A Comparison of Alternative Audit Sources for Web Server Attack Detection

Magnus Almgren Erland Jonsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Magnus.Almgren@ce.chalmers.se
Erland.Jonsson@ce.chalmers.se

Ulf Lindqvist
Computer Science Laboratory
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025, USA
Ulf.Lindqvist@sri.com

Abstract

Most intrusion detection systems available today are using a single audit source for detecting all attacks, even though attacks have distinct manifestations in different parts of the system. In this paper we carry out a theoretical investigation of the role of the audit source for the detection capability of the intrusion detection system (IDS). Concentrating on web server attacks, we examine the attack manifestations available to intrusion detection systems at different abstraction layers, including a network-based IDS, an application-based IDS, and finally a host-based IDS.

Our findings include that attacks indeed have different manifestations depending on the audit source used. Some audit sources may lack any manifestation for certain attacks, and, in other cases contain only events that are indirectly connected to the attack in question. This, in turn, affects the reliability of the attack detection if the intrusion detection system uses only a single audit source for collecting security-relevant events. Hence, we conclude that using a multisource detection model increases the probability of detecting a range of attacks directed toward the web server. We also note that this model should account for the detection quality of each attack / audit stream to be able to rank alerts.

Keywords: intrusion detection, attack manifestations

1. Introduction

In the field of intrusion detection, people speak about the importance of using diverse detectors to have better attack detection coverage. It has been claimed (Tombini et al. [23]) that when combining *different* intrusion detection systems (IDSs), one would improve the attack detection of the system as a whole. For example, one can either combine misuse IDSs with anomaly-based IDSs, or a commercial misuse system with an open-source misuse system. Unfortunately, the results have been mixed: deploying some systems together has not increased the diversity but rather just the number of (similar) alerts.

Other researchers have proposed using separate audit sources to collect security-relevant events, thus improving the detection of attacks. Most of these papers focus on a single system, such as an application-based IDS, with a short description of its advantages and disadvantages compared to the ubiquitous network-based IDS. In this paper we take a slightly different approach. We examine what could be gained by combining the alerts from several intrusion detection systems using different audit sources. We do this by examining the role of the audit source to the attack detection capability of the IDS, and whether this role changes based on the attack type.

More specifically, we define a typical misuse IDS and show what type of security-relevant events it can collect from different audit sources: one audit source at a time and one attack at a time. We then discuss how a specific attack class would manifest in a particular audit stream and what implication this manifestation has on the attack detection by the IDS. Thus, we can informally rank the audit sources based on the quality of information they provide to the IDS. Or, if the quality of information is dependent on the type of attack in progress, we can point out which audit source is most suitable for detecting a particular attack class, thus supporting the claim that a multisource detection model does increase the reliability of attack detection for the IDS.

We limit our discussion to attacks directed at web servers and related software. The web server is a complex piece of software, often outside the perimeter defense, and accessible to anyone in the world. To complicate matters, the web server often forwards requests to inside resources (legacy databases) that were never designed with a robust security model. Being the analogy of the front door to a company, numerous attacks have been directed toward web servers and the resources with which they communicate. There also exist open-source web server alternatives that are mature enough to allow direct instrumentation (to collect security-relevant events).

The rest of the paper is organized as follows. In Section 2 we outline our approach. We start with discussing the template IDS used in the paper, and then the three instances of it that collect data from three different audit sources. We describe these audit sources in detail and the type of information available to an IDS using that particular audit source. This section is followed by a description of the attacks we have chosen. To limit the scope of the paper, we concentrate on three representative attacks. The execution of the exploits and the underlying flaws are discussed in Section 2.2. We describe our analysis methodology in Section 3, and in Section 4 we discuss what manifestations we expect to see from the attacks in the audit streams described previously. For each attack, we discuss how well an IDS using a particular audit source would detect the attack. We summarize our findings in Section 5 and discuss related work in Section 6. The paper is concluded in Section 7.

2. Background

To be able to investigate the importance of the audit source to the detection capability of an IDS for various attacks, we need to consider attacks and intrusion detection systems together.

In Section 2.1, we use a taxonomy by Debar et al. [8] to define a (theoretical) intrusion detection system that can use different audit sources. We limit the attack scope by considering only attacks directed at web servers and related software. These attacks are taken from the database described by Ingham et al. [12]. The attacks we have chosen are presented in Section 2.2. In Section 4 we then discuss how these attacks manifest themselves in the audit source used by a particular IDS.

2.1. Description of Intrusion Detection Sensors

When discussing the suitability of an audit source for attack detection, one indirectly uses an IDS as a point of reference; it is the IDS that analyzes information captured from the audit source using the actual detection algorithm. For this reason, we are going to describe three similar IDS that only differ in the audit source data they use for their analysis, but which, in other aspects, are very similar. Using the IDS taxonomy of Debar et al. [8], we concentrate our investigation on the *audit source location* attribute and fix the other attributes found in the taxonomy, resulting in the example shown in Table 1.

In short, we have chosen three misuse systems: one network-based IDS, one application-based IDS, and finally

Detection Method:	knowledge based
Behavior on Detection:	passive alerting
Usage Frequency:	continuous monitoring
Detection Paradigm:	transition based
Audit Source Location:	host log files | application log files | network packets

Table 1. The example IDS used in the paper, categorized according to Debar et al. [8].

a host-based IDS (using a subset of system calls as its audit source). We focus on misuse systems, as these are easier to reason about than anomaly-based systems. The latter require a normal profile that may change depending on the environment.

For the purpose of our analysis, we assume a single misuse-based detection engine (our template IDS), which can take input from either one of the different audit sources we consider. To simplify the presentation, we refer to our template IDS connected to a network packet audit source—including preprocessors and other support components—as a network-based IDS. We use the terms *application-based IDS* and *host-based IDS* in a similar fashion.

2.1.1. Network-based IDS (NIDS) We use Snort as an example of a network-based IDS, i.e., the audit source location attribute is “network packets” in the Debar [8] taxonomy. The core function of Snort is in its rules, even though a number of preprocessors and protocol analyzers are critical to the function of Snort and its ability to alert for certain events. A Snort rule is shown in Figure 1. This rule attempts to detect any access to the external cgi-program phf, infamous for its security weakness [11].

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-CGI phf access"; flow:to_server,established;
uricontent:"/phf"; nocase; reference:arachnids,128;
reference:bugtraq,629; reference:cve,1999-0067;
classtype:web-application-activity; sid:886; rev:11;)
```

Figure 1. A typical Snort rule.

A Snort rule contains a header and a rule option (the part within parentheses). A detailed explanation of Snort and its rules can be found in [6]. In short, the Snort rule shown in Figure 1 triggers for any request to the web server that contains the string /phf. Before the pattern match, several plugins have preprocessed the data, such as reassembling the TCP stream and normalizing the HTTP request. In previous versions of Snort, there was only an option to match on the data content after TCP/IP reassembly. However, this left an opening for attackers to evade the detection, so newer versions of Snort include the option to match on uricontent, where the content has been processed further, including collapsing “xxx/../” into “/” and decoding any %xx-encoded characters [10]. The hope is to make it easier to write new rules to detect web attacks with such an option, and more difficult to evade detection from such rules. We will discuss this rule further in Section 4.

2.1.2. Application-based IDS (appIDS) A network-based IDS monitors the traffic from the point of view of the network. However, one can also directly monitor the application in question, thus using an IDS with “application logs” as the value of the attribute audit source location. Either one can use the logs written by the application for analysis [2], or the monitoring can be instrumented directly into the application [3]. The latter paper describes several advantages with pulling the information directly from within the application, such as having a richer environment, and being more resistant to any evasion attempts by the attacker.

Instrumenting the application directly or using the information found in the logs are from the audit source view quite similar: both approaches use data from the application directly instead of collecting, from the network, raw data that needs processing before a proper analysis can be done. Using the log can be seen as a subset of direct instrumentation, as the information available in the log is also available to the built-in monitor at the “logging stage.” However, the built-in module has access to more information and if one would like more options to react to a request, i.e., letting the attribute *behavior on detection* from the Debar taxonomy be more than just *passive alerting*, a module is able to direct the behavior of the application based on its analysis. Nevertheless, using the data from log files might still be preferable under certain conditions: direct instrumentation is more expensive and more intrusive, and logs from many applications can be sent to a single source for wide-spread monitoring (compare syslog). Unfortunately, if the application crashes before the final logging state, there will be no record to analyze. The Debar taxonomy makes no difference between these two modes and seems to only have considered using the (external) log files. We base this on the comment found in Section 6.3, saying that attacks can be detected only when the application log is written.

The appIDS we use for discussion in this paper is similar to the system described by Almgren and Lindqvist [3]. We take the data from an instrumented module within the web server and consider how the attacks manifest there. The information available to an IDS module within Apache is mostly found in variables concerning the request, which includes the client address, the URI submitted, and so on. The whole structure can be found in the source code or from [21], but below are three variables shown as an example.

```
char* request_rec::unparsed_uri,
char* request_rec::uri,
char* request_rec::filename
```

One can either analyze the original data or analyze the parsed URI. Notice how these variables correspond to the rule options content and uricontent discussed in the previous section for Snort, but with the notable exception that the module is less dependent on preprocessing steps. Reusing the knowledge of the Snort rule shown in Figure 1, we would match phf to the string uri. However, we can go one step further and exert even more control over the analysis. To avoid false alarms and evasion attempts, we can match directly on the name of the file that corresponds to this response: the filename variable.

2.1.3. Host-based IDS (HIDS) Finally, we consider a system using host log files as its audit source location. There are different types of host-based IDSs in the literature, using different detection methods and audit sources (within the group of host log files). For example, Lindqvist et al. [18] use the BSM auditing features of Solaris as input to a forward-reasoning expert system. Forrest et al. [9] use the sequence of system calls to find anomalous behavior. An aggregated log source found on UNIX computers is the syslog. Many programs can send their log messages to this service where they are aggregated into a single source. Vigna et al. [25] use this audit source for their IDS logSTAT.

In this paper, we consider “lightweight logging” as defined by Axelsson et al. [5]. By logging each invocation of the exec(2) system call, including the arguments, one can trace most intrusions according to [5]. The HIDS we consider in this paper uses the information gained from the exec(2) system call, coupled with a pattern matching module that is similar to the Snort “content” option described above. A typical invocation of execve, a front end to exec(2), looks like the following:

```
execve("/usr/bin/perl",
 ["perl", "./checkPage.pl"],
 [/* 97 vars */]
 ) = 0
```

In this example, the program perl is run with the arguments “./checkPage.pl,” i.e., the perl script. The environment variables are suppressed in this printout (97 vars) but can be included. We choose this particular approach, as it matches the principle of the other two template IDSs described above. Collecting more data, such as the sequence of system calls, would have been beneficial for the analysis. However, we only found anomaly detection systems doing such analysis because it requires intimate knowledge of both the normal and attack behavior. Having the classification attribute *detection method* fixed to *knowledge-based* in Table 1, we forgo such deep analysis.

2.2. Attack Descriptions

The attack database described by Ingham [12] contains a total of 65 attacks, of which some are targeting the same underlying vulnerability or a similar vulnerability. To limit the scope of the paper, we choose three representative attacks for a more thorough examination. Here we outline our reasoning behind choosing these particular attacks and then describe the attacks and the underlying flaws they exploit in detail.

As with audit sources for IDS, we would like to base our selection of attacks on well-founded categories within a taxonomy. However, in surveying available taxonomies we did not find any that suited our purpose. We are interested in the type of manifestation an attack leaves in different audit sources, but most attack taxonomies concentrate on the effect of the attack, e.g., user to root. One notable exception is the defense-centric taxonomy by Killourhy et al. [13], but they concentrate on a single audit source for an anomaly detection system, and their taxonomy is thus not applicable here. In the attack database [12], each attack has informally been described by five attributes.

- Against, i.e., which software / platform is vulnerable
- IDs, i.e., the different IDs the attack / vulnerability has in the community
- Category, i.e., the type of attack (see below).
- Effect, what can the attacker achieve. The categories include DoS and file disclosure.
- Source, i.e., how knowledge of the attack was recorded: Bugtraq, captured with Snort, packetstorm, and so on.

For our purposes the category attribute is the most interesting. The nine different labels are shown below. The value in parentheses is how many attacks in the database have this label.

- Input validation error (22)
- URL decoding error (22)
- Buffer overflow (10)
- Signed interpretation of unsigned value (4)
- Failure to handle exceptional conditions (2)
- Poor resource management (2)
- File disclosure (1)
- Information leak (1)
- Poor memory management (1)

Note that some of these labels are, from the point of view of other taxonomies, subsets of another label. For example, the class *input validation error* usually contains both *URL decoding error*, *signed interpretation of unsigned value*, and *buffer overflow* (see for example [24]). Likewise, *poor resource management* usually contains *poor memory management*. The class *file disclosure*, on the other hand, seems to be more of an effect than a category and looking closer at this particular attack in question, we would have categorized it as an *input validation error* (or a *URL decoding error*).

We initially considered using one attack from each category above, but to limit the scope of the paper we then decided to concentrate on only three attack groups. These groups were selected based on our perceived notion of the importance of the attacks within the different categories. We also accounted for the ubiquitousness of the attacks, partly supported by the number of attack instances found in the database for each category. Thus, we decided to analyze one attack each from the classes *input validation error*, *URL decoding error*, and *buffer overflow*. Having fixed the types of attack, we decided to choose attack instances that would exploit flaws located in different parts of the system, such as within the web server or in an external cgi program. Targeting flaws located at different parts of the system should influence the way attacks manifest themselves in events from a particular audit source. In Table 2, we show the attacks studied in this paper.

In the next three sections, we go through each attack in detail and describe how the attack is executed and what vulnerability it targets. Even though we use a single attack for illustration, we point out that many other attacks within each of these attack category groups share similar traits with the attack we have chosen.

2.2.1. Attack Category 1: Input Validation Error The most prevalent attack category group is the one named input validation error. The attacks numbered 27–32, 34–39, 41, 42, 49, 51–56, 59 have been categorized into this category. We examine attack 35 in detail.

Description:*(The phf attack)* Early popular web servers contained a library routine called `escape_shell_cmd()` that tried to prevent exploitation of shell-based calls, such as `popen()`, by removing characters with special meaning to the shell. Ironically, an early version of the routine contained a flaw and did not properly account for the newline character. The cgi-program phf, included with these servers, used this routine and was therefore vulnerable [11]. The Nimda attack variant shown in the database is

```
GET /cgi-bin/phf?Qalias=x
%0acat%20/etc/passwd HTTP/1.0
```

Note the hex encoding of the newline character %0a (see RFC-2396, Section 2.4). Here, the phf program is run with a query followed by a request to list the contents of the password file. More specifically, the web server accepts the request above and sets the environment variable QUERY_STRING to the characters after the question mark (excluding the HTTP/1.0). The web server then passes control to the external program phf, which in turn performs certain checks on the data passed

	Attack 35: the phf attack	Attack 1: the MS IIS %xx attack	Attack 25: the BO attack
Against	Apache, NCSA on Unix	IIS on Windows	Netscape FastTrack 2.01a on SCO UnixWare
IDs	OSVDB: 136; Bugtraq: 629; CVE-1999-0067; LincolnLabs:1999-Phf	MS01-020; Bugtraq: 2708; CVE-2001-0333; CERT-Vuln: 111677; [...]	Bugtraq: 908; CVE-1999-0744
Category	Input validation error	URL decoding error	Buffer overflow
Effect	Unauthorized file access	Remote access	Remote access
Source	Packetstorm	Captured with Snort	Packetstorm
Target	external (cgi) program	web server	web server

Table 2. Descriptions of the attacks used in the paper. The Target attribute is not part of the attack database description [12].

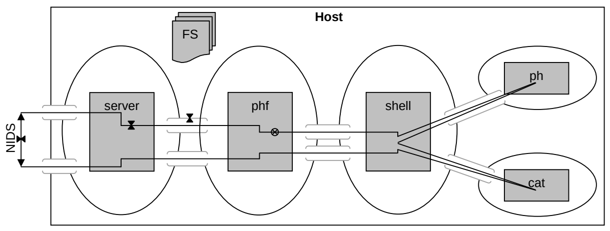

Figure 2. The chain of programs and the interfaces between them and the host OS for the phf request, where the triangles are available audit sources and the circle a flaw.

through the environment variable, and then calls the function `escape_shell_cmd()` to ensure that there is no character within the input that has special meaning to the shell. This is the step that fails, as the routine does not account for the newline character; phf then calls the function `popen()` to pass control to /bin/sh. The shell runs the local program ph with the query. The shell interprets the newline character as a command separator and thus the request to list the contents of the password file is also executed. The output from both these commands is passed back to phf, and phf formats this output into a web page before giving back control to the web server. The web server finally returns the created web page to the client (attacker). This sequence is depicted in Figure 2.

The attacker is exploiting a flaw located outside the web server in this case, but the data is passed through the web server and it is the server that starts executing the chain of programs necessary to complete the request. This is a good example to illustrate the complexity of certain web requests. The exploit is targeting the shell's special interpretation of certain characters, but the flaw is located in phf, as it does not properly filter the input before passing it to the shell. Thus, the input validation error is from within this cgi program.

2.2.2. Attack Category 2: URL Decoding Error The next attack category we consider is the URL decoding error. In the attack database, the attacks numbered 1–18, 43, 44, 57, 58 are in this category. The first 18 attacks belong to Nimda variants, and most of them are quite similar. We concentrate on attack 1 here.

Description: *(The MS IIS %xx attack)* In RFC-2396 (Section 2.4) it is described that characters in a URI may be escaped in certain cases. The character is then encoded into a triplet consisting of "%" followed by two hexadecimal digits representing the ASCII code for the encoded character in question. The RFC also includes a sentence warning implementers to unescape a URI only once, as one otherwise might have a misrepresented string.

The flaw in the web server Internet Information Services (IIS), according to Hernan [10], is that IIS under certain circumstances decodes the URI twice. It first decodes the path, followed by a check of whether the server is authorized accessing the resource in question. When the request concerns a cgi program, the argument to the program is decoded in a second phase. By mistake, the path also is decoded at this time, resulting in it being decoded twice. The cgi program that the finally decoded URI represents is then run. Thus, the file access control mechanism checks the permissions for one resource while another resource is finally executed (the twice-decoded resource). Attack 1 consists of the following request.

```
GET /scripts/..%%35%63../winnt/system32/
cmd.exe?/c+dir HTTP/1.0
```

The attacker is trying to run the command interpreter cmd.exe and list the contents of a directory. Under no condition should an interpreter be run through the web server using arbitrary arguments from the user. In normal circumstances, with a correctly configured web server and with proper file permissions on the file cmd.exe, all such accesses should fail. However, the attacker takes advantage of the decoding flaw described above. In the first step, %35 is decoded into "5" and %63 to "c." The file

`/scripts/..%5c../winnt/system32/cmd.exe` is used when checking the file permissions. It should not exist or give an error. In the second decoding step, "%5c" is decoded into "\" resulting in `/scripts/..\../winnt/system32/cmd.exe`. The web server then runs the command interpreter cmd.exe.

This attack targets a flaw inside the web server code. Other attacks have targeted similar flaws (see for example [7]). Note the difference between this attack and the phf attack described in Section 2.2.1. Even though both attacks use hex encoded characters, the underlying flaws are very different. In this case, the attacker uses the fact that the URI is decoded twice and can thus circumvent file access control mechanisms. In the phf attack, the attacker is using the fact that one character with special meaning to the shell is not properly removed from the input.

2.2.3. Attack Category 3: Buffer Overflow The final attack example we use is a buffer overflow. Examples of such attacks have included the infamous Code Red attack. Attacks 19, 24–26, 40, 45–48, 50 are categorized as buffer overflows in the attack database. We concentrate on attack 25 here.

Description: *(The BO attack)* By sending more than 367 characters in a GET request to the web server, one can overflow the stack. By carefully crafting the request, arbitrary code can be executed with the privileges of the web server.

```
GET /AAAAAA...
```

In the database, a sample attack program sends a series of NOP instructions followed by buffer overflow code. This attack probably directly targets a routine within the web server. Compare this to the Code Red attack (attack 24) that is directed to a flaw in an extension. These two attacks thus target two different parts of the system, a point that is important when discussing their manifestations.

3. Analysis Methodology

The discussion of the manifestations the attacks leave within the different audit streams is done from a theoretical point of view. We have not run these particular attacks and captured their traces for each audit source discussed in this paper. We base the discussion on our previous experience with building intrusion detection systems and we consider the following points.

- Is any attack manifestation found in this audit stream?
  - How much preprocessing is necessary before the data can be extracted reliably?
  - How sensitive does this preprocessing step seem to be to evasion attacks? A typical example is described by Ptacek and Newsham [20] but other schemes are also possible.
- How would one detect this attack, considering the misuse IDS we use?
  - How sensitive is it to false positives, false negatives?
  - Can one easily detect variants of the attack exploiting the same or similar vulnerabilities?
- Can the audit source be enriched with other types of knowledge that help in attack detection (for example, having access to the file system)?
  - Is this type of knowledge naturally found in relation to this audit source? For example, illegal file accesses are more naturally expressed in an HIDS than in an NIDS.

We try to determine if the information available in the audit source coupled with the knowledge an IDS using such an audit source should have results in a satisfactory method of detecting this particular attack and, where appropriate, any attack directed at the specific vulnerability in question.

We can formalize the previous discussion within a formal framework. We investigate how the audit source influences the detection capabilities of the IDS, if at all. We do this qualitatively, by, from a theoretical standpoint, comparing how well an IDS can detect a particular attack by using different audit sources. Let $\mathcal{I}$ be the set of possible intrusion detection systems (excluding the audit source), $\mathcal{A}$ be the set of possible attacks, and finally $\mathcal{S}$ be the set of possible audit sources. As described in previous sections, we have the following.

$$\left\{\begin{array}{lcl} \mathcal{I} & = & \{\text{misuse system with pattern matching}\} \\ \mathcal{A} & = & \{\text{phf attack}, \text{\%xx attack}, \text{BO attack}\} \\ \mathcal{S} & = & \{\text{network}, \text{application}, \text{exec-calls}\} \end{array}\right.$$

Furthermore, let $I \in \mathcal{I}$, $a \in \mathcal{A}$ and $s \in \mathcal{S}$. We examine the following hypotheses.

H1: The audit source, s, does not influence the attack detection for the IDS, I.

$$P(I \text{ detects } a_i | I \text{ uses } s_k) = P(I \text{ detects } a_i)\ \forall i, k \qquad (1)$$

The first part should be read "The probability that the IDS I detects the attack a_i, given that the IDS I uses the audit stream s_k for collecting security-relevant events." As we use only a single IDS in this paper (I is constant), we omit I in the rest of the paper.

H2: One audit source is better than all others for allowing an IDS to detect attacks. If so, the following holds.

$$\exists k \text{ s.t. } P(a_i|s_k) > P(a_i|s_l)\ \forall i, l, l \neq k \qquad (2)$$

Note that even if this is true, combining events from several audit sources may still increase the probability of detecting a particular attack (complementary information).

H3: There exists a pair of audit streams, where one is better than the other for one attack but not for another.

$$\exists i, j, k, l \text{ s.t.} \begin{cases} P(a_i|s_k) > P(a_i|s_l) \\ P(a_j|s_k) < P(a_j|s_l) \end{cases} \quad (3)$$

If this is the case, a particular audit source is better than another for certain attacks *only*, i.e., the suitability of the audit source for IDS detection is attack dependent. This is interesting from a correlation point of view. When collecting alerts from several IDSs using these two audit sources, one must then account for the source when deciding the priority of the final alert or when solving conflicts, such as when the NIDS claims attack, while the appIDS is silent.

If H1 is true, the remaining hypotheses are trivial. In the next section, we qualitatively compare the suitability of each audit source for each attack. More specifically, for each attack, a_i, we informally try to rank the quality of detection by pointing out if we consider one audit source to provide better events for attack detection than another one: $P(a_i|s_k) > P(a_i|s_l)$. Using these informal rankings, if any exists, we then consider the truth of the hypotheses above in Section 5.1.

4. Attack Analysis: Manifestations of Attacks

We go through the attacks we described in the previous section, and consider how these attacks theoretically manifest in different audit streams. We then consider if the IDS, using that particular audit source, would be able to detect the attack reliably.

4.1. Attack Category 1: Input Validation Error

The first attack we described in Section 2.2 is the phf attack. For the attack to be successful, the URI must contain two parts: (1) a request to run the vulnerable program phf and (2) the encoded newline using hex encodings to exploit the flaw. An arbitrary command can then be run.

4.1.1. NIDS An NIDS would have several problems with attacks directed at the web server and cgi programs. First, the NIDS must be placed in such a way that it can listen to and analyze traffic to the web server. Switched high-speed networks have made it more difficult for the NIDS to perform its analysis fast enough to keep up with the network speed. However, this problem and possible solutions are described elsewhere [14]. Here, we assume the NIDS can capture the traffic and not drop packets.

Even though the attack is manifested in the network packets, an extensive preprocessing step must happen before a reliable match can be made. IP defragmentation and TCP stream reassembly should take place before any content match. For certain early IDSs, this was not done correctly and the attacker could then easily circumvent the detection by, for example, dividing the attack into two packets. Such circumvention techniques are in special cases still possible [20] but network-based IDSs are much more resistant today. As the attack is sent over HTTP, the NIDS also needs to account for this fact. In Snort, the preprocessor *HTTP Inspect* takes care of decoding HTTP-specific encodings among other things.

After these preprocessing steps, one can try to match on the content of the stream. Several Snort rules may be applicable to the detection of the phf attack.

- **Rule 886** detects the string "/phf" within the URI regardless of its location (see Figure 1).
- **Rule 1762** detects an attack variant where the URI contains "/phf" combined with the hex encoding "%0a." However, the rule also requires the string "QALIAS" to be present in an *unencoded* form. This means that the attacker can circumvent this rule by, for example, encoding Q into its equivalent hex encoding.
- **Rule 1882** detects output from the id command being returned to the client. If the phf attack ran the command id (to check the permissions under which user cgi programs are launched), this rule would then possibly alert. Other similar rules detect the output from other common UNIX utilities.

The last rule, rule 1882, does not apply to our attack example as the attacker directly tries to access the contents of the password file. Rule 1762 is better but can unfortunately be circumvented (creating false negatives). Finally, rule 886 does alert for any URI containing the string "/phf," one of the two necessary components that must be present in the URI for the attack to succeed. However, this rule runs the risk of creating false positives in that it alerts for every URI that contains the string "/phf," regardless of it being a cgi program to be run or just part of the data sent from the client to the server. More specifically, this rule does not differentiate between an attempted access to the cgi-program phf:

```
GET  /cgi-bin/phf?Qalias=...
```

and a database query to pull up the details of the attack:

```
GET  /swsearch?sbm=%2F&metaname=alldoc&
     query=%2Fphf&x=0&y=0
```

directed at search.securityfocus.com. Thus, the rule cannot determine whether the web server will interpret the string phf in the context of an external program that should be run, or as an argument given to another external program (here swsearch). Furthermore, the NIDS, using this rule, can also not determine

- whether the resource exists
- whether the web server can successfully access the resource
- whether the attack finally succeeds

Based on the current Snort rules, it seems that a network-based IDS can give some indications of whether an attack against phf is executed, but there may be a significant number of false positives, depending on the local traffic mix. What is worse, one of the rules may lead to false negatives. Looking beyond the currently existing rules and this particular attack, we ask ourselves if a more general rule can be created to detect similar types of attack. The flaw exists in an external program outside the web server, and this resource can have any name and is not governed by any standards. Furthermore, the exact type of vulnerability may vary. Thus, the answer is probably that it is difficult to write any kind of generalized rule, unless we change the detection paradigm (see Kruegel et al. [15]).

4.1.2. appIDS The appIDS has several advantages over the NIDS for detecting this type of attack. There is no need to preprocess the information before the analysis, because the host operating system decodes the data before handing it to the web server process. In fact, the request the appIDS analyzes is based on the actual data used by the web server. For this reason, the risk of being vulnerable to evasion attacks is minimized (see Almgren [3]). Furthermore, the appIDS can enrich the information found in the audit source by the use of system calls to access the file system. Using the example above for the NIDS, an appIDS using data from within the web server has no ambiguity between a request for the external program phf or the program swsearch. The appIDS, being located on the host, can use system calls to verify

- whether a file resource exists
- whether this file can be accessed by the web server
- with what arguments

For this reason, it is easier to use the audit source available to the appIDS to detect this particular attack. Reusing the knowledge of the Snort rules above (but matching on the appropriate variables instead) we can create rules having fewer false positives and false negatives. Furthermore, if the appIDS detects something slightly suspicious in the request it can analyze the resulting response before it is sent to the client. For example, coupling the two Snort rules 886 (detecting phf) and 1882 (detecting output from the utility program *id*) described above is quite easily done for an appIDS, resulting in fewer false alarms from the IDS and a more efficient system. There is no need to analyze the output sent to the client unless there is some evidence of questionable behavior. However, as the flaw is located outside the web server, it is difficult to create rules that would capture unknown (but similar) attacks.

4.1.3. HIDS Finally, we look at what type of manifestations the HIDS sees from the phf attack. The HIDS is analyzing the exec system calls and there would be evidence that the program phf is being requested to run. The input to the program is available in the environment variable *query_string*. Thus, one could have a similar rule as within the appIDS and match on phf and the newline character. However, the audit source available to the HIDS is in many ways inferior to the audit source available to the appIDS for these types of attacks. First, the web server may pass data to the external program through environment variables or through the stream *STDIN*. The latter is not visible to the HIDS. The output from the program is sent over the stream *STDOUT*, and is thus also hidden from the HIDS (contrary to the appIDS). Thus, an appIDS can use the name of the program coupled with the input/output for its analysis. The HIDS may have access only to the name.

Now consider a generalized rule to detect similar attacks directed at other cgi programs. The HIDS we have described in this paper with a simple content matching detection engine has problems detecting variants, unless they are explicitly specified in its knowledge base. However, by extending the HIDS by adding capabilities to keep state, we have a more powerful system. If the HIDS follows each process in detail, capturing all its system calls and the system calls of its subprocesses (keeping state), the HIDS may be able to create an execution chain such as the one shown in Figure 2. Only the top chain is valid for this type of request, and detecting the bottom chain should result in an alert. By including other system calls and their arguments, the HIDS might be able to detect that *cat* uses the system call *open* to read the password file. No chain starting with the web server should end with reading/writing the password file. With such an HIDS, one can specify normal access patterns that are allowed as well as resources that are very sensitive, thus having some protection against similar flaws in other cgi programs. This detection method works only if the attacker gains access to a resource that normally is not used by the program. For example, if the attacker used a flaw to extract credit card numbers from an e-shopping site we would have trouble seeing this fact from chains such as the one in Figure 2. The database with credit card numbers is a resource often used by the web server / cgi program running the e-shopping site, and a chain containing this resource would thus look normal.

4.1.4. Summary For the phf attack, it seems the audit stream available to the appIDS is the best one. The attack manifests itself within this stream and there is only a small risk for evasion attempts. The HIDS might be seen as slightly better than the NIDS to detect this attack as it can verify exactly which file within the file system is being run. However, depending on how the input is passed to the cgi program, the HIDS might be blind to the arguments given to the program. Trying to build a general rule to detect any attack within this group is difficult for all three systems. A powerful HIDS would probably have the best detection. It could detect strange execution patterns but it would not detect the particular attacks directly.

4.2. Attack Category 2: URL Decoding Error

Let us now consider how this common type of attack manifests itself for the three audit sources.

4.2.1. NIDS There are (or rather have been) some regular Snort rules in regard to this exploit.

- **Rule 970** is old and is now located within the file deleted.rules (meaning that it should no longer be used).
- **Rule 3201** alerts for accesses to the file httpodbc.dll, thus having nothing to do with the attack we are discussing here.
- **Rule 1002** detects the pattern "cmd.exe" within a URL. As we wrote in Section 2.2.2, the command interpreter should never be run with unfiltered arguments from the user. Thus, this rule would detect this attack, but also other URLs of which some may be innocuous. It does not detect the exploit directly.

Rule 970 and rule 3201 are the only ones from Snort's rule database with references to CVE-2001-0333 and Bugtraq ID 2708 (see Table 2).

One problem of detecting this attack through a pattern matching rule is that it is trivial to create a range of attack variants: any character in the path can be doubly encoded. Encoding a different character in the path still exploits the flaw but changes the pattern of the attack. Instead, Snort relies on the *HTTP Inspect* preprocessor that normalizes input. It has code that alerts for attacks seemingly exploiting this underlying flaw through the option *double_decode*.

The attack is targeting a flaw in the web server's file access control mechanism. Using the network as its audit source, the NIDS does not have any intrinsic knowledge about the file system on the web server host. The NIDS can try to rebuild the whole transaction, but, in the end, it cannot control whether a certain file is accessed and thus the detection is always going to be indicative but never certain. Generalizing the detection rule to detect any attack within this class is as difficult here as it is for the class of input validation errors described in Section 4.1.1. Unless the flaw is known in detail, the NIDS will have problems detecting it.

4.2.2. appIDS Using similar reasoning as done for the phf attack in Section 4.1.2, one would believe that an appIDS would be successful in detecting the MS ISS %xx attack. The main problem here is that the URI decoding is split into two parts. If the appIDS uses the data after the first step (the same data used for the file access control step), the attack might go undetected. However, by comparing the data after the first decoding step with the data after the final decoding, the appIDS can detect the difference and also alert for the fact that the command interpreter is going to be accessed. In hindsight, knowing the details of the exploit, it is easy to claim that the appIDS should be able to detect this attack. Considering that the web server programmers failed to use the correct data for the file access control mechanisms, it may not be as simple in practice and similar mistakes could have been done for the audit stream extraction.

4.2.3. HIDS The host-based IDS using the data from the exec system calls can easily detect the request to run the command interpreter. For reasons similar to the ones outlined in Section 4.1.3 for the password file, the command interpreter is a sensitive resource and all such invocations should be investigated. The HIDS has no problems with characters being encoded twice, as the argument to the exec call is the actual file that will be executed. However, therein is also the problem. The HIDS can never detect this exploit directly but only the implicit effects of it. The exploit targets a flaw within the web server and this attack does not directly manifest in the audit stream used by the HIDS. There are no hex encoded characters for the HIDS to analyze. Thus, as this audit stream is detached from the web server, it detects possible effects of the attack but not the attack itself directed at the web server.

4.2.4. Summary The attack is accessing a resource normally not available through the web server, by using a flaw within the web server to circumvent the file access control. By the time the attack manifestation reaches the audit stream used by the HIDS, the actual attack is no longer visible but only the attempt to access a sensitive resource. For that reason, the HIDS may alert for a symptom but not directly for the attack. This also implies that the HIDS may be better to detect similar attacks trying to achieve the same result. The appIDS has the best chance of detecting the actual attack, but its success depends on collecting the data correctly from the web server. If the data is pulled before the final decoding step, the attack detection might fail. If the appIDS uses the information found in the logging phase, the attack has already taken place. The attack is manifested in the audit stream used by the NIDS, but the NIDS can only give an indication that the request contains an attack and may suffer from false positives and false negatives.

4.3. Attack Category 3: Buffer Overflow

Unfortunately, a misuse IDS may not be the best system to discover buffer overflows (see Northcutt et al. [19], page 276 and page 279–281). Despite this fact, we consider the quality of the information available from the different audit sources.

4.3.1. NIDS After the common preprocessing steps described above, the NIDS can detect overlong fields within the HTTP request. If the length stands out, the NIDS can alert for the potential buffer overflow. Trying to match a certain exploit is fraught with peril as the attacker quite easily can change the pattern but still achieve the attack.

4.3.2. appIDS A buffer overflow attack usually changes the execution path for the vulnerable software. The attack either crashes the software or forces the software to execute a foreign code sequence. The flaw targeted by attack 25 is most likely located early in the request loop within the web server, meaning that the attack effect (disable the web server) will happen before the instrumented module can extract the attack manifestations from the audit source. Thus, the appIDS may be blind toward attack 25 because the attack code is executed before the module gains control.

If the attacker is targeting a flaw further on in the request loop, such as the Code Red attack (attack 24), the appIDS can extract the data and alert for the attack in a similar fashion to the NIDS described in the previous section.

4.3.3. HIDS The HIDS cannot see any direct signs of this type of buffer overflow attack, because the attack does not directly manifest itself within the exec calls. These overlong strings are targeting a flaw in the web server and its extensions. However, as discussed above, the HIDS may be able to detect indirect effects of this attack, such as any irregular resource being accessed by the web server (by the inserted foreign code). It should also be noted that as shown in [17], an HIDS can be very well suited to detect other types of buffer overflow attacks that are targeting exec calls.

4.3.4. Summary Considering that the attack results in the instrumented module not being run, the appIDS is not suitable for detecting this attack. In this case the NIDS has the most reliable detection of the actual attack. As with the previous attacks described above, the HIDS may detect indirect effects.

5. Discussion

We summarize the observations made in the previous section.

Observation 1 *The closer to the flaw, the easier it is to interpret the events from the audit source.* It seems that it is easier to detect an exploit the closer the audit source is to the flaw in question. The data has been decoded by the routines within the program, meaning less risk of being vulnerable to evasion attempts.

Observation 2 *The closer to the flaw, the higher the risk that the audit source also is affected by the attack.* If the attack is exploiting a flaw before the instrumented module can extract and analyze the data, it will in reality be blind to the attack. Hence, any audit source located close to a flaw may be vulnerable to the very same attack. Both application-based and host-based IDSs can therefore be blinded by attacks that crash the application or operating system or otherwise disable the audit source.

Observation 3 *Enriching the events collected from an audit source with local information leads to better detection.* For example, knowing whether a file exists may influence the priority of an alert. Being able to probe the local environment to verify parts of an attack is something an NIDS usually cannot do, with it not being located on the host. Another way to consider such enrichment is that the IDS is able to query an audit source *actively* [16]. Thus, such an audit source (with local information) can be made available to any IDS, regardless of its other collection methods.

Observation 4 *Each audit source has its advantages and disadvantages.* Network-based IDSs see most attacks toward the host, but have problems with evasion attacks and are blind if the data is encrypted. Application-based IDSs have very specific data, but are blind for attacks targeting other services. The HIDS has the same interface regardless of program, meaning that it can be used to monitor any application. On the other hand, it sometimes lacks data important for the analysis. This leads us to the next observation.

Observation 5 *Using several audit sources to collect security-relevant events should improve the attack detection.* As the audit sources provide such different type of information, an IDS (correlation engine) should aim to use as many as possible. Furthermore, *actively* cooperating could create better detection. As an NIDS first sees the data, it can analyze it and look for buffer overflows. If it is safe, it is passed on to the application. There, the final processed result from the NIDS is compared with the processing steps within the application, and any discrepancy is an evasion attempt. If something seems suspicious, the HIDS collects a full trace of the external program's system calls for analysis; otherwise, only a subset (exec) is collected.

Observation 6 *Using data from system calls is a neutral audit source, which works on any program.* This has been mentioned in other papers and also partly in observation 4 above, but we would like to stress it. An NIDS may be able to collect all information going to a host, but it depends on specialized preprocessors to rebuild sessions. The appIDS takes advantage of the local routines available in the application but if an exploit is targeting another application, also the appIDS needs to be able to interpret the data from the point of view of the other application (through preprocessing). As instrumenting an application is expensive, it will probably not be done for all programs, especially not the small cgi programs that are the most sensitive from a security point of view. The HIDS, on the other hand, uses data from a standardized source (system calls), which does not change on a program basis. Thus, it can analyze any program (but may lack some data as we noted in observation 4).

As can be seen, each audit source comes with its own set of strengths and weaknesses. We were looking for an abstract model of *when* an audit source is suitable to detect an attack, but we found it difficult to make any generalized statements. Using the observations above, we can consider only an *inside – outside* model. Considering Figure 2, if the flaw is located inside the server, an audit source within the server (but not affected by the attack) would probably have the best chance of detecting the attack.

5.1. Theoretical Framework

Let us now consider the hypotheses listed in Section 3.

H1: *The audit source, s, does not influence the attack detection for the IDS, I*. Equation (1) cannot hold, because then $P(a_i|s_k) = P(a_i|s_l) \forall k, l$ holds. With the BO attack, we consider the IDS using network packets to have a better chance of detecting the attack than any of the other two systems, thus disproving equation (1).

H2: *One audit source is better than all others for allowing an IDS to detect attacks*. Equation 2 does not hold for the attacks and audit sources we have examined. The IDS using network packets is better for detecting the BO attack than the other two, but probably worse than the appIDS for detecting the phf attack, considering the NIDS susceptibility to encrypted content and dependency on preprocessors.

H3: *There exists a pair of audit streams, where one is better than the other for one attack but not for another*. Equation 3 is true (see the example given in H2 above).

Thus, given the discussion in Section 4, we have shown that the audit source does influence the attack detection for the IDS. Of the audit sources we examined, no one stands out as a winner, and developers of intrusion detection systems would benefit from using a combination of audit sources. Finally, if one combines the alerts from several detectors (using different audit streams), it is important to know which source should be trusted the most on an attack-per-attack basis.

6. Related Work

We investigate the role the audit source plays for detecting attacks. Similar investigations have been made for another attribute, the detection method. For example, Almgren et al. [4] examine whether an attack class, as defined by a taxonomy, can give any indication of whether a detection algorithm is successful in detecting its members. Tan et al. [22] go through the types of attacks the anomaly-based system *stide* detects. Killourhy et al. [13] extend this work and focus on the manifestations left by an attack and how these can be detected by an (anomaly-based) IDS sensor. Attacks that have similar manifestations are categorized into the same class, with the hope that if a detector can detect one attack in a certain class, the detector can detect all attacks in the class. They only consider a single audit source when discussing detection ability. We consider their work to be complementary to this paper.

Alessandri presents a framework to theoretically decide whether an IDS can detect different attacks [1]. His approach is quite theoretical, and he has not examined the role of the audit source in the same detail as we have.

7. Conclusions and Future Work

By considering three attacks and three different audit sources, we have examined the role of the audit source to the detection capability of a template IDS. For each attack, we have in detail discussed how the attack would manifest, if at all, in events coming from a particular audit source and how this would affect the detection of the attack for the IDS. We found that it may be better to use an audit source collecting events close to the flaw, as this limits evasion attacks and the risk of the attack being encrypted. However, in such a case, there is a risk that also the audit source is affected by the attack in question.

Contrary to the wide-spread dependence on only network-based intrusion detection, we show that using a combination of audit sources would improve the number of attacks that can be detected reliably. However, when combining information from different audit sources, it is important to take the attack into consideration. An IDS using a particular audit source may be better at detecting a certain attack, and when one has conflicting evidence of an attack in progress, such a ranking can help solve the conflict.

In the future, we will examine this latter fact further and consider how such rules can be incorporated into a correlation engine.

References

[1] D. Alessandri. *Attack-Class-Based Analysis of Intrusion Detection Systems*. PhD thesis, School of Computing Science, University of Newcastle upon Tyne, United Kingdom, 2004. `http://homepage.hispeed.ch/alessandri/Alessandri-Thesis.pdf`.

[2] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In G. Tsudik and A. Rubin, editors, *Network and Distributed System Security Symposium (NDSS 2000)*, pages 157–170, San Diego, USA, Feb. 3–4, 2000. Internet Society.

[3] M. Almgren and U. Lindqvist. Application-integrated data collection for security monitoring. In W. Lee, L. Mé, and A. Wespi, editors, *Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 22–36, Davis, California, Oct. 10–12, 2001. Springer-Verlag.

[4] M. Almgren, E. Lundin, and E. Jonsson. Implications of IDS classification on attack detection. In S. J. Knapskog, editor, *Eighth Nordic Workshop on Secure IT Systems (NordSec 2003)*, pages 57–69, Gjøvik, Norway, Oct. 15–17, 2003. Published by the Department of Telematics, Trondheim, Norway.

[5] S. Axelsson, U. Lindqvist, U. Gustafson, and E. Jonsson. An approach to UNIX security logging. In *Proceedings of the 21st National Information Systems Security Conference*, pages 62–75, Arlington, Virginia, Oct. 5–8, 1998. National Institute of Standards and Technology/National Computer Security Center.

[6] B. Caswell, J. Beale, J. C. Foster, and J. Posluns. *Snort 2.0 Intrusion Detection*. Syngress Publishing, Massachusetts, 2003.

[7] CERT/CC. CERT advisory CA-1998-04 microsoft windows-based web servers access via long file names. CERT/CC; Internet, Feb 1998. `http://www.cert.org/advisories/CA-1998-04.html`.

[8] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, Apr. 1999.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. 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.

[10] S. V. Hernan. CERT advisory CA-2001-12 superfluous decoding vulnerability in IIS. CERT/CC; Internet, May 2001. `http://www.cert.org/advisories/CA-2001-12.html`.

[11] S. V. Hernan. 'phf' CGI script fails to guard against newline characters. CERT/CC; Internet, Jan 2001. `http://www.kb.cert.org/vuls/id/20276`.

[12] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning DFA representations of HTTP for protecting web applications. *Comput. Networks*, 51(5):1239–1255, 2007.

[13] K. S. Killourhy, R. A. Maxion, and K. M. C. Tan. A defense-centric taxonomy based on attack manifestations. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[14] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, Oakland, CA, May 2002. IEEE Press.

[15] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and communications security*, pages 251–261, New York, NY, USA, 2003. ACM Press.

[16] U. Lindqvist. The inquisitive sensor: a tactical tool for system survivability. In *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pages C–14 – C–16, Göteborg, Sweden, July 1–4, 2001.

[17] U. Lindqvist and P. 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*, pages 146–161, Oakland, California, May 9–12, 1999.

[18] U. Lindqvist and P. A. Porras. eXpert-BSM: A host-based intrusion detection solution for Sun Solaris. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, pages 240–251, New Orleans, Louisiana, Dec. 10–14, 2001.

[19] S. Northcutt, M. Cooper, K. Fredericks, and M. Fearnow. Intrusion signatures and analysis, 2001.

[20] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[21] L. Stein and D. MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.

[22] K. M. C. Tan and R. A. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 188, Washington, DC, USA, 2002. IEEE Computer Society.

[23] E. Tombini, H. Debar, L. Me, and M. Ducasse. A serial combination of anomaly and misuse IDSes applied to HTTP traffic. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 428–437, Washington, DC, USA, 2004. IEEE Computer Society.

[24] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy*, 3(6):81–84, 2005.

[25] G. Vigna, F. Valeur, and R. A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on foundations of software engineering*, pages 88–97, New York, NY, USA, 2003. ACM Press.