Host-based Intrusion Detection Systems

Pieter de Boer & Martin Pels

Revision 1.10 – February 4, 2005

Abstract

Host-based Intrusion Detection Systems can be used to determine if a system has been compromised and can warn administrators if that happens. We recognize four different methods of host-based intrusion detection:

- Filesystem monitoring.
- Logfile analysis.
- Connection analysis.
- Kernel-based intrusion detection.

Implementations of intrusion detection systems generally use one of these four methods to detect intrusions. We have studied multiple implementations, determined their features, ways of evading their restrictions and ways to prevent evasion. We have also given insight into the reasons why certain systems should or should not be used and to what extent, based on their effectiveness and ease of configuration and maintenance.

Contents

Chapter 1

Introduction

Intrusion Detection Systems (IDSs) are a valuable asset in the security of systems and/or networks. IDSs "attempt to monitor and possibly prevent attempts to intrude into or otherwise compromise your system and network resources"[19]. There are three types of IDSs: Those that monitor the network for malicious traffic (Network-based IDS or NIDS), IDSs that monitor activity on a single host (Host-based IDS or HIDS), and systems that correlate events from different Host- or Network-based IDSs (Distributed IDS or DIDS).

This document is about Host-based Intrusion Detection Systems. Host-Based Intrusion Detection Systems can be divided into four types:

File system monitors Systems checking the integrity of files and directories.

Logfile analysers Systems analysing logfiles for patterns indicating suspicious activity.

Connection analysers Systems that monitor connection attempts to and from a host.

Kernel based IDSs Systems that detect malicious activity on a kernel level.

In each chapter of this document we study one of the above types, and discuss two reference implementations. Goal of this research is to find out if a type of HIDS can be put to use in an organisation, and if the value the HIDS adds to the security of the organisation's systems justifies the cost of installing and maintaining it. In order to achieve this goal we look at four different aspects of each type of HIDS:

- Features
- Easy of installation and maintainability
- Techniques for evading the IDS
- Ways to alter the implementations in order to mitigate the effects of evasion attempts

All implementations mentioned in this document were tested on a PC running FreeBSD 5.3-RELEASE-p4[6], unless mentioned otherwise.

Chapter 2

Filesystem monitoring

The first type of HIDS we look at in this paper is filesystem monitoring. HIDS implementations that use filesystem monitoring regularly compare files on a machine with previously gathered information about these files, such as size, owner, and last modification date. This way, if an attacker gains access to the system and changes files, these changes will be detected.

We discuss several aspects of filesystem monitoring in this chapter. First we look at the different features a filesystem monitor may have. After that we discuss the ease of installation and maintainability. In section 2.3 we look at possibilities for an attacker to avoid being discovered by the filesystem monitor, and in the last section we give a number of suggestions on how to make filesystem monitors more resistant to evasion attempts.

To explain how different programs perform on the above topics we look at two reference implementations. These implementations are the Advanced Intrusion Detection Environment (AIDE)[1], version 0.10 and the FreeBSD implementation of Mtree[12], as shipped with FreeBSD 5.3-RELEASE. The first was chosen because it has the same functionality as the well-known Tripwire[26], while being free. Mtree was chosen because, in contrary to other implementations, it is available on a default BSD installation.

2.1 Features

Filesystem monitors can check files on a large number of different characteristics. The list below shows the checks both tested implementations support:

Permissions Changes in the permissions of a file or directory, and the addition or removal of suid/sgid/sticky bits are detected.

Inode If a file on the system still exists but refers to a different inode this is reported.

Number of links The number of hardlinks to a file's inode can be monitored.

Owner/group If the owner or group of a file or directory is changed this is detected.

Size If a file grows or shrinks in size this is reported.

Directory size Adding or deleting of files in a directory is detected.

Mtime, atime & ctime Both filesystem monitors check for changes in the mtime (last modification time), atime (last access time), and ctime (last time the owner, permissions, etc. where changed) of a file or directory.

Checksums The integrity of a file or directory can be checked using a cryptographic hash. This type of checking is based on the fact that it is very difficult (to near impossible) to change a file's contents without affecting the unique hash of the file.

Both tested implementations support a number of different hashing algorithms for this purpose. The most commonly used algorithms are md5[11] (the de-facto standard), and SHA-1[27] (NIST standard).

Type If, for example, a file is replaced with a directory or device of the same name this is detected.

In addition to the above list, Mtree has a number of additional features. These are:

BSD flags Mtree has support for detecting changes in flags. Flags are used in BSD-based operating systems to set special security permissions on files, such as “append only” or “immutable”.

Links Where for symbolic links AIDE supports the standard list of detection mechanisms mentioned above, Mtree is also able to check if the file a symbolic link points to still exists or changes.

Active changing Apart from the checking features Mtree has a number of options for restoring file properties if they have changed. This makes Mtree a useful tool for more than just intrusion detection. Some of the properties Mtree can restore are directory structures, and access and modification times.

2.2 Configuration & maintainability

To set up a filesystem monitor a number of steps need to be performed. First the files and directories that will be checked need to be specified, together with the checks that are performed on each item. After this the initial database needs to be created. This database is used to compare the filesystem with. Finally the program needs to be set up to perform checking on a regular basis (e.g. daily).

2.2.1 Configuration file

AIDE and Mtree use a different approach when it comes to configuration. AIDE uses a configuration file in which all files and directories that need to be checked are specified, together with the checks that need to be performed. It is also possible to mark a directory so that it is checked recursively, or to exclusively mark a file so that it is omitted. Because AIDE allows the use of regular expressions in the specification configuration is very flexible.

As said, Mtree uses a different approach. It does not use a configuration file to specify which checks are performed on what files and directories. Instead, it requires the user to specify a directory that will be checked recursively, and a number of checks that need to be performed on this directory. To make sure parts of the directory tree are omitted an exclude file can be specified. This file contains a simple list of directories or files.

Because of the more simplified approach that Mtree uses it does not allow for something like omitting an entire directory except one file or subdirectory. This can make configuring this tool quite a hassle.

2.2.2 Checking

Filesystem monitors generally don’t use a realtime approach. They check files on a regular basis. Both tested implementations have a command to compare the filesystem with a database file. This command can be put in a system’s crontab to have it executed regularly (.e.g. daily).

To make sure the results of the checks are E-mailed to an administrator the crontab should include a pipe to an E-mail application. AIDE’s manual page mentions a configuration option to send reports via E-mail directly. However, the version of AIDE that we tested does not yet support this option.

2.2.3 Maintenance

Because filesystems change regularly due to things like installation and removal of applications the database file will soon no longer match the filesystem, although no intrusion occurred. This affects the amount of false positives that the tool generates dramatically. To keep up with these legitimate changes in the filesystem the database needs to be updated regularly. How often this needs to be done depends on how much the system is used, and how strict the monitor is configured. Determining the strictness of this configuration is a trade-off between the importance of catching all attacks, and the cost of extra maintenance and sifting through a lot of false positives. A possible configuration that will probably not generate too many false positives is one that only covers basic system binaries and libraries, because these files rarely change. Updating the database can be done automatically. This is however not recommended[2].

Another part of filesystem monitors that needs maintenance is the configuration file. When new applications get added to the system there may be new files that certain checks should not be performed on (e.g. because their size changes often).

In an ideal situation, updates of the database and configuration file are performed dynamically, based on anomaly detection. This is unfortunately difficult to achieve, since something like the installation of a program can be completely legal in one case, while being an indication of an intrusion in another. We are not aware of any tools that use anomaly based detection for filesystem monitoring at this time.

2.3 Evasion possibilities

An important aspect of an IDS is how it behaves against attempts to evade it. In this paragraph we look at different techniques that an attacker can use to evade detection[8]. We divide these techniques in three categories: general evasion attacks, attacks possible due to implementation bugs, and attacks that require root privileges. The list we provide is probably not comprehensive. Still, it gives a good indication of the options an attacker has. In the next section we discuss ways to prevent implementations against these attacks.

2.3.1 General

There are several things an attacker can do to evade a filesystem monitor. The list below shows those techniques that do not require the existence of an implementation bug, or acquiring root privileges on the target system. These techniques are mostly possible because filesystem monitors do not do their checking in realtime, but usually once a day. The list of general evasion tactics we gathered is as follows:

Use unmonitored directories The contents of directories like **/tmp** tend to change a lot. Because of this they may not be incorporated in a filesystem check. This gives an attacker a good place to store files.

Hide between false positives Installing applications can create a lot of false positives. An attacker can use this knowledge to wait with editing files until he sees an administrator perform a program installation. This way the malicious activity might not be noticed between all the false positives.

Restore mtime and atime Filesystem monitors generally don't work realtime. This gives an attacker the time to restore things and cover up his track. An example of this is restoring the mtime and atime of a file after making changes to it. This can easily be done using the *touch* command. The ctime of a file can not be restored using this tool.

Restore filesize Another example of the above is restoring a file's size after removing data from it. This can be done by padding the file with bogus data.

Remove traces from logfiles The filesystem monitors we studied are limited on at least one topic, which can help an attacker. This topic is the monitoring of logfiles. Since logfiles tend to grow in size a lot, checking them for a change in size or hash is pointless. Because of this, changes made by an attacker will also not be caught by the monitor.

Abuse hash collisions Things get more difficult for an attacker if the filesystem monitor uses hashes to check files. This type of checking can only be circumvented if the attacker is able to change the contents of the file, while keeping the hash the same. To accomplish this, an attacker needs to find a collision in the hashing algorithm. Although theoretically possible, this is generally regarded infeasible at the time of writing. This may however change in the future.

2.3.2 Implementation bugs

Every application has bugs, including the tools we discuss in this chapter. Sometimes these bugs can be used against the system. The list below describes a number of (mostly) hypothetical implementation bugs that could help an attacker evade a filesystem monitor:

Obscure filename handling The first bug we discuss in this list is one that actually existed in older versions of Mtree[3]. Because of a software bug filenames with the # character in them where treated as comments and ignored. This bug was fixed in the version of Mtree we tested. We tried out above, and various other obscure characters in filenames to find out if AIDE or Mtree have problems dealing with them, but did not find any problems.

Long files and deep trees Another issue we tested both implementations on is how they deal with very long filenames and very deep directory trees. Badly coded implementations may break if they don't allocate enough space to hold long names, or recurse into deep directory trees. Both implementations we tried turned out fine on this test.

2.3.3 Evasion using root privileges

The last type of tactics we discuss in this chapter have a more active approach. The techniques discussed here can be used by an attacker to actively break the filesystem monitor, so an administrator will not receive alerts from it. Assuming the monitor is installed and configured properly these attacks require the attacker to gain root privileges on the system before he can execute these attacks. The following attacks (and variations on them) can be launched once an attacker gains root privileges:

Restore ctime As mentioned earlier, the ctime of a file can not be restored using the *touch* command. There is however another way for an attacker to restore this file property. If the attacker gains root privileges he is able to change the time on the system he compromised. This ability can theoretically be used to modify the file at exactly the right time.

Cronjob modification A more aggressive option an attacker with root privileges has is modifying the cronjob that performs the periodic checks on the system. This way the administrator will always receive E-mails that all is well, and never know about any intrusion.

Database modification Another possible attack to evade detection is to modify the database of checked files and directories, so files belonging to the attacker are excluded.

IDS modification The last method we list here is altering the filesystem monitor itself. By replacing the monitor with an altered version the IDS may omit files belonging to the attacker in its report. This tactic may also be performed on shared libraries that are used by the IDS. The version of Mtree we tested is dynamically linked with two libraries on the system. Therefore this implementation is potentially vulnerable to this type of attack.

2.4 Evasion prevention

In the previous section we described a number of ways to evade a filesystem monitor. In this section we discuss how implementations or their configuration may be altered to prevent these evasions or mitigate their effects.

2.4.1 Configuration

The following points should be taken into account when configuring a filesystem monitor, to prevent evasion attacks:

Careful configuration An administrator needs to consider carefully which files or directories to include (or exclude) in the configuration, and which checks to perform on each file or directory. This can unfortunately be a time consuming job, especially when using Mtree's inflexible exclude list.

Use BSD securelevels To prevent an attacker from changing the operating system's time or date in order to reset the ctime of a file BSD, securelevels may be used. When a system is booted into securelevel 2 or higher the system's time and date may only be altered by one second at the time. This makes making large changes a very timely business.

Use BSD flags As mentioned in the previous section, the tested implementations have problems detecting malicious changes in logfiles, because the attributes of these files change often for legitimate reasons. A way to better protect these types of files is to use BSD append-only flags to

prevent deletion of information from these files, and to monitor the addition or removal of these flags. This may however cause problems with logrotation programs.

Use hash checking Because a number of attributes (e.g. atime, mtime and size) can be restored by an attacker after changing the contents of a file, hash checking should be done where possible. This reduces the chances for an attacker to successfully change a file without being detected.

Compile with static libraries If an attacker alters a system's libraries this should not affect the IDS. To achieve this the IDS should be compiled with static libraries, on a system that has not had any connection to the outside world yet. This is possible with both tested implementations.

Prevent altering of IDS To even further prevent an attacker from altering the IDS, both the IDS executable, its configuration file, and the database should be stored on read-only media. Another option is to read-only mount the machine's filesystem on another host (e.g. through NFS) and do the checking there.

2.4.2 Implementation improvements

Implementations may be improved in the following ways to prevent evasion or improve overall checking reliability:

Realtime detection This type of detection eliminates the possibility for the attacker to restore file or directory properties to evade detection. Realtime detection may be possible using the *kqueue(2)* and *kevent(2)* system calls[9].

Anomaly detection Anomaly based intrusion detection may help to lower the false positive rate of filesystem monitors. As mentioned in section 2.2.3 this is difficult to achieve, and we are unaware of implementations that implement this in filesystem monitoring.

Pentesting The last suggestion for improval we discuss here is penetration testing, or pentesting. Implementations should be carefully tested on how they react to evasion and other attacks before they are released to the public. While this will probably not uncover all bugs, it can surely help improving the security of these programs.

2.5 Conclusion

Filesystem monitors are an interesting type of HIDS. Because they can help detect a break-in on a system after it has occurred they are a good last line of defense in a system's security.

Unfortunately, this type of intrusion detection has a number of disadvantages. Because filesystems tend to be very dynamic in nature it is hard to create a configuration that catches all intrusions, while not producing many false positives. Something trivial as installing a new application can create a huge number of alarms on an IDS of this type. Another disadvantage is that this type of IDS generally doesn't work realtime. It is therefore possible for an attacker to cover up his tracks before being detected.

Altogether, we believe that, looking at the current state of filesystem monitors and the implementations we tested, an IDS of this type should only be used to check files that rarely change, such as system binaries and libraries. Because these are also the types of files an attacker will want to alter after a successful break-in, this will still be quite effective.

Chapter 3

Logfile analysis

In this chapter we discuss logfile analysis, one of the four methods of Host Intrusion Detection. By analysing logfiles and determining if intrusion attempts were logged, an intrusion detection system can warn system administrators about possible intrusions taking place.

There are multiple ways of analysis possible. In this chapter we discuss two reference implementations, Swatch and Sec. They both use pattern matching for their analysis and can issue warnings using multiple methods. We have chosen Swatch because we were both familiar with it and because it is used often, and because it is an open source and easily useable program.

Sec can be seen as an evolution of Swatch. We have chosen Sec because it uses the same principle as swatch (pattern matching), but adds some complexity to that simple principle. By evaluating both Swatch and Sec, we were able to see the advantages and disadvantages of both applications more clearly than we could have if we had just chosen one program to test. So in effect, Sec was chosen as a reference implementation of a tool complex enough to be different enough from Swatch, yet simple enough to work with in the context of our project.

We look at the shortcomings of the pattern matching approach in section 3.3 and show alternative approaches in section 3.4. The effects of the extra complexity in Sec compared to Swatch is outlined in both sections.

3.1 Features

3.1.1 Overview

Logfile analysis tools come in a few flavours. There are tools that perform:

- Pattern matching
- Pattern matching with correlation between events
- Anomaly detection

We've taken an in-depth look into two programs: Swatch[22] and Sec[20]. Swatch being a simple pattern matching tool, whereas Sec can be used to correlate between events.

Swatch offers the following features:

- Applying (extended) regular expressions to logfiles.
- Echo matching loglines to the terminal Swatch is running on.
- Mail an alert to a predefined e-mail address on a match.
- Run a defined command on a match.
- Warn a logged in user, using *write*
- Flood protection by throttling alerts.
- Thresholds (e.g. two failed logins doesn't issue a warning, but three failed logins does).
- Follow rotated logfiles.

- Only act on matches during a defined time-frame.
- Run Perl miniprograms in a global context, in the main loop, in matches or in throttle loops.

Sec offers almost the same features as Swatch does, but adds correlation to the mix:

- Create a context when a line is matched.
- Add items to a context.
- Alias or unalias a context, so it can be referenced using another name.
- Only match when a context exists.
- Compress multiple events of the same type into one, effectively throttling actions.
- Assign values to variables to be used later.
- Run Perl miniprograms as part of an action, variable assignment or context creation.
- Perform actions based on the output of scripts run after a match. A different action can be taken if a script fails compared to when a script runs successfully.
- Perform actions based on the matching of two different lines in a defined time window.
- Create events based on the current time and date.

3.1.2 Conceptual differences

From the feature lists it becomes clear that Sec is a more powerful tool than Swatch. However, the feature lists don't clearly show the conceptual difference between a simple pattern matching tool like Swatch and an event correlation tool like Sec.

The main difference between a tool like Swatch and a tool like Sec is that Swatch is one-dimensional, which means it hardly considers the timing of events in its matching algorithm nor in the actions it performs. Sec on the other hand, is at least two-dimensional, since it can both act on input events as well as on timing information, called contexts. Those contexts can exist during a certain time period and can be controlled by events. Although matching is still done the same way, the expressive power of Sec's constructs increases its useability a lot. We discuss how this effects its effectiveness as an IDS in sections 3.3 and 3.4.

3.2 Configuration & maintainability

3.2.1 Installation and configuration

Both Swatch and Sec are written in Perl and consist of only one Perl script and one configuration file. Swatch can monitor one input file per instance, Sec can monitor multiple files. This severely limits Swatch' use in big environments, where multiple files need to be watched. Although it is possible to run hundreds of Swatch instances on today's hardware, it will be quite hard to maintain and debug all those Swatch instances. Sec accepts a pattern as input filename argument and can thus be used to monitor multiple files simultaneously.

The configuration files for both applications are quite straight-forward. They both have a simple syntax. Consider the examples below.

Swatch:

```
watchfor        /error/
       mail admin@example.org,subject=Error
```

Sec:

```
type=Single
desc=Match 'error' and mail the admin
ptype=RegExp
pattern=error
action=pipe '%t: $0' /usr/bin/mail -s "Error" admin@example.org
```

Swatch and Sec would match any logline containing the text *error* and mail the admin about it. The Sec configuration is more complex, due to its bigger feature-set, but it's still quite easy to understand. The example above is a simple one and more complicated examples would give more complicated rules. However, even with Sec, there's a limited set of commands, a limited set of rule types and a limited set of actions. Effectively, this means that configuring Sec and Swatch isn't very difficult in itself. However, rulesets will have to be hand-made which is a large and difficult task easily leading to errors in the configuration itself or unexpected result due to incorrect assumptions being made.

3.2.2 Maintenance

Due to the inherent nature of Swatch and Sec — their reliance on configuration files containing rules which input files are matched against — they can be hard to maintain. This depends greatly on the usage.

When used to match on static strings in log output of daemons, the rules won't need to be changed very often. It may be necessary to add a few rules for logs indicating exploitation attempts of certain new bugs every now and then, but for the most part the rulesets can stay intact.

When Swatch or Sec are used to monitor more dynamic logfiles, like Apache logs, maintenance becomes a lot harder. Consider the case where an administrator working for a shared hosting company wants to monitor Apache logs for intrusion attempts on the company's shared hosting platform. Every day new exploits are released for applications written in scripting languages like PHP, ASP and Perl, possibly running on Apache. The administrator has to write match-rules for every possible exploit. This is a lot of work, even if some rules can be combined or generalised.

In conclusion, the maintainability of Swatch and Sec depend greatly on its use. When they are used for matching strings in a more or less static context, such as for example, the output of a SSH daemon, their maintenance is doable. However, maintenance increases if Swatch or Sec are used to detect new exploits as they are released. A better choice would be to use a network IDS like Snort[23] or a tool capable of anomaly detection in such cases.

3.3 Evasion possibilities

3.3.1 Event correlation

As we have outlined in the previous sections, Swatch is limited in its use due to the simple pattern matching method it uses and due to every instance only monitoring one log file. Innocent loglines may not be so innocent anymore when they show up over multiple logfiles in a short period of time. However, Swatch doesn't have the ability to correlate between events from multiple inputs and thus fails to detect such events.

3.3.2 Thresholds

A dangerous option existing in both Swatch and Sec is setting a threshold on matching input lines. This means that Swatch and Sec will only perform an action if a defined number of matches have occurred during a certain time window. An example would be to only warn an administrator when six failed logins occur in a time window of one minute. If an attacker, performing a brute force password attack, throttles his or her login attempts to only four per minute, the attack will go undetected. In Swatch the situation is worse than in Sec, because in Sec it's possible to create a context for a failed login *per user*, so setting thresholds can be done per user. In Swatch the threshold will be set for *all the users* and, for a busy system, has to be set quite high to avoid false positives. The higher the threshold, the easier avoiding detection becomes.

3.3.3 Encoding

When using Sec or Swatch on dynamic logfiles like webserver access logs, there's a risk of not detecting certain events. An attacker can use all kinds of encoding techniques to thwart the matching rules administrators have built. Let's consider the following example. Let's say the administrator wants to detect anyone trying to obtain the file **/etc/passwd** using some kind of web exploit. The corresponding logline would be:

```
127.0.0.1 - - [31/Jan/2005:03:17:51 +0100] "GET
/page.php?page=../../../etc/passwd HTTP/1.1" 200 2313 "-" "" "-"
```

Matching the string "/etc/passwd" would trigger a warning, which the administrator can act upon. However, let's consider the following logline:

```
127.0.0.1 - - [31/Jan/2005:03:17:51 +0100] "GET
/page.php?page=..%2F..%2F..%2Fetc%2Fpasswd HTTP/1.1" 200 2313 "-" "" "-"
```

The attacker has rewritten the request to use hex-encoding for the forward slashes, effectively rendering the match on "/etc/passwd" useless.

3.3.4 Evasion using root privileges

As with filesystem monitors, an attacker can alter the behaviour of logfile monitoring tools when he or she gains root access. Most intruders kill any running syslog daemons as soon as possible after gaining root. Since many daemons use syslog as their logging method, killing the syslog daemon will effectively disable all logging. Clearly no single log watcher can detect anything happening after that point in time. This only applies to logs created by a syslog daemon, of course. Other logfiles can be monitored without problems when syslogd is killed.

Not only a syslog daemon can be killed, the log watcher itself can be killed too. Obviously, log watchers are useless when not running.

3.4 Evasion prevention

We've seen a few evasion possibilities with regards to log watchers. In this section we discuss a few ways of preventing against evasion or mitigating the effects thereof.

3.4.1 Event correlation

The main difference between Swatch and Sec is that the latter can correlate between events (hence its name, *Simple Event Correlator*). This can be useful in many circumstances, especially when monitoring multiple logfiles which may come from multiple machines. This evasion possibility thus only applies to Swatch.

The easy fix would be just not to run a tool like Swatch but use something like Sec instead. Since Sec offers the same features as Swatch, and a lot more, it can be used as a stand-in replacement for Swatch. A more elaborate scheme would be to use Swatch' mini program feature to implement a rudimentary correlation tool inside Swatch. When a line is matched, a mini program could be started which could set up a context for that match, just like Sec does. Other matches could run mini Perl programs which use the said context to determine their actions. Another way to implement something similar would be to write another tool which gets its input from Swatch whenever certain lines are matched, which it aggregates and perform actions on just like Swatch does. However, since such adaptations to or additions on Swatch are basically gross hacks (with all the disadvantages gross hacks generally have), the use of a tool like Sec would be a better choice. The somewhat more complicated syntax of Sec's configuration file shouldn't be any reason not to use it over Swatch.

3.4.2 Encoding

Matching on possibly encoded lines using regular expressions is a nightmare. Because there are tens if not hundreds of possibilities of encoding even for small strings, simple pattern matching tools proof to be inadequate. There are some options, however.

Because matching on encoded strings is almost impossible, a tool could be used to decode the strings to ASCII and pipe that to the log watcher used. Let's take the example from chapter 3.3.3 again. Consider the following logline:

```
127.0.0.1 - - [31/Jan/2005:03:17:51 +0100] "GET
/page.php?page=..%2F..%2F..%2Fetc%2Fpasswd HTTP/1.1" 200 2313 "-" "" "-"
```

Matching on "/etc/passwd" won't work with such a line. But, after the decoder has decoded the hex-encoded %2F to a forward slash, things become better:

```
127.0.0.1 - - [31/Jan/2005:03:17:51 +0100] "GET
/page.php?page=../../../etc/passwd HTTP/1.1" 200 2313 "-" "" "-"
```

However, even with such a simple example there are more problems. Let's consider this example (after decoding):

```
127.0.0.1 - - [31/Jan/2005:03:17:51 +0100] "GET
/page.php?page=../../../etc//passwd HTTP/1.1" 200 2313 "-" "" "-"
```

Adding another forward slash will make the match-rule useless, although the attack still works. Luckily, this decoded example can be matched quite easily using regular expressions: matching on one *or more* forward slashes will do the trick.

There still is a problem using a stand-alone decoder, though. What if one wanted to match on the encoded string? Many intrusion attempts make use of encoding to be able to actually work. One could let a log watcher match on both the encoded and decoded strings, but for a lot of files that would be quite a waste of resources. A better approach would be to use a pattern matching tool written with encoding in mind or use an anomaly detector[4].

3.4.3 Thresholds

When using thresholds to limit false positives and prevent flooding, an administrator opens possibilities for undetectable attacks. As we have seen, this problem is worse in a pattern matching tool like Swatch when compared to a tool capable of correlation, like Sec. As with the previous evasion possibilities, there are ways to prevent against attacks which take notice of the set thresholds.

A way to make Swatch more useful to this regard is combining logfiles into one stream for Swatch to process. A tool like multitail[13] can be used to do just that[1]. By combining the log files, the same events happening over multiple systems can be matched using one match rule and one threshold. The threshold can thus be set for the entire range of systems, reducing false positives. The problem persists, though.

Consider a threshold of five failed logins for any given account in a time window of one minute. Failed logins for telnet, ssh, ftp and webmail are aggregated, so a round-robin bruteforce password crack session over multiple daemons doesn't help an attacker much. However, if the attacker makes sure (s)he stays below five failed logins over all daemons, the attack will still be undetectable. Luckily this can be prevented using a tool like Sec using two contexts. There could be a short-lived context for the five failed logins per minute threshold and a long-lived context for, say, twenty failed logins per week. A determined attacker could still mount an undetectable attack, but since he or she can only try a maximum of nineteen passwords per week it won't go really fast. Since an attacker normally doesn't know the system is monitored using such thresholds, (s)he wouldn't limit the login attempts that much and thus the attack will be detected.

3.4.4 Evasion using root privileges

After an attacker gains root, he or she normally can do anything on the local machine. This includes killing the syslog daemon or any log watcher running. There isn't much one can do about that, but there are some options.

One option would be to run a log watcher from within init, which will restart it as soon as it's killed. The init configuration file, the log watcher binary and the log watcher configuration file have to be protected in order to make this approach useful. In Linux the notion of file attributes exists, but root can delete those flags as easily as add them. On BSD securelevels can be used to make it impossible to remove file flags, but securelevels make maintenance a lot harder.

A better approach would be to gather all logfiles on one (hardened) system using the remote syslog capabilities of syslogd and using NFS for other log files. Especially NFS would be useful, although a root user may unmount the NFS mount and restart daemons logging to a NFS mounted filesystem to make them use the local filesystem instead of the remote one. This can also be detected (for example by watching for markers every 5 minutes, alarming an administrator if a system doesn't report in), but such schemes can be thwarted by determined attackers, too. In the end it's all about useability and maintainability of the logging infrastructure and about the attacker's skills. Automatically restarting a syslog daemon logging to a remote syslog system will prove to be sophisticated enough for most

[1] In fact, multitail can be used for the same purposes as Swatch

scenarios. Although an attacker may disable the syslog server when (s)he finds out it gets restarted all the time, the restarts themselves can trigger an event on the remote syslog system and alert an administrator, effectively detecting the intrusion.

3.5 Conclusion

In this section we have looked at the concepts behind pattern matching log watchers and determined to what extent they can be used and misused. We've seen that although Swatch can be successfully used to detect intrusions, the more sophisticated Sec is preferable because of it's correlation capabilities.

However, for some logfiles, such as webserver logs, anomaly detection systems are preferable over tools like Sec. Anomaly detection systems have the advantage that, when first run, they don't need to know anything about the legitimacy of loglines. After learning how the normal loglines should look like, anomaly detection systems can detect anomalous loglines and report about them. Difficult decoding isn't needed with such systems, since they will detect the anomaly wether it's encoded or not. A sane setup would be to use a pattern matching log watcher to watch over more or less static log files, like those from a SSH daemon and use anomaly detection systems for more dynamic logfiles like those from webservers.

Chapter 4

Connection analysis

The third type of HIDS we discuss in this document is network connection analysis. Connection analysing HIDS implementations detect incoming network connections to the host they run on. They do not perform pattern matching and correlation of events directed to different hosts. This is the domain of Network-based IDS implementations, such as Snort.

As in the previous chapter we discuss several aspects of this type of HIDS. We look at features, configuration and maintainability, evasion tactics, and ways to mitigate the effects of evasion attempts.

The implementations we use as reference in this chapter are Scanlogd[21] version 2.2.5, and version 1.1 of PortSentry[18]. The first was chosen because it is a simple and effective example of a detection tool. It was originally designed for Phrack Magazine[17] to"illustrate various attacks an IDS developer has to deal with." PortSentry was chosen because it is one of the more well-known implementations of this type of HIDS.

Because PortSentry's advanced features use raw sockets, these options can not be enabled on a BSD system. For testing this aspect of the implementation we used Debian 3.1 (Sarge), using kernel image 2.4.27-1-386[5]. All tested scanning types we discuss in this document where executed using version 3.75 of the popular Nmap[14] tool.

4.1 Features

Connection analysers monitor connections that are made to a system. This allows them to detect unauthorised connections, and various types of portscans. The list below shows which features Scanlogd supports:

Unauthorised TCP connections Scanlogd is able to detect connections to unauthorised TCP ports and report this in the system logfile.

Portscan detection Apart from normal connections Scanlogd also detects SYN, FIN and XMAS type portscans.

Passive detection Scanlogd passively monitors the network. It does not show any signs of detection to the connecting party.

Flood protection Scanlogd has a flood protection facility to prevent the system's logfile from being filled up. It stops displaying alerts when the amount of scans of a certain type reaches a set value.

PortSentry is a more advanced implementation. As the following list illustrates this shows in the number of available features:

Unauthorised TCP and UDP connections PortSentry is able to detect unauthorised connections on both TCP and UDP ports.

Port binding PortSentry's basic mode binds to administratively selected ports. This not only allows it to detect connections and scans to these ports, but also effectively prevents unauthorised daemons on the system from binding to these ports.

Passive detection Aside from the basic mode, PortSentry has two more modes of operation: stealth mode and advanced mode. Stealth mode passively listens for connections on administratively

selected ports, using raw sockets. Advanced mode listens on a complete range of ports up to a selected port (e.g. all ports up to 1024), also using raw sockets.

Host blocking The most distinguishing difference between Scanlogd and PortSentry is that the latter allows for active blocking of an offending host. The tool can be set up to add a host to the system's **/etc/hosts.deny** file if an unauthorised connection is detected. Another option is to run a specified command to block the host (e.g. adding an entry to the system's firewall). Goal of this is to prevent further connections or scans on the system.

Banner display A more friendly option PortSentry offers is to display an informational banner to the offender, instead of blocking them.

4.1.1 Conceptual differences

As the above featurelists illustrate there is a conceptual difference between the two discussed implementations. Scanlogd is designed to use a passive and secure approach. It does not use port binding, and uses flood protection. This approach also shows in the default configuration. By default, Scanlogd will not show an alert until seven unauthorised connections from a host are received.

PortSentry uses a much more active approach. It binds to ports (in basic mode) and actively responds to an offense by informing the connecting party, or even blocking it. A default configuration of PortSentry responds to any unauthorised connection, even if only one offending connection attempt has been received. This can be considered as a quite aggressive approach.

4.2 Configuration & maintainability

Connection monitors have a number of settings that can be configured. There's the ports that the tool has to monitor and the amount of connections that triggers an alert. For Scanlogd there also is a setting for when to trigger the flood protection (x scans within y seconds). PortSentry has additional settings for the displaying of a banner, and the blocking of hosts.

The text above shows that the amount of settings is quite limited. The tools are also not likely to be reconfigured a lot (except when services on the system are added or removed), which is probably why Scanlogd does not even use a configuration file, but requires the administrator to specify the desired settings at compile time.

The one thing that does need a lot of maintenance is the host-blocking configuration of PortSentry. Firewall rules and entries in **/etc/hosts.deny** need to be cleaned up once in a while to prevent the list from getting too large. It is also very well possible that friendly hosts will end up getting blocked because they accidently connect to a wrong port, or to a service that no longer exists on the system. These will need to be removed manually. Setting the amount of scans that trigger a block may reduce the amount of times these accidents occur, but will not completely remove this problem.

4.3 Evasion possibilities

As with the other HIDS types we discuss in this paper, there are ways to evade detection. In this section we discuss several ways to evade detection by a connection analyser. We also discuss a number of options an attacker has to actually break the IDS or cause a Denial of Service.

4.3.1 General

The following general options are available to an attacker to evade detection by a connection analyser:

Use of unsupported protocols The implementations we discuss in this chapter do not offer support for the IPv6 protocol. Scans directed towards an IPv6 interface are therefore not detected. Scanlogd also does not offer UDP support, so does not detect scans of this type either.

Slow scanning If an attacker scans ports on the system slowly (e.g. one port every hour), he will not reach the trigger amount of the connection analyser. This method will of course not be successful if the trigger amount is set to zero (which is the default setting for PortSentry).

Use unsupported scans The connection analysers we tested do not detect every type of portscan. Protocol scans, NULL scans, SYN—ACK scans, and ACK scans are not detected. While not as effective as for example SYN or FIN scans, these type of scans can reveal interesting information about the system to an attacker. A SYN—ACK or ACK scan, for example, can show an attacker which ports on the system are filtered by a firewall. This can give an indication on which services are running on the system.

Use decoy scanning A decoy scan performs a portscan on a system from a large number of source IP's, including the real source of the scanner (to actually get some results). Due to Scanlogd's floodprotection it is possible to successfully scan a system using a decoy scan, without having the real source IP appear in the logfile. When PortSentry is used, this scan is only useful if blocking is not enabled.

Use a large pool of IP's The last option an attacker has is to simply use a lot of IP's to scan from. This can, like the slow scan mentioned above, be used to stay below the trigger threshold. It can also be used if an attacker wants to scan a system, and has plenty of machines to his avail. Even if a lot of these machines get blocked due to scanning he will still get his results.

4.3.2 Denial of Service

If an attacker succeeds in causing a Denial of Service on the connection analyser he might be able to continue his probes without being blocked. Another reason why an attacker would want to exploit this is because he simply wants the system that the connection analyser runs on to go offline. The following methods can be used to cause a Denial of Service on a connection analyser:

Fill up the harddrive The first method an attacker can use to cause a Denial of Service on a connection monitor is to fill up the harddrive of the system. This can be achieved by filling up the logfile with a large number of probes. This method is hard to achieve when Scanlogd is used, due to its floodprotection facility.

Spoof IP's of vital external services PortSentry ignores scans from its own IP address by default, to prevent the local system from getting blocked. An attacker can however still cause trouble by spoofing other addresses to have them blocked. For example, if the IP address of the system's network gateway is used as source address in a connection attempt this will cause the connection monitor to block this IP, and effectively disable its connection to the Internet.

4.3.3 Inside detection

PortSentry's basic mode effectively prevents users from running unauhorized daemons. When using Scanlogd or PortSentry in stealth or advanced mode, these daemons are not blocked, and connections to their ports are not even detected. While blocking or detecting these daemons is not the primary goal of the implementations we looked at, this is an important fact to keep in mind.

4.4 Evasion prevention

In the previous section we discussed a number of ways to evade, or even break a connection monitor. In this section we talk about ways to prevent these kinds of attacks.

4.4.1 General

The following things can be done to prevent evasion attacks on a connection monitor:

Implement unsupported protocols By adding support for IPv6 and UDP to a connection monitor, connection attempts using these protocols can also be detected.

Set a lower trigger value Evasion using slow scans can be prevented by blocking a host after a single unauthorised connection attempt. This however, increases the false positive rate dramatically. Another option is to set multiple thresholds (e.g. 3 per minute and 20 per day).

Add detection for more scans Both tested implementations lack detection of a number of scans (see the previous section for a list). By adding support for these scans the implementations can be made more effective.

Use smarter blocking The initial response to evasion attempts using a large number of IP's may be to set wider blocks (e.g. whole subnets instead of single hosts). This strategy will, however, work counter-productive since the amount of false positives will increase enormously.

4.4.2 Denial of Service

While preventing them completely is infeasable, there are a number of ways to reduce the risk of a Denial of Service attack. These are:

Implementation of flood protection By implementing flood protection, like Scanlogd does, huge logfiles can be prevented. Unfortunately, this allows an attacker to evade blocking using a decoy scan.

Ignoring of vital services By putting IP addresses of vital services (e.g. network gateway and DNS servers) in the list of ignored hosts the Denial of Service risk can be reduced. This will, however, require extra configuration.

4.4.3 Inside detection

Blocking unauthorised daemons may be achieved by always using daemons on the configured ports. Of course this is not a very efficient approach if an administrator wants to monitor all 65536 ports on a system. A better method would be to alter the implementations so that connections to an unauthorised port will be detected if a daemon is running on this port.

4.5 Conclusion

Connection monitors perform their task very effectively. Those that support proactive measurements like blocking are also very powerful.

Both tested implementations are mostly effective when it comes to portscan detection. This type of malicious activity is, however annoying, not a real danger to a system if it is configured and patched properly. For an administrator, the time needed to evaluate generated alerts is not worth the advantage these tools bring. This is especially true when the tool blocks the host from which a scan originates. Automatic blocking is also discouraged in general, since it creates a large Denial of Service risk.

The only type of connection monitor that could be a real asset to an administrator is one that effectively recognizes unauthorised daemons. PortSentry's basic mode does this well (it prevents them from even starting), but is inefficient when a large number of ports need to be monitored.

Chapter 5

Kernel based IDSs

The last method of host based intrusion detection we discuss is kernel based intrusion detection. A kernel based IDS is an addition to or adaption of a kernel to have the kernel itself detect intrusions. There are many ways to detect intrusions this way, including:

- Anomaly detection based on a user's system usage.
- Logging possibly maliciously used system calls.
- Anomaly detection on the order of system calls in processes.
- Anomaly detection on the arguments of system calls in processes.
- Logging changes made to system binaries.
- Logging port scans or probes.

In this section we discuss the features, use, evasion and evasion prevention of the two kernel based IDSs we have studied: LIDS[10] and an IDS based on process behaviour rating[7], to be called IDSpbr. We've chosen LIDS because it's a well-known IDS for Linux. IDSpbr was chosen for it's unique and experimental concept of detection. Comparing both methods could potentially provide us with interesting insights in kernel based intrusion detection.

5.1 Features

The two kernel based IDSs we have studied are quite different from each other. LIDS is much more feature rich than IDSpbr due to it's longer existence and broader scope. However, they both fulfill the same purpose: detecting intrusions using in-kernel functionality. Below, we look into the specific featuresets of both systems to determine what they are capable of.

5.1.1 LIDS' featureset

LIDS has a lot of features, most of them meant to protect against intrusions and alterations of system binaries and directories. While protection isn't really a job for an IDS, LIDS can detect intrusions by logging offenses against the defined protective rules. Below a list of LIDS' features:

- File and directory protection, preventing alterations even by root.
- Hiding of files and directories.
- Setting capabilities on a per process basis
- Protect processes, blocking signals from possibly unauthorized users.
- Blocking network related tampering, like changing firewall settings.
- Preventing kernel module loading or unloading.
- Preventing raw disk I/O.

- Discovery of needed ACLs.
- Send out security alerts using SMTP.

A process can have many capabilities assigned to it, which override the system wide LIDS settings. This way the system binaries can use certain kernel features while other binaries cannot.

5.1.2 IDSpbr's featureset

Although IDSpbr is already useable in it's current state, it's only meant to be a prototype to see if a theoretic approach to intrusion detection would work in practice. Therefore it's featureset is quite small:

- Detecting exploitation of stack or heap buffer-overflows by monitoring *execve()* calls.
- Detecting exploitation of symbolic link race conditions or other insecure symbolic link use.
- Detecting local denial of service attacks.
- Killing processes involved in an attack.
- Sending alerts using SMTP, writing to the console or syslog.
- Increasing the level of suspicion when processes are more dangerous (examples being a process running as root, or a process accepting TCP connections).
- Perform anomaly detection based on the order of system calls per process.

Detecting execve() calls from the stack or the data segment of processes will enable the IDS to detect most exploits for today's software. The symlink related detection mechanism may prove useful too, although such attacks happen less often. Other tests can be implemented, enhancing the IDS's useability.

5.2 Configuration & maintainability

Because LIDS and IDSpbr differ so much, their installation, configuration and maintenance greatly differ too. That's the reason we have split this topic in two separate topics, one section per IDS.

5.3 Configuring and maintaining LIDS

The installation of LIDS is quite involved. Since it's a kernel patch, the right kernel version's sources should be downloaded first. After applying the patch a LIDS-enabled kernel can be built, but not after first configuring the kernel to enable LIDS. Before booting into the new kernel, the LIDS ACLs should be configured.

Configuration is done using two tools: *lidsconf* and *lidsadm*. *lidsconf* can be used to configure the LIDS ACLs, while *lidsadm* can be used to administer LIDS itself. This includes enabling LIDS or setting system-wide capabilities. During installation, the make script asks for a password. This password can be used to temporarily disable LIDS for only one terminal for administrative purposes. During installation the administrator should make sure the system remains useable after booting into a LIDS enabled kernel. This means (s)he should not disable anything yet, because that may lead to an unusable system. After booting into a LIDS enabled kernel, the configuration can be built bit by bit using the configuration tools.

Depending on the administrators paranoia, a lot of ACLs can be set. For most uses, however, it's easiest to secure only the system binaries and configuration files against tampering. Such an ACL can be made like below:

```
lidsconf -A -o /sbin -j READONLY
```

This will make everything in the "/sbin" path read-only, which means system binaries in that directory can't be tampered with. Setting the READONLY flag for all system binaries will effectively block attackers from installing a rootkit which overwrites system binaries like "netstat", "ps" and "ls". There are all kinds of capabilities, most of which are essentially ways of hardening the system. A paranoid administrator can harden his or her system to an almost unusable state, if (s)he wishes to.

This introduces the topic maintenance. In general it's true that the more a system is hardened, the harder it is to administrate. The same goes for LIDS hardened systems. If an administrator wants to install a new application, or update an already running one, (s)he needs to create a LIDS free session (LFS) in which LIDS is totally disabled. During that session the administrator can perform administrative duties, like installing applications or configuring LIDS itself. After updating or installing an application, LIDS may need to reload the configuration file to protect any newly installed files. So every time something needs to be done, an LFS needs to be started to be able to get work done. This can become quite a burden.

5.4 Configuring and maintaining IDSpbr

Installation of IDSpbr is quite straight-forward: the kernel module needs to be built and loaded. Building is merely a question of running *make*, loading can be done using *kldload*.

Configuration is a bit more involved. The IDS can be configured entirely using *sysctl*. For every test, there are a number of settings to configure. The "exec" test, for example, has the following settings:

```
ids.tests.exec.active: 1
ids.tests.exec.points_stack_exec: 0.00
ids.tests.exec.points_suid_exec: 0.00
ids.tests.exec.points_sgid_exec: 0.00
ids.tests.exec.points_suid_exec_bad_dir: 0.00
ids.tests.exec.bad_dirs: /tmp:/home:/var
ids.tests.exec.permitted_suid_programs:
ids.tests.exec.shells:
ids.tests.exec.points_shell_exec: 0.00
ids.tests.exec.points_symlink_exec: 0.00
ids.tests.exec.points_symlink_exec_bad_dir: 0.00
ids.tests.exec.points_ld_vars: 0.00
ids.tests.exec.points_susp_env: 0.00
```

Configuring these settings could be considered a black art, but luckily there's a nice tool to do it automatically. The IDS can be configured to dump an XML-message for every process that has been assigned to a defined number of points. By configuring all tests to give one point and setting the XML-dump threshold to one, every process that has a test succeed, will lead to an XML-dump. This dump can then be run through an application which uses a genetic algorithm to decide which settings generate the least false positives. Instead of letting the IDS learn, the configuration file shipped with the IDS distribution can be used.

There isn't much maintenance needed for this IDS. When false positives happen too regular, the settings may be adjusted a bit to make it perform more reliable. There isn't much else to maintain.

5.5 Evasion possibilities

With LIDS, there aren't many evasion possibilities when the system is properly configured. Since only root can override the limits LIDS imposes, a normal user can't do much. Even when an attacker gains root access, (s)he won't be able to do much, like loading a kernel module or installing backdoored system binaries as "ps" and "netstat". There may be a possibility to capture the LIDS password by letting the administrator run a backdoor *lidsadm*. Although LIDS does have the capability to only allow root to create an LFS on a designated console, be it a virtual terminal on the console or a serial console, in many situations this will be too much of a hassle. In such cases, an attacker may let the administrator run a backdoor (by changing the PATH environment setting in root's shell's profile for example). This backdoor can then capture the password and mail it to the attacker.

Evading the limits IDSpbr imposes is easier. There are a few problems with the current available module, which we'll outline below. The author acknowledges some of these problems in his thesis[25] and discusses future work.

We have devised a simple program behaving like an exploit for a stack buffer overflow bug. The "exec"-test detected that the test program executed */bin/sh* from the stack and the IDS sent out a warning e-mail. However, by altering the program to jump into the code segment of the C library to call the "execve" system call from there, the IDS could not longer detect the otherwise perfectly working 'exploit'. When an IDS like IDSpbr would become mainstream it's quite possible that exploits will be altered to use the same trick as we did. If so, the "exec"-test becomes useless.

IDSpbr has a setting called the "TTL". This is used to reset the points a process has accumulated after a given amount of time. When an attacker executes an exploit that would trigger multiple tests eventually leading to reaching the warning threshold, (s)he could delay the exploit to wait till the accumulated points get reset. This way an attack can go undetected.

Another way to evade the IDS would be to run an attack using multiple processes in stead of one. This way the points are assigned to multiple processes, without any process ever hitting the warning threshold.

5.6 Evasion prevention

The LIDS evasion technique presented in the previous topic can be prevented by making sure that the executables and shell configuration settings for root are all read-only. It should not be possible for an attacker to trick root into running the wrong *lidsadm* executable without him or her noticing it. This is already possible with LIDS, but not something every administrator would think about. LIDS already is capable of only accepting an LFS from a designated console, like a serial console. However, this is quite a hassle for many situations, so a handier method would be useful.

IDSpbr is still in it's infancy and could be developed further to detect more attacks and handle the evasion techniques discussed. The "exec"-test can't be made much better with regards to calling the "execve()" system call from the stack or data portions of a process's memory. Since an exploit has full control over the process, the kernel can't avoid the exploit jumping to the code section and calling syscalls from there. However, it's possible for the OS to set the stack and data sections non-executable[15]. This would be a more efficient method, although still evadible[16] (but exploiting a simple buffer overflow becomes a lot harder).

IDSpbr could be altered to not only count on a per-process basis, but also keep a global counter. When a certain threshold for the global counter is reached, a warning could be e-mailed. The warning message could contain all processes involved in increasing the global counter. This way, even if malicious behaviour is divided over multiple processes, the attack as a whole will still be detected.

5.7 Conclusion

Both LIDS and IDSpbr have their own merits. LIDS can be effectively used as a way of protecting the system against tampering and can be used to warn an administrator when anyone tries to tamper the system. IDSpbr is capable of detecting many of today's exploits, but still needs more work to make it even more worthwhile to run.

Although LIDS' maintenance can be hard, using some of it's features may still be useable without placing too much burden on the administrator. Only protecting system binaries and configuration files and disallowing to load kernel modules doesn't impose too much restrictions while still protecting the system against many intrusions, especially those involving an attacker gaining root access. The more esoteric options are probably not very useable in most environments, because they make maintenance a lot harder.

IDSpbr still needs more work, especially it's anomaly detector. The anomaly detector uses the order of system calls, but could possibly adapted to use the arguments for system calls. This has been done before[24] and seems to be a useful approach to in-kernel anomaly detection.

Chapter 6

Conclusion

In this document we discussed several types of Host-based Intrusion Detection systems. All of these have their own set of advantages and disadvantages.

Filesystem monitors are a good last line of defense in the security of a system, but are hard to maintain if the files that are being monitored change often. This type of IDS should therefore only be used to check files that rarely change, such as system binaries and libraries.

Logfile analysers can be very helpful in detecting intrusions as well. For this type of tool to be useful it is important that it offers support for correlation of events from multiple logfiles.

Connection analysers are good at detecting portscans, but require too much management, compared to the actual use they bring to administrators. The only type of connection analyser that could be a real asset to an administrator is one that effectively recognizes unauthorized daemons. The implementations we looked at in this document either do not provide this functionality, or are very inefficient at it.

Kernel-based intrusion detection systems are very effective at preventing and detecting intrusions. They can be used effectively for protecting system binaries and configuration files. More advanced configuration is possible, but not recommendable for large scale deployment.

Altogether, Host-based Intrusion Detection Systems, if implemented and configured correctly, are a useful asset in system security.

Bibliography

[1] *Advanced Intrusion Detection Environment (AIDE)* , http://sourceforge.net/projects/aide/

[2] *AIDE Manual* , http://www.cs.tut.fi/ rammer/aide/manual.html

[3] *Analysis of LOKI2, Using mtree as a Forensic Tool, and Sharing Data with Law Enforcement* , A. Korty, 2003, http://www.giac.org/practical/GCFA/Andrew_Korty_GCFA.pdf

[4] *Anomaly detection library and articles* , http://www.cs.ucsb.edu/ rsg/libAnomaly/

[5] *Debian GNU/Linux* , http://www.debian.org/

[6] *FreeBSD* , http://www.freebsd.org/

[7] *IDS based on process behavior rating* , http://plusik.pohoda.cz/thesis/

[8] *Intrusion Detection evasion* , C. Del Carlo, 2003, http://www.giac.org/practical/GSEC/Corbin_DelCarlo_GSEC.pdf

[9] *Kevent(2)* , FreeBSD 5.3, 2000, http://www.freebsd.org/cgi/man.cgi?query=kevent

[10] *Linux Intrusion Detection System* , http://www.lids.org/

[11] *The MD5 Message-Digest Algorithm* , http://www.ietf.org/rfc/rfc1321.txt

[12] *Mtree(8)* , FreeBSD 5.3, 2004, http://www.freebsd.org/cgi/man.cgi?query=mtree

[13] *Multitail, a tool to tail multiple files at once* , http://www.vanheusden.com/multitail/

[14] *Nmap Security Scanner* , http://www.insecure.org/nmap/

[15] *Non-executable stack implementation in NetBSD*, http://www.netbsd.org/Documentation/kernel/non-exec.html

[16] *Evading non-executable stack protections* , R. Wojtczuk , 1998 , http://community.core-sdi.com/ juliano/non-exec-stack-problems.html

[17] *Phrack Magazine: Designing and Attacking Port Scan Detection Tools* , 1998, Volume 8, Issue 53, article 13 of 15, http://www.phrack.org/phrack/53/P53-13

[18] *PortSentry* , http://sourceforge.net/projects/sentrytools/

[19] *SecurityFocus: Intrusion Detection, Theory and Practice* , David Elson, 2000, http://www.securityfocus.com/infocus/1203/

[20] *Simple Event Correlator* , http://kodu.neti.ee/ risto/sec/

[21] *Scanlogd* , http://www.openwall.com/scanlogd/

[22] *Simple watchdog* , http://swatch.sourceforge.net/

[23] *SNORT, The Open Source Network Intrusion Detection System* , http://www.snort.org/

[24] *On the detection of anomalous system call arguments* , C. Kruegel, e.a., 2003 , http://www.cs.ucsb.edu/ dhm/publications/ids/kruegel03:syscalls.pdf

[25] *Intrusion detection system based on process behavior rating* , T. Pluskal , 2004 , http://plusik.pohoda.cz/thesis/thesis.pdf

[26] *Tripwire* , http://www.tripwire.org

[27] *US Secure Hash Algorithm 1 (SHA1)* , http://www.ietf.org/rfc/rfc3174.txt