connected to enable information sharing between different organizations. The tools deployed within a Genoa cluster are supported by a set of servers that provides persistent data storage with fined grain access control. This set of servers called CrisisNet is the basic infrastructure that allows Genoa tools to exchange data and collaborate.

To study the practicality of an architecture-based approach to intrusion tolerance, we are studying its application to Genoa's CrisisNet. [5] This research is still underway. Preliminary results include the identification of high-level security risks within a Genoa cluster, taking into account both outsiders and insiders. We are currently developing an intrusion-tolerant architecture for CrisisNet that should ensure integrity and availability of the various documents and information stored.

14. Related Work

Although intrusion tolerance as a concept has not been studied before, some facets of it are already substantially developed. These include the earlier stages of intrusion detection (*Has an intrusion occurred?*) and intrusion location (*In which part of the system did it happen?*) — see, e.g., [15, 16, 18]. By comparison, the later and much more challenging stages of intrusion diagnosis (*How much damage was done?*), intrusion containment (*Can the movements of the intruder be restricted so as to halt the propagation of damage to healthy parts of the system?*) and intrusion recovery (*Can the system be reconfigured to isolate the damaged region and introduce new connectivity that restores information flow to minimize impact on operations?*) are only now being developed [21, 2, 5, 6, 11, 14] and are not well-understood.

15. Summary

It is an unfortunate fact of life that systems, no matter how carefully designed and extensively tested, will always contain unanticipated vulnerabilities. The more complex the system, the more subtle and difficult to eradicate the vulnerabilities — and systems are becoming increasingly complex. Acquisition trends (in particular, the increased use of COTS components) and technological advances (such as mobile code) also exacerbate the problem. Our approach to addressing this problem is to design systems with software architectures that allowed continued (albeit degraded) operation after intrusions and support measured response to detected intrusions.

Intrusion tolerance is an ambitious goal. We will produce usable intrusion tolerance technology and artifacts by exploiting state of the art software engineering technology in the form of software architectures, by learning from safety engineering, by leveraging our achievements in constructing multilevel secure software architecture hierarchies, and by focussing development in the specific context of the Genoa architecture and harnessing the resulting synergies and insights. In our approach, the *assurance is a consequence of the development methodology*. Our rigorous intrusion tolerance development methodology will produce software systems from abstract architectural specifications by a series of intrusion tolerant preserving transformations, thereby guaranteeing maximal intrusion tolerance assurance.

[5] This study is being performed in collaboration with NAI Labs.

References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1995.

[2] P. Ammann, S. Jajodia, C. McCollum, and B. Blaustein. Surviving Information Warfare Attacks on Databases. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

[3] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[4] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, 15(3):567–575, April 1997.

[5] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, volume 3, pages 317–326, Kona, Hawaii, January 1998.

[6] J. Knight, M. Elder, and X. Du. Error Recovery in Critical Infrastructure Systems. In *Computer Security, Dependability and Assurance: From Needs to Solutions*, pages 49–71. IEEE Computer Society, 1999.

[7] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[8] P. Lee and T. Anderson. *Fault Tolerance: Principles and Practice (2nd edition)*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1990.

[9] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, number 989 in Lecture Notes in Computer Science, pages 137–153, Sitges, Portugal, September 1995. Springer-Verlag.

[11] J. Millen. Local Reconfiguration Policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, 1999.

[12] M. Moriconi, X. Qian, R. Riemenschneider, and L. Gong. Secure Software Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93, Oakland, CA, May 1997.

[13] M. Moriconi and R. Riemenschneider. Introduction to SADL 1.0. Technical Report SRI-CSL-97-01, SRI International, Computer Science Laboratory, March 1997.

[14] P. Neumann. Practical Architectures for Survivable Systems and Networks: Phase-One Final Report. Technical report, Computer Science Laboratory, SRI International, Menlo

Park, CA, January 1999. Available at `http://www.csl.sri.com/~neumann/arl-one.ps`.

[15] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, MD, October 1997.

[16] P. Porras and A. Valdes. Live Traffic Analysis of TCP/IP Gateways. In *Proceedigns of the 1998 ISOC Symposium on Network and Distributed System Security (NDSS'98)*, San Diego, CA, March 1998.

[17] R. Riemenschneider. Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures. In P. Donahoe, editor, *Software Architecture*. Kluwer Academic Press, 1999.

[18] J. Ryan, M.-J. Lin, and R. Miikkulainen. Intrusion Detection with Neural Networks. In *Advances in Neural Information Processing Systems*, Denver, CO, 1997.

[19] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 307–323. Springer-Verlag, 1996.

[20] R. Taylor, N. Medvidovic, K. Anderson, J. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[21] E. Totel, J.-P. Blanquart, Y. Deswarte, and D. Powell. Supporting Multiple Levels of Criticality. In *FTCS-28, The 28th Annual Fault Tolerant Computing Symposium*, pages 70–79, Munich, Germany, June 1998.

[22] S. Vestal and P. Binns. Scheduling and Communication in MetaH. In *Proceedings of the Real-Time Systems Symposium*, pages 194–200, Raleigh-Durham, NC, December 1993.