File Integrity Checking

A thesis submitted in fulfilment
of the requirements of the degree

MASTERS IN SCIENCE (Computer Science)

of Rhodes University

by

Yusuf Moosa Motara

December 2005

Abstract

This thesis looks at file execution as an attack vector that leads to the execution of unauthorized code. File integrity checking is examined as a means of removing this attack vector, and the design, implementation, and evaluation of a best-of-breed file integrity checker for the Linux operating system is undertaken. We conclude that the resultant file integrity checker does succeed in removing file execution as an attack vector, does so at a computational cost that is negligible, and displays innovative and useful features that are not currently found in any other Linux file integrity checker.

Keywords: **integrity checking, signed binaries, file integrity, filesystem monitor, safe execution, whitelisting, trojan detection**

Acknowledgements

بِسْمِ اللهِ الرَّحْمٰنِ الرَّحِيْمِ

In the name of God, the Compassionate, the Merciful

We begin with the name of Allah, Lord of all Creation, He who knows all that is secret or open. All knowledge contained within this document that is true and useful is due to the guidance and grace of Allah, and all that is false or misleading is due to my own feeble understanding. We ask for the help of Allah in all that we do, for He is the most beneficent, the most merciful.

The supervisor of this thesis has been Barry V Irwin, whose suggestions have been most appreciated and useful. This thesis would not be of the quality that it is without his assistance.

I would also like to thank my proofreader and wife, Shehnaaz, for her proofreading and kind suggestions. May Allah grant her happiness and peace for all her days, and reward her greatly for her unstinting support.

Finally, the assistance of the German Academic Exchange Service / Deutscher Akademischer Austausch Dienst (DAAD) and the National Research Foundation (NRF) is gratefully acknowledged.

Conventions

In this document, the following conventions are used:

- Filenames and command-lines are printed in a `monospaced` font.
- Words that appear in the Glossary (Appendix A) are printed in ***boldface italics*** the first time that they appear in the text.
- Source code, variable names, constants (such as 5 or NULL), and configuration file contents are printed in a sans-serif font. Typenames (such as **int**, **size_t** and **struct imon_core**) are printed in boldface sans-serifand may (for clarity) occasionally have the "struct" part of the type omitted.
- *Italicized* text is used for emphasis or to draw attention.
- Where sequential steps are important, or points are referred to specifically in subsequent text, a numbered list is used.
- Where the order of points is not important, a bulleted list is used.
- Where points could benefit from being referred to via a descriptive name in subsequent text, a descriptive list with items consisting of **boldface** words and indented paragraphs is used.
- Chapters, Sections, and Subsections are numbered and nested.
- Synonyms for a word are given as a marginal note, with the word in question being underlined in the text. If the word is glossarized, synonyms are also given in the glossary definition.

text: argument, body, consideration, content, document, passage, paragraph, thesis

When referring to other parts of this document, the following conventions are used:

- Sections and subsections are identified by dotted-decimal number. For example, 4.2 refers to Chapter 4, Section 2; 3.1.4 refers to Chapter 3, Section 1, Subsection 4; and C.4 refers to Appendix C, Chapter 4.
- To refer back to a point in the most recent numbered list, the point number enclosed in brackets is used. For example, (7) refers to point 7 in the most recently-seen numbered list.
- To refer back to a point in a descriptive list, **boldface** is used. For example, **Golden braid** refers to the point that bears that title in a recently-seen descriptive list.

Contents

List of Tables

List of Figures

List of Code Snippets

Chapter 1

Introduction

In this chapter we discuss the problem that this thesis attempts to address, and give an overview of what the proposed solution is. We also discuss the scope of this research and provide a brief description of some applications of file integrity checking. The chapter is concluded with an introduction to our own practical contribution to file integrity checking.

1.1 Research

In this section we explain what the problem that we are addressing is, and what our proposed solution is. We end the section after discussing the focus of the research and the constraints upon it.

1.1.1 Problem Statement

File execution is a common attack vector that is exploited by ***trojan horse***s, ***rootkit***s, ***spyware***, and ***virus***es to compromise a system. The seriousness of this attack vector is underlined by Catuogno and Visconti in [10, 9]: they use the terms *strong intrusion* and *weak intrusion* to differentiate attacks on a system which are able to install themselves on a system and thereby be effective across reboots from those which are not able to do so. In their words, cited from [10, pp. 1–2]:

> Once [an intrusion] has been accomplished, the attacker has complete control of the system and access to all the data stored on the machine. Obviously, if for

some reason the machine is rebooted, the attacker has to start again. Moreover, the legitimate administrator of the system could detect the ongoing intrusion, kill the shell session and terminate the intrusion. If the software bug that allowed the intrusion has been discovered, the system administrator can install a new version of the network **daemon** and thus the attacker cannot repeat the same attack on the machine; instead, he has to find another weak daemon or exploit a weakness of the new version of the same daemon (which, unfortunately, most of the times is easy to do). We refer to this form of attack as a weak intrusion attack.

A more serious threat comes from an attacker that, once root privileges have been gained, tries to colonize the system; that is, the attacker tries to keep control of the machine across reboots. We refer to this kind of attack as a strong intrusion attack.

In this research, we examine file integrity checking as a means to remove file execution as an attack vector.

1.1.2 Proposed Solution

The content of a valid executable file may be differentiated from the content of an invalid executable file by comparing cryptographic ***hash***es of their respective contents to a list of known-good file hashes. The metadata of a valid executable file may similarly be verified by comparing it to known-good metadata of that file. The process of verifying a file's integrity in such a manner is called "file integrity checking", and our proposed solution to the problem of unknown executables is to verify them before execution is allowed.

hash: digest, finger-print, check-sum

We therefore propose the creation of an "ideal" file integrity checker to address the issue of unknown executable files on a system. If successful, this solution should close off the avenue of file execution as an attack vector and thus reduce any strong intrusion to a weak intrusion.

1.1.3 Focus and Constraints

Our focus is on creating a proof-of-concept implementation that uses the Linux operating system; however, the design of such an implementation should be portable to most other architectures. In this thesis we use many terms that are specific to the Linux and Unix family of operating systems but have analogues on other platforms.

We have chosen to work on Linux due to the ease with which it may be modified to suit our needs, and the existing framework [12, 13] for adding security features to a system that it provides. Linux is a popular, well-tested, developer-friendly platform: if a proof-of-concept implementation can be seen to work well on Linux, then the real-world applicability of this research is assured.

1.2 File Integrity Checking

The goal of file integrity checking is to determine whether a given set of files is valid; that is, to determine whether or not they deviate from a set of known-good metrics. In the case of many existing file integrity checkers (see chapter 2 for examples) these metrics imply not only that the file contents have not been tampered with, but also that the file metadata remains valid.

To demonstrate why it is important for file metadata to be examined as well as file content, we take the case of a file which has not had its contents tampered with at all, but which is assigned ***permissions*** that make it readable by everyone when it should not be: an example of such a file might be the password file on certain systems. Though such a file would be bit-for-bit identical to a copy that is correct, we would argue that its integrity has been compromised by the incorrect permissions that it has.

permissions: mode

Related to integrity checking is the concept of an ***opportunity gap***: that space of time between a file being altered and the file being tested during which the file is accepted as being valid. Some uses of a file integrity checker demand that the opportunity gap be made small or nonexistent, whilst other uses make no demands relating to it at all.

1.2.1 Blacklists and Whitelists

File integrity checking is a form of ***whitelist***ing. In this section we examine the difference between whitelisting and the more common ***blacklist***ing, using set notation to clarify exactly what is meant by each point that is made. At the end of this section, it should be clear that whitelisting is superior (from a security perspective) to blacklisting.

Using standard mathematical set notation (see Appendix B), if we define

B: the set of files that are "bad";

G: the set of files that are "good"

$\hat{B}$: the set of files that we know to be "bad" ($\hat{B} \subset B$)

$\hat{G}$: the set of files that we know to be "good" ($\hat{G} \subset G$)

U: the set of files whose status is unknown to us ($(G \setminus \hat{G}) \cup (B \setminus \hat{B}) = U$);

Ω: the set of all files that exist in the world;

F: the set of all files that we have access to;

then we can see that, accepting $G \cap B = \emptyset$, any file that exists in the world is either "good" or "bad": $B \cup G = \Omega$. The actual definition of "good" or "bad" can be made as broadly or narrowly as one likes – meaning as little as "the file is / is not known" or as much as "the file is / is not a virus" – and is unimportant in this discussion of the blacklist and whitelist logic; we leave the line between "good" and "bad" up to system administrators to delineate for various systems. What *is* important is acknowledging that, given knowledge about any particular file, it is *always* possible to say that it falls under the aegis of one or the other of these categories. Accepting this as true, it is possible to categorize any file that exists as one that we *know* to be "good", one that we *know* to be "bad", or one that we know nothing about ($\hat{B} \cup \hat{G} \cup U = \Omega$). It is also reasonable to say that of the files we have access to, the status of any given file (on a ***baseline*** system, at least) can be determined ($\hat{G} \cup \hat{B} = F \Leftrightarrow U \cap F = \emptyset$). There are some files we do not have access to ($\Omega \setminus F$), so we cannot remove these files from U and place them into either $\hat{G}$ or $\hat{B}$. Therefore, we cannot ensure that the status of every file is known to us, and we cannot make $\hat{B} \cup \hat{G} = \Omega$ true. Having laid out these definitions, we shall use them in this section to logically demonstrate the difference between whitelists and blacklists.

A blacklist is a list of that which is forbidden. In a software context, a common example of software that uses blacklists is anti-virus software, which uses heuristic analysis and a virus signature database to determine which files should *not* be allowed to execute. Each time that a new forbidden-content variant emerges, the signature database and/or heuristic routines must be updated to detect it. This leads to an interval between a threat appearing and the update occurring during which a system can be exploited, and (more importantly) also leads to a situation in which a threat must first emerge before it can be dealt with; in other words, anti-virus software cannot guard against future threats effectively. To put the matter in the terms used above, a blacklist must comprise and test all elements of B and ensure that $B \cap F = \emptyset$. Assuming the set of "bad" files that exist is more than the set of files we have access to $(B \not\subseteq F)$, and assuming that unknown files exist whose status might be bad $(B \cap U \neq \emptyset$, it is not possible to say that $B \cap F = \emptyset$ – and blacklisting can therefore not ensure that "bad" files are kept from the system.

Conversely, a whitelist is a list of that which is allowed – and anything that is not explicitly on this list is forbidden. File integrity checkers can be considered as systems that follow the principle of whitelisting in that they flag as "bad" any content that is not known-good – in contrast to anti-virus software which flags as "good" any content that is not known-bad. Once again, if we refer to set notation we find that if $F \subseteq \hat{G} \Leftrightarrow \hat{B} \cap F = \emptyset$ for a baseline system, then ensuring that $(B \cup U) \cap F = \emptyset$ becomes easy since we have access to all members of $\hat{G}$; after all, we have constructed F from $\hat{G}$! All files that are not in $\hat{G}$ are treated as either unknown or "bad"; relating this back to standard computer security terminology, whitelisting provides for a ***default-deny*** stance that is impossible to obtain using blacklists, which reflect a ***default-allow*** stance.

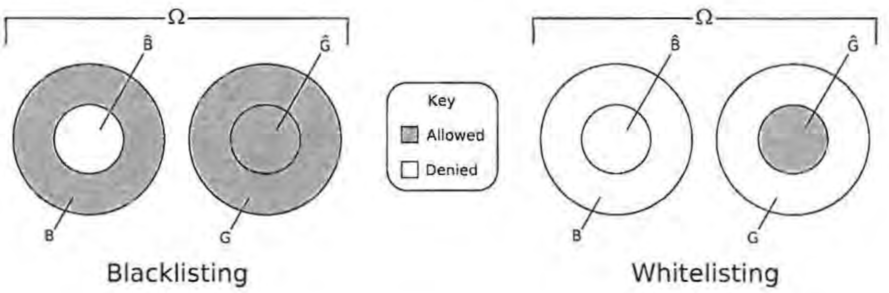


Figure 1.1: Blacklisting vs. whitelisting

Figure 1.1 shows the difference between blacklisting and whitelisting that we have described above. In this diagram the shaded area represents that which is allowed, and the clear area represents that which is not allowed. The familiar set names G, $\hat{G}$, B, $\hat{B}$, and Ω indicate which areas represent which sets; note that, just as has been defined above, the following relationships are shown in the diagram:

- $\hat{G} \subset G$
- $\hat{B} \subset B$

- $(G \setminus \hat{G}) \cup (B \setminus \hat{B}) = U$
- $B \cup G = \Omega$

From Figure 1.1 we can see that whitelisting allows a much smaller, more exact set to be specified than does blacklisting, and that this set includes no files from the unknown set.

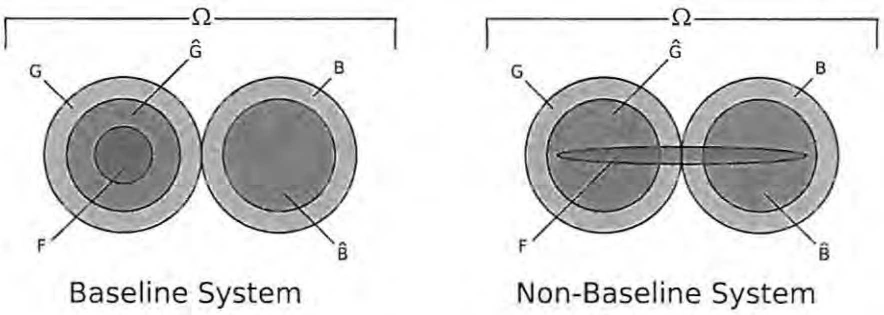


Figure 1.2: Venn diagram: baseline and non-baseline systems

Figure 1.2 can be compared to Figure 1.1: the same sets are shown, with the single addition of the F set. However, since Figure 1.2 is a Venn diagram, the shading indicates areas of overlap. On the left side of the figure we see a baseline system for which $F \subseteq \hat{G}$ holds true. On the right side of the figure we see a non-baseline system which, during the normal course of operations, has gained and lost some files, thus causing F to change. As we can see by comparing the baseline with the non-baseline, we can no longer be certain that $F \subseteq \hat{G}$ is true, and this makes it more crucial than ever to be certain that the files we are executing are known-good. We have already logically demonstrated that blacklisting cannot assure us of this, but whitelisting can.

From the above we can draw the conclusion that whitelisting has the benefit of being able to detect *any* unauthorized file on the system without needing to be specifically updated to detect such a file, and for this reason alone may be considered as superior to blacklisting. However, it should be noted that, during the execution of various programs, $\hat{G} = F$ may no longer hold true as additional files, placed into U, are created; for example, temporary files may be created by applications as backups, lock-files, and so forth. Restrictions placed upon these files should include denying them permission to execute until, should the system administrator choose to do so, they may be examined and placed from U into $\hat{G}$.

Example

Let us assume that we are faced with three executable files: good, bad, and unknown. As their names imply, the first is known to be an allowed file, the second is a known threat, and we know nothing about the last file. We also have the opportunity to run either anti-virus software (which uses blacklisting) or file integrity checking software (which uses whitelisting), both of which test files before they are executed.

Filename	Allowed by anti-virus?	Allowed by integrity checker?
good	✓	✓
bad	✗	✗
unknown	✓	✗

Table 1.1: Blacklisting vs. Whitelisting

Table 1.1 gives the results of attempting to execute the files. As can be seen, the only difference between the anti-virus software and the file integrity checking software is that they treat unknown differently. However, this difference is crucial!

Let us assume that we are using a file integrity checker instead of an anti-virus checker. If we find out at a later date that unknown is a harmless executable that should be allowed, then we must modify the file integrity checker to classify it as such, and we can say that the anti-virus software was correct in allowing it to execute. We have lost nothing but the convenience of being able to type in a new command or click on a new icon and have it instantly execute a program. However, if we find out that unknown is an example of ***malware***, then we have been saved from whatever malicious payload it might have carried.

Now let us assume that we are using anti-virus software instead of a file integrity checker. If we find out at a later date that unknown is a harmless executable, then we can be glad that the anti-virus software needed no modification in order to let it execute. However, if we find out at a later date that unknown is an example of malware, then we must modify the anti-virus software to recognize it, and we must accept that our system has now been infected with whatever malicious payload unknown happened to be carrying: we have lost our security, which some consider to be worth far more than convenience. Upgrading the anti-virus checker to disallow the execution of unknown is now a case of too little, too late: the damage has already been done.

Using both anti-virus software and a file integrity checker is pointless since the latter will allow through a subset of the former (see Figure 1.1). It therefore equivalent to using only a file integrity checker.

File Invalidity Checking: AntiExploit

If a list of "bad" files is kept instead of a list of good files, then what would be a file integrity checker becomes a file invalidity checker. An example of this is AntiExploit [55]. AntiExploit checks files upon access, much like a realtime file integrity checker, but tests to see if a hash of the contents of the file matches a known-bad hash. The documentation for AntiExploit recommends that the database of "bad" files used should be updated at least once a day.

This example is provided to show that it is not always obvious that files from G should be tested instead of files from B even in a project that, but for its choice of what to test, is very similar to certain other file integrity checkers.

1.2.2 Integrity Checker Applicability

Integrity checkers can be said to come in two varieties: periodic and realtime. *Periodic* file integrity checkers are those which periodically test files upon every n accesses, or after m minutes have passed, or at a certain time each day or week or month, or whenever the user requests that a check should be done. *Realtime* file integrity checkers, on the other hand, test files just before they are opened and/or (in the case of executable files) executed. There are some uses to which realtime integrity checkers are suitable and periodic integrity checkers are not; this section, however, covers three situations in which either type may be used.

Rudimentary Change Management

Being able to detect whether a given file is valid gives rise to the notion of protecting important files from alteration. In an organizational environment this can be invaluable, especially when access to files is granted to a number of people, but (for example) changes must be approved by committee [54].

If multiple "valid" measurements are kept for certain files, historical copies of those files can be validated as well; depending on the comprehensiveness of the data kept for measurement purposes, files may also be restored to a correct state either fully or partially. In an organization, this ability means that rudimentary change management and enforcement becomes easy. An example of file integrity checking software that is currently being used for change management is Tripwire (see 2.12 and [67]).

Malware Detection

Certain categories of malware (such as trojan horses, rootkits, viruses, and spyware) may choose to disguise themselves by either presenting themselves to the user as new, useful software or replacing existing software with a functionally-similar copy that also performs unwanted actions. A file integrity checker can detect these changes and indicate to the user exactly which files are affected. It is important to note that a file integrity checker is *not* a complete solution to the problem of malware, as malware may reside away from the filesystem entirely (e.g. in a machine's boot-sector, or accessed via email account across a network). Even if malware is resident on the system, it may be resident in a part of the filesystem that is entirely unsuitable to file integrity checking: an example of such a location is an email mailbox or newsfeed, which is frequently updated with messages and therefore almost impossible to generate a consistent and valid baseline image for. On a positive note, however, message-borne malware that relies on being *executed* as opposed to interpreted by an email client or newsreader may be stopped by a realtime integrity checker (see 1.2.3).

The ability of a file integrity checker to detect malware, preferably before it executes, is seen by [68, 6, 2, 53] among others to be an overriding concern.

Corruption Detection

A failing hard disk frequently leads to data integrity problems that an operating system does not pick up. This is because the operating system trusts the hardware to return the correct bits and has no way of verifying whether they are, in fact, the correct bits. Using a file integrity checker, one can detect both failing hardware and filesystem corruption by the results of an integrity check which a corrupted file would fail. This can be seen as an additional benefit of an integrity-checking system.

1.2.3 Realtime Integrity Checker Applicability

Realtime integrity checkers have properties that make them suitable for a greater (and sometimes different) number of applications than periodic checkers. For example, all periodic checkers suffer from the opportunity gap problem mentioned above. This does not matter much in the case of, for example, a document repository that is accessed at infrequent, scheduled intervals; in such a case, the files could be checked just before a scheduled access, and there is no opportunity gap. However, for the majority of change management and malware detection applications, it seems advisable for alterations to files to be detected as soon as possible. The problem becomes especially severe in the case of malware since running an unauthorized application even once can have very serious consequences – including being compromised and used for a distributed attack on other systems [21], which is an increasingly common occurrence.

Some applications that are only feasible with a realtime checker are discussed below. For these applications it is assumed that the integrity checker can not only detect an invalid file, but can also act upon that knowledge in some (possibly automated) fashion; this is in contrast to the more traditional approach of merely reporting that which is invalid, possibly via an occasionally-checked logfile.

Malware prevention and Honeypots

By testing each file before it is executed, we are able to do more than traditional malware detection (see 1.2.2): we are able to actively *prevent* an unauthorized executable from being executed. As soon as the file fails a check, we can take any number of actions, the most obvious of which is to both deny file execution and log the attempt. Of course, the exact action to be taken once the invalid file has been detected is up to the system administrator, as the next paragraph shows.

A ***honeypot*** is a system that exists to be attacked, and to log and/or perform analyses of attacks as they are in progress; we examine it as a form of malware prevention that takes a non-preventative action. In [50], we examine the digital forensics impact of a realtime integrity checker. Assuming that all valid files on a machine are registered as such, it is easy to detect any content that is invalid. This ability is invaluable in a honeypot situation since a common strategy used by malware involves removing the original executable malware file during or after execution. Through realtime detection of such an "unauthorized" file on the system, action such as making a backup copy, logging the behaviour of the executable, dumping executable images and/or sandboxing the executable can be taken.

Defined Execution Profiles

By allowing only certain executables to run on a target machine, and denying the execution of other executables, an organization is able to make use of a generic set of executables for all machines, a concept that is explored briefly Van Doorn, Arbaugh and Ballentijn in [68]. We define the ***execution profile*** of a machine as that set of executables that are recognized as valid by the realtime file integrity checker, and create execution profiles for various people within the organization. These profiles have nothing to do with the level of access granted to a particular user: they apply to all users of a system irrespective of any privilege systems at work. Due to the nature of a realtime integrity checker which needs to have a baseline to compare a file against, execution profiles are also only able to deal with "real" files, a defined series of bytes that have a specified beginning and end, and may therefore not be used to mediate access to "virtual" files such as may be found in the Linux `/dev/`, `/proc/` and `/sys/` filesystem hierarchies since the content of these files resides exclusively in kernel memory.

Another way to look at the same idea is to think of perspectives into a machine. We can see the machine as the entire "world" of possible executables, and those executables allowed by a certain profile are the "perspective" that that profile provides to the world. The word "perspective" is used to reinforce the analogy of having a vast space of possible views, but being limited or constrained by perspective to only see one or some of them.

Figure 1.3 shows an example set of execution profiles for a university environment. The entire set of possible executables is shown by the the largest circle; each circle within it depicts a particular execution profile. From this diagram we can see that laboratory machines are only allowed to execute a certain set of applications, as allowed by their execution profile; a lecturer's machine can execute all applications available to students, and a few more; an administrator's machine can execute some applications of a lecturer or student, and some applications that are not available to either lecturer or student; and a mathematician's machine cannot execute any of the applications associated with the other three profiles, but can execute a set of applications that no other machine can. Importantly, the number of executables on each machine is exactly the same: this means that all lecturer-machine executables

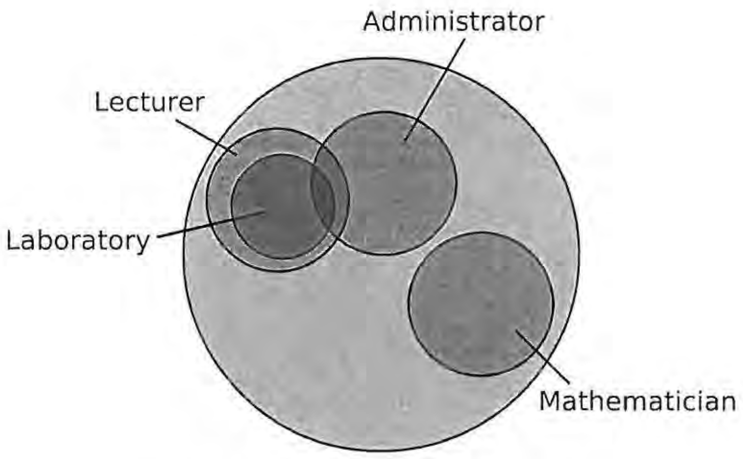


Figure 1.3: Venn diagram: execution profiles

are available on a mathematician-machine, but they cannot be executed because they are not within the execution profile of a mathematician's machine. An implication of this is that defining execution profiles could reduce the time spent testing various combinations of applications for unfavorable interactions: only one set of executables need be created for which many execution profiles could exist.

It should be noted, however, that this technique is *not* suitable for those machines on which new executables are developed: each executable would have to be validated before being run, leading to an increased development time. In other words, on such a system the definition of what constitutes a valid "baseline" system would be continually changing. The alternative to validating each executable would be defining certain names, paths, users or groups which would not be checked; however, these can easily be spoofed or taken advantage of, rendering the point of execution profiles moot.

Licence Compliance

In any large organization it is difficult to ensure that the licence requirements of software are being adhered to [23]. In the event that an organization does not have a good idea of which software is running on their machines, it cannot know whether it is in compliance with restrictive[1] software licenses.

A realtime file integrity checker that denies the execution of any unauthorized

[1]we use this term in the non-pejorative sense to mean licenses which restrict the duplication of software

executable is one way of ensuring that machines within an organization are kept entirely in compliance with software licenses. In contrast with execution profiles, this use would typically allow anyone access to all binaries on the system, but stop unauthorized executables from running. In turn, stopping unauthorized executables from running would discourage the introduction of pirated software onto systems: whilst such software could still be introduced, it would fail to execute, rendering the point of introducing it null and void.

At present, many organizations that bear the brunt of a software audit find it difficult to prove that none of their machines has placed them out of compliance with their licensing agreements [23]. The most common method of ensuring this once a software audit has been started is for each machine to be physically visited by and checked by an individual – an extremely time-consuming and expensive task! Having a realtime file integrity checker installed on each machine could reduce the cost of a software audit to virtually nothing as an organization could simply point out that running unlicensed copies of software on certain machines is not possible, and that their machines must therefore be in compliance.

Lockdown

As covered in our previously published work [50], a response to a perceived risk might be a *lockdown* of the machine. This would mean that only known-good tools, as determined by a realtime integrity checker, would be allowed to run. Importantly, the lockdown of a machine need not affect vital services running on the machine, though it may affect casual users who do not have their executables registered as valid. Another way of understanding the idea is to think of it as an integrity checker that runs all the time, and applies a strict policy some of the time.

Lockdown may be viewed as one step away from digital forensics (see 1.2.3) in that ***unauthorized code*** may (in a lockdown state) be sandboxed, have its behaviour traced, and so forth; however, an important difference is that lockdown is a temporary state enacted on a production machine that increases the security of the machine until a system administrator can determine whether or not there is a threat. By contrast, a honeypot would be "locked down" permanently and would most probably gather data on unauthorized executables instead of denying access to them.

1.3 IMon

In this section we introduce IMon, a realtime file *I*ntegrity *Mon*itor which forms the practical basis of this thesis. We lay down the rationale for why it has been created, what goals it seeks to achieve, and the scope that the project covers. Details of research done previously on the subject of file integrity checkers are discussed in chapter 2.

1.3.1 Rationale

Though many file integrity checkers do exist, we consider them to be flawed in certain respects, which are discussed in chapter 2. As yet, we have found little evidence of an attempt to see beyond the implementation of a file integrity checker to the rationale *behind* such an implementation. This thesis seeks to address that lack and follows a design process that keeps implementation in line with design goals, with each design decision and trade-off being well thought-out and motivated by a perceived or real need rather than being apparently ad-hoc.

Though 1.2.2 and 1.2.3 make it clear that a file integrity checker may be created for a number of purposes, we have chosen to create a security-focused system. In securing the machine from unauthorized code and thereby addressing the problem mentioned in 1.1.1. Furthermore, a secure system of the described type may be modified to be used for other purposes (such as creating defined execution profiles); however, in order to select and rank characteristics that are desirable for this project and in order to compare it meaningfully against the majority of file integrity checkers, which position themselves in the security field, we submit that this is a reasonable starting point.

1.3.2 Goals

As expanded from our paper [49], there are several desirable goals or attributes that every file integrity checker should strive for. In no particular order, these are:

Comprehensive checks Every possible aspect of a file should be validated, from its contents to its related metadata such as timestamps and permissions. The more comprehensive the checks made are, the less chance there is of an unauthorized file being able to pass an integrity check. Once again, it bears emphasising that file integrity is not only about the file contents: it is also about the meta-information of the file, such as owner, timestamp and permissions.

Comprehensive checking implies that not only should every possible aspect of a file be tested, but that every executable file should be tested as well. Many existing file integrity checkers only test certain executable formats, ignoring (for example) scripts that could potentially do a large amount of damage; see [6, 3, 75, 10] and chapter 2.

Automation An integrity checker should not depend on being run by the user; indeed, the slow reaction time of even technically-minded users when it comes to security matters is a cause of concern [4]. In the same vein, a tool that requires user input before action is taken is essentially useless until a systems administrator intervenes – leading to a time lapse between detection and action which may well be exploitable. Examples of this are the Code Red, Slammer, Nachi/Blaster and Sasser ***worm*** epidemics which would never have taken place had systems administrators applied the patch that had been readily available for a number of weeks [4]. The perception seems to be that security is a non-essential operation, and that running security checks may be skipped if one is short on time or simply forgetful.

It is also important that the action taken once an invalid file is found should be automated, rather than depending on a user scanning through logfiles to find it. The action could be as simple as denying access to the file or emailing the system administrator, but if *some* action is not taken then the enforceability of any policy related to integrity checking is reduced or eliminated entirely.

Relevance The problem of automation is taken a step further if the tool used provides copious amounts of output for a human to sift through. Once again, this places the burden of processing on the human – and whilst computers rarely make mistakes, humans are prone to do so. Missing a detail in a mass of irrelevant information is easy. The conclusion that we can draw from this is that the output of an integrity checker should be clear and unambiguous or, at the least, easily machine-processible so that exceptional situations can be found quickly.

Self-protection If files on the system are being modified, it is both prudent and reasonable to assume that an attacker has gained privileges that he should not have. In such a case alteration of an unprotected integrity checker database or other crucial files – such as the integrity checker program itself – is trivial, and it may be that the checker then does more harm than good in providing the false security of assuring the administrator that all is well when all is not very well at all!

Realtime checking An integrity checker that only spots untrusted executables after they have potentially been executed is less useful than one which can detect an untrusted executable before it has been executed: the primary difference is that the latter reduces the chance of damage being done to the system. In fact, the latter may prevent a system compromise entirely in the case of an executable needing to be run in order to breach security and gain unauthorized capabilities.

Related to this point is the fact that checking files periodically leads to an opportunity gap for an attacker, who has the amount of time between checks to do as he would like on the system with no fear of detection. Importantly, this can lead to the compromise of related systems in a networked environment, after which detecting and fixing the damage caused on the original system still leaves the network vulnerable to outside influence.

Maintainability As new vulnerabilities in programs are discovered, or as new versions of a program come out, it should be easy to make the integrity checker recognize the new version as valid *and* recognize the old version as invalid. A failure to do the former can lead to a denial of service as trusted executables are not allowed to execute, and a failure to do the latter leads to a situation in which a new version of a program may be replaced by an old (and potentially flawed) version, leading to an exploitable machine. Both situations represent a failure of the integrity checker.

Ideally, upgrading the system should require little or no user interaction, and should be able to be done without requiring machine downtime.

Efficiency A periodic file checker that is inefficient is a burden to the system that encourages the administrator to ignore it; a realtime file integrity checker that is inefficient is far worse. Statistics cited in [6] and [75] show that the slowdowns that at least two realtime checkers impose on a system are prohibitive for any production system resulting in either minimal checks or non-use. In order for a security measure to be accepted by users, it should affect their day-to-day activities as little as possible: slowing binary execution down by an order of magnitude or more [6, 75] is certainly not in keeping with this principle.

Efficiency also leads towards a secondary benefit, transparency. An efficient implementation is almost indistinguishable from a system that has nothing "extra" running on it at all. In the case of a honeypot, it is of critical importance that an intruder does not know that the system is examining his movements until it is too late for him to do anything about it. Transparency through efficiency aids in this effort. IMon aims to be efficient enough to achieve the benefit of transparency.

It is useful to rank these qualities in order of importance to our project. Should there be a trade-off to be made during the design process, this will allow us to make such compromises in an informed fashion. We therefore rank them by importance, with reasons, as follows:

1. **Comprehensive checks** encompass the fundamental purpose of a file integrity checker, and as such this quality is ranked as most important.
2. **Realtime checking** deals with *when* file integrity should be tested. Since an opportunity gap leads to the checker being bypassed, we consider this to be very important.
3. **Efficiency** is very important for the system to be acceptable to users and transparent. As we are attempting to create a model file integrity checker, it seems important that we ensure its operation is as efficient as we would like it to be.
4. **Automation** is required for some uses (execution profiles, honeypots) but not necessarily for others (malware detection, change management). We consider it to be of middling importance as some uses would not be possible without it.
5. **Relevance** may be ranked as slightly less important than automation. We may be able to tune the output more thoroughly in a future release of the software, and need not focus our attention on it at present.
6. **Self-protection** is also a good feature, but it is neither required for all uses of the file integrity checker nor is it central to the fundamental operation of a file integrity checker. It is of little importance in our design and may be "tacked on" at a later stage.

7. **Maintainability** would be a good feature to have, but it is not necessary for the system to function. It is of little importance when creating a proof-of-concept best-of-breed system.

1.3.3 Scope

The deliverables of this project are

- A thorough investigation of current file integrity checkers, leading to the implementation of a proof-of-concept ideal file integrity checker.
- An explanation of the design decisions made during the implementation of the IMon checker that is both in-depth and comprehensive, exploring all reasonable alternatives.
- An evaluation of the implementation in terms of security, performance and applicability.

1.4 Structure

Chapter 2 presents prior work in the area of file integrity checkers. In chapter 2, we examine basic features of file integrity checkers and then look at selected file integrity checker projects, extracting features and lessons from them as we go.

Chapter 3 builds on the previous chapter to create the best possible design. The advantages and disadvantages of various architectures and approaches are laid out in an accessible format, and both practical and theoretical considerations are discussed. As far as possible, all major aspects of file integrity checker design are examined. The chapter ends with a discussion of performance-enhancing algorithms. By the end of chapter 3, the design of a best-of-breed file integrity checker is complete.

Chapter 4 creates a Linux implementation, called IMon, of the design created in chapter 3. All important parts of the implementation are discussed, but special attention is given to the implementation of the database used by IMon and the code that does file integrity checking. Sections that may be difficult to understand are accompanied by diagrams and code snippets. At the end of this chapter, a reader should understand the most important parts of IMon's inner workings.

In chapter 5 we evaluate the implementation created in the previous chapter. We examine whether the security that it purports to provide actually exists, test the performance of IMon, and discuss situations for which IMon is particularly suited or not suited. The chapter ends with a list of features that sets IMon apart from other realtime integrity checkers on Linux, and a summary of the evaluation conducted.

Chapter 6 evaluates whether the goals laid out in chapter 1 have been met. It lays out future work that can be built upon or using IMon, and presents the conclusions of the thesis.

Appendix A is the glossary in which we present terms and meanings for selected words and phrases. In Appendix B, we take a basic look at those parts of set notation that are useful to know in order to understand some of the text of this thesis. Appendix C contains the explained and abbreviated source code for a big-number implementation that was created as part of the implementation described in chapter 4.

Chapter 2

Prior Work

The previous chapter introduced the research area of file integrity checking and some of the theory behind it, including a justification of file integrity checking as a method of whitelisting. This chapter deals with file integrity checker projects and research that is directly related to file integrity checking. We start by providing the practical background of file integrity checkers, we give a summary of basic file integrity checker features, and then examine each file integrity checker and extract best practices as we proceed. Finally, we summarise the chapter and introduce the next chapter as a logical successor to this one.

2.1 Background

Integrity checking has roots and branches that spread to encompass code authentication projects such as Microsoft® Authenticode [26], secure boot architectures such as Aegis [74], access control frameworks such as SELinux [37], and hardware-based trusted computing platforms such as described by the United States Department of Defense in their Orange Book [15] and by Balacheff, Chen, Plaquin and Proudler in [5]. In a sense, all of these may be called "prior" work, and some (such as trusted computing platforms) may safely be termed alternatives to this project; however, in another sense, the details and goals of each of these is different enough that examining them will not result in a meaningful comparative background being given. Therefore, in this section we have only presented projects that are close enough to our own to allow us to incorporate innovative ideas from them and learn from their mistakes, and we encourage the interested reader to explore the roots and branches of this field of research by starting with the references included in this paragraph.

The prior work done in the area of file integrity checking is difficult to organize in any meaningful fashion. Some possible organizations are by similarity, thus providing the reader with a continuum of gradually-changing features; by specific features that we decide are "important"; or by architecture, relying on *where* file integrity checking takes place. All of these organizational schemas are, given that feature-sets overlap and any "continual" gradient of change is subjectively imposed, inadequate. Therefore we present previous work in the area ordered lexicographically by project name, and we provide a summary at the end of this chapter that will allow readers to quickly find a project that has certain features or is of a certain architecture.

It should be noted that despite the term "file integrity checker" being used for all of the projects explored in this chapter, it is not always the case that a project has been created *primarily* as a file integrity checker. For example, Radmind (see 2.9) was created primarily as a system for monitoring and correcting changes made to a set of machines. However, all of them function effectively as file integrity checkers, and it is in this light that they are discussed.

2.2 Basic Features

It makes sense to first understand file integrity checkers using a common set of characteristics, and then describe special features that set them apart. All file integrity checkers share certain features or characteristics. For example, all integrity checkers must keep track of "good" file metrics and must therefore store a baseline for a given file somewhere; however, *where* the file metrics are stored differs between file integrity checkers. It makes sense to list the differences in basic features of each file integrity checker (such as where metrics are stored) in one place for comparison and reference purposes, and this is exactly what we do in Table 2.1.

Each column in Table 2.1 is explained as follows:

Project name This is the name of the file integrity checker, or the name of the paper in which the file integrity checker is described (if the integrity checker itself is unnamed).

Architecture It is possible to encompass both where the integrity checker is situated as well as how it is designed by using the term "architecture". All

Project name	Architecture	Multiple metrics	Metric storage	Realtime/ periodic	Configuration granularity	Coverage	Performance limiter(s)
AFICK	Application	Yes	Database	Periodic	File	Any	No. of files, operating system
AIDE	Application	Yes	Database	Periodic	File	Any	No. of files, metrics used
CryptoMark	Kernel module	No	Same-file	Realtime on-execution	User	Any binaries	None
DigSig	Kernel module	No	Same-file	Realtime on-execution	None	All binaries	None
I^{3}FS	Kernel module	Yes	Database	Realtime on-access	File	Any	No. of files, policy used
Osiris	Client/Server	Yes	Database	Periodic	Directory	Any	No. of files
Radmind	Client/Server	Yes	Database	Periodic	File	Any	No. of files
Samhain	Client/Server or Application	Yes	Database	Periodic	File	Any	No. of files
Signed Executables for Linux	Kernel and interpreter patches	No	Same-file	Realtime on-execution	None	All	None
Tripwire	Application	Yes	Database	Periodic	File	Any	No. of files, metrics used
TrojanProof	Kernel patches	No	Database	Realtime on-execution	None	All binaries	No. of files
Veriexec	Kernel subsystem	No	Database	Realtime on-execution on-access	File	Any	None
WLF	Kernel module	No	Same-file	Realtime on-execution	None	All	None

Table 2.1: File Integrity Checkers: Basic Features

file integrity checkers that are discussed here can be classified as one of the following architectures:

- *Application*: the project is situated in ***userspace*** and carries out file integrity checks from userspace.

userspace: user-mode, user-level

- *Kernel module/patch*: the project is situated in ***kernelspace*** and carries out file integrity checks from kernelspace.
- *Client/Server*: the project is situated in userspace as a client/server system. Machines with files that require testing are clients, and run a client program; the database of file metrics resides on a server. Integrity checking may take place on the client or the server, depending on the project.

Certain systems may be situated in kernelspace but have a userspace component. Two examples of this is are WLF (see 2.15) and CryptoMark (see 2.5), both of which rely on userspace utilities to modify files so that they can subsequently be tested. Similarly, certain systems may be situated in userspace but have a kernelspace component: an example of this is Samhain (see 2.10), which may be run in "stealth mode" using a kernel module to hide its presence on a machine. We have classified systems by architecture in 2.1 without reference to any subsidiary utilities or optional components.

Multiple metrics Some projects use only a fingerprint or digest to check the integrity of the file's contents. Other projects check multiple metrics, such as the file size, timestamps, permissions, and so forth. This column indicates which systems provide at least the option of using multiple metrics, and which do not.

Metric storage Baseline metrics must be stored somewhere so that they can be tested against during an integrity check. The two options that present themselves are storing the metrics within the file to be tested and storing them apart from the file; we use the terms *same-file* and *database* to distinguish between these respective options. Note that "database" merely means a file apart from the one being tested, and does not imply a standard database format or database management system being used.

Realtime/Periodic At the beginning of 1.2.2 we explain what the difference between a realtime and periodic file integrity checker is. This column indicates which type of file integrity checker each project is. In the case of a realtime integrity checker, the event(s) that prompt an integrity check are also indicated:

- *on-access* means that the file is tested whenever it is read for any reason.
- *on-execution* means that the file is tested whenever it is to be executed.

Configuration granularity The granularity of configuration indicates the smallest entity that may be separately configured via the integrity checker. For example, if the granularity is *file*, then we can say that the integrity checker may be configured on a per-file basis; put another way, we can configure the integrity checker to treat certain files differently. If the granularity had been *directory*, then we would be able to treat certain directories differently, but not files within those directories. When no configuration or only global configuration is possible, we have said that the configuration granularity is *none*.

Coverage The coverage of a file integrity checker indicates which files it checks:

- *Any* means that any file, but not necessarily all files, may be tested. For example, an integrity checker that tests only files that have an entry in a database would be classed as having "any" coverage since it can test any file, but does not test all files.
- *All* means that all files are tested.

In addition, the type of file that may be tested could be limited. An example of this is DigSig (see 2.6), which tests every file that is in the native binary executable format of the Linux platform; therefore, DigSig's coverage is mentioned as "All binaries".

Performance limiter(s) If there are any significant considerations that limit the performance of a given file integrity checker, they are mentioned in this column. In the case of periodic file integrity checkers, performance is taken to be the time required for one integrity check; in the case of realtime file integrity checkers, it is taken to be the time required to test a single file. Since the integrity checking process of a periodic and realtime checker differ, this dual definition of "performance" is necessary.

For example, the periodic integrity checker Tripwire's performance is drastically reduced if one chooses to use two or more hash algorithms to test the content of each file instead of settling on a single hash algorithm; therefore, "metrics used" is listed as a performance limiter. All periodic file integrity checkers are affected by the number of files that must be tested; hence, this is a common limitation.

As an example of a realtime integrity checker with performance limitations, we use I^{3}FS (see 2.7). This integrity checker can be configured to generate a policy for every file that is created; using this sort of policy, a performance penalty is incurred whenever many files are created (by unpacking an archive, for example).

All performance data is gathered from papers or documentation written about the project or from [73].

2.3 AFICK

Information in this section is summarized from [22] and an examination of the AFICK source code. AFICK is downloadable from [22].

AFICK is an acronym for *A*nother *F*ile *I*ntegrity *C*hec*k*er. Written in Perl, it consists of three packages: a core tool that does the actual testing of files and provides a command-line interface, a graphical interface and a web-based interface. The syntax of the configuration file is flexible and simple. It permits the user to use regular expressions to exclude files from searches, use only certain tests on certain files, and much more. AFICK may be run in either update mode, in which case the database is updated, or compare mode, in which case differences between the current filesystem state and the database are noted. According to Gerbier [22], the performance of AFICK running on Linux is equal to or greater than the performance of AIDE (see 2.4), which is compiled to machine code; however, AFICK runs around five times slower on a Windows® system.

2.4 AIDE

AIDE, written in C, is the *A*dvanced *I*ntrusion *D*etection *E*nvironment. AFICK is largely based upon it, and all features of AFICK mentioned in 2.3 are present in AIDE. An interesting quotation from the AIDE homepage at [33] is worth discussion:

> Unfortunately, Aide can not provide absolute sureness about change in files. Like any other system files, Aide's binary and/or database can also be altered.

This important criticism holds true for all userspace file integrity checkers: without the surety that even the super-user account is unable to modify the integrity checker, it is extremely difficult for a userspace file integrity checker to come close to guaranteeing that it has not been compromised.

2.5 CryptoMark

Information in this section, unless otherwise referenced, is derived from [6].

CryptoMark is a system written in C that consists of two parts: a set of utility programs and a Linux kernel module called KernelGuard. The utility programs are used to digitally sign ***ELF*** binary files, calculating a signature for the executable portions and adding the signature to the file itself as an SHT_NOTE (see [66]) section. A signature consists of a 128-bit MD5 [56] hash encrypted using the El Gamal algorithm. ELF binaries so signed are then transferred to the target system that is running KernelGuard. Nothing other than the MD5 hash of the file is tested by KernelGuard, which tests file integrity from kernelspace upon attempted execution.

Configuration of CryptoMark consists of deciding which ELF binaries should be tested based on the user identity under which a binary will be executed. For example, one could configure CryptoMark to require every binary that will run in an administrator context (as `root`, to use Unix parlance) to have a signature. Due to the nature of the signing process, only ELF binary files may be tested. The performance overhead imposed by CryptoMark is on the order of ~10-12%.

In [6, p. 4], Beattie, Black, Cowan, Pu and Yang discuss the storage of file metrics, ending with the statement that:

> A possible compromise (which has not yet been implemented) would be to use embedded certificates for ELF files, and an auxiliary table for other executables.

This raises the important point that the use of a database *and* same-file signatures is not impossible – in fact, the two are complementary approaches. It is also true that the generality of a database comes at a performance penalty, but if we can find a way to reduce this penalty then the database option is certainly the more attractive one.

Another salient point is raised by Beattie *et al* in [6, pp. 5–6]:

> Note that if the secret key is stored on the same machine that is being protected, it is very likely that attackers who are able to become root on that machine

would also be able to steal the secret key, and thus would be able to certify their own binaries. For this reason we consider such a configuration unsafe, and do not recommend it.

Implicit in this is the understanding that the system should be difficult to compromise even when an attacker has gained the highest privilege level. Therefore, it should feature both some form of self-protection to safeguard itself from tampering and also not depend on cryptographic private keys being present on the system.

2.6 DigSig

Information in this section, unless otherwise referenced, is derived from [3, 2].

DigSig is a Linux kernel module written in C that checks for a digital signature in ELF binaries. It uses the bsign [61] utility to digitally sign ELF files, creating an SHT_NOTE [66] to hold data. The ELF file is tested within kernelspace upon attempted execution, and the only metric used to determine the integrity of the file is a cryptographic hash of the file contents.

DigSig requires no configuration file, though it does take parameters upon being loaded that determine its behaviour. Only ELF binary files are tested since the way signatures are stored makes embedding signatures within arbitrary files a difficult affair. DigSig uses a caching strategy to reduce the overhead imposed: if a file has not been written to in the interval between a previous test and the current test, then it is allowed to execute without verification. Only files on the local filesystem are candidates for caching.

Signature revocation (see 3.7.2) is addressed by DigSig through the creation of a revocation list containing "bad" signatures. This list is accessed as a hash-table with the first byte of the signature as the key, leading to reasonably quick lookup times as long as the revocation list does not grow to be too large.

The caching mechanism of DigSig provides a distinct performance boost, with Apvrille, Gordon, Hallyn, Pourzandi, and Roy noting in [2, p. 9] that once caching is enabled, "there is now hardly any impact when DigSig is used". We should certainly consider implementing a similar caching mechanism to improve performance in a file integrity checker that we design; however, as is pointed out by Apvrille *et al* in [3, p. 8]:

> Caching signature validations can be risky. We must ensure that an attacker cannot use this feature to cause an altered version of a file to be loaded without the (now invalid) signature being checked. In the simplest, case, a new file is copied in place of the validated file.

More complex attacks on a caching mechanism are possible and have been carried out successfully by Slaviero, Kroon and Olivier in [47]. On p. 8 of that paper, Slaviero *et al* note that:

> The caching problem is more difficult to solve. By introducing caching, security is weakened. Therefore if security is paramount, the authors recommend disabling signature caching.

2.7 I^{3}FS

Information within this section is derived from [53]. Component systems used by I^{3}FS are referenced as necessary in this overview.

I^{3}FS (*i-cubed-FS*), written in C, is the *I*n-Kernel *I*ntegrity Checker and *I*ntrusion detection *F*ile *S*ystem, developed for the Linux kernel. It is a system composed of a stackable filesystem, which means that it can be mounted "over" any other filesystem such as NFS [52, 60] or Ext2 [7, pp. 574–600], and in-kernel Berkeley databases [29] that store cryptographic hashes and policies.

I^{3}FS is configured through policies that are entered in a secure fashion, using a passphrase, and are stored within one of the in-kernel databases. Policies are used to determine which metrics of each file should be tested, and may be inheritable such that files created within a directory can inherit the policy of that directory. At runtime, ***ioctl***s may be used to modify the behaviour of I^{3}FS. The in-kernel databases are populated at boot time using a user-level tool to specify the files on the filesystem that contain the appropriate data. Any file that has a policy attached to it may be tested; however, for performance reasons, an option is included to test files every N times that they are accessed instead of every time. A cryptographic checksum of file contents may be tested on either a per-***page*** or whole-file basis.

At runtime, file checksums can be updated automatically by having the administrator set a flag after entering a password securely. This makes upgrading a system easier since checksums are recomputed on a mismatch whilst the flag is active.

The performance of I^3FS is, predictably, directly related to the amount of filesystem activity and the policy used. According to Patil, Kashyap, Sivathanu, and Zadok [53], a very IO-intensive workload can result in slowdowns of up to 4.5 times; however, a more "usual" workload results in an overhead of around 4% or less. The idea of implementing per-page checksums is an attractive one that can be used to increase performance of a file integrity checker if performance becomes an issue.

In [53, p. 3], I^3FS authors Patil *et al* raise the following point about why it is necessary to use a hash instead of simply relying on meta-data being changed when a file is altered:

> Checksumming different fields of the meta data of files helps detect whether important files have been re-written by malicious programs through the file system. Checksumming file data helps detect unauthorized modification of data possibly made without the knowledge of the file system. An example of this is a malicious process that can write to the raw disk device directly in Unix like operating systems.

We consider ***raw device*** access to be a very valid concern, especially if caching is used to increase file integrity checker performance, and must take precautions in our design to avoid it becoming an issue.

Patil *et al* "provide two modes of operations for I^3FS: one that allows updates and another that does not" [53, p. 5]. We believe that this opens a possible security hole: in auto-update mode, an attacker can replace an executable file with malware and have it accepted as valid in future. It is possible to avoid this by only engaging auto-update once the machine is effectively isolated from external attack vectors (by disconnecting it from a network, for example), but this is just as convenient as unloading I^3FS and reloading it with new databases: since the latter option is easier to implement, we question the importance of adding an auto-update feature at all.

In [53, p. 7], Patil *et al* state that:

> Since whole file checksumming is a costly operation, we provide an option for specifying the frequency of integrity checks in the policy. For performance reasons, one can set up a policy for a file such that it will be checked for integrity every *N* times it is opened, where *N* is an integer value. Every time a file with a policy is opened, we check if it has a frequency number associated with it. If yes, the counter entry for the file in the access database is incremented by one. When

the value is equal to N, integrity check is performed and the counter is then reset to zero.

We believe that including such a feature is a mistake, especially when one considers the problem of raw device access. It is conceivable that an attacker could modify a file via a raw device, then execute it at a point where the integrity checker is not testing the file for performance reasons.

2.8 Osiris

Information in this section is derived from [24].

Osiris is a system, written in C, that consists of a server and several clients. The server runs a management console program and a management application, and each client runs a background process. Checking is done on a scheduled basis, with the client sending filesystem data to the server for comparison purposes; checks are initiated at the behest of the server.

Configuration is done through utilities provided on the server, and the range of options provided allows for a flexible monitoring scheme to be put into place.

2.9 Radmind

Information in this section is derived from [14].

Radmind is a system written in C that consists of a set of command-line tools and a server. The tools run in userspace on each client machine and relay filesystem information to the server. The server contains a number of "overloads", each of which defines files to be monitored and managed on each client: a set of overloads geared towards a particular machine or machine configuration is called a "loadset". If the filesystem data sent does not match the appropriate loadset, then data is sent from server to client to ensure that it does. The correct metrics are stored on the server, and data representing the actual filesystem state is generated on-the-fly and sent across a secure connection for comparison. Testing occurs on a user-specified basis and is initiated by the client.

Radmind is configured via positive overloads, which specify files to be monitored, and negative overloads which specify files that should *not* be monitored. In much the same way as Tripwire (see 2.12), performance depends largely on the number of files that need to be tested and the computational expense of each metric tested, though network traffic and the current workload of the server may play a part in any slowdown.

In [14, p. 1], Craig and McNeal state that:

> Also of note is the degree to which filesystem integrity checking conflicts with these [large-scale configuration management] system management tools. Groups like SANS and CERT list filesystem integrity checking as one of the basic procedures that all system administrators should use to help secure their computers. However, if a cluster is running both a filesystem integrity tool for intrusion detection and, e.g., rsync for software updates, each time an update occurs, every machine will report a security event. For large clusters, these reports are noise in which real problems may be lost. In order to update the managed systems without triggering security events, the system management tool must be aware of (or integrated with) the intrusion detection tool.

We agree with Craig and McNeal, and conclude that it should be easy to make the file integrity checker upgradeable via a system management tool. If we use same-file storage exclusively, then the problem does not arise; if we choose to use a database either exclusively or as a complement to same-file storage, then the upgrade should ideally consist of replacing the database in a secure fashion.

2.10 Samhain

Information in this section is derived from [72, 73].

Samhain is a system, written in C, that may be used either as a standalone file integrity checker for a single machine or to create a larger network of monitored machines. If not run in standalone mode, Samhain can be configured to run as a server or client. As a client, it communicates with the specified server and obtains a database of metrics to test against; as a server, it listens on a specified port and sends across a database of metrics that is appropriate for the client. All interactions are initiated by the client. In standalone mode, file metrics are stored in a database on the filesystem which may (optionally) be signed; in client/server mode, file metrics are stored on the server's filesystem. Testing of files is done on each client machine.

Samhain can test files at specified intervals if run as a daemon; alternately, it can test files whenever a user requests it by either starting the file integrity checking executable manually or sending a signal to a running daemon.

Configuration is flexible and powerful, with inclusion rules, exclusion rules, recursion, and more. According to Section 5.15 of the Samhain user manual, Samhain may be set to alter it's scheduling priority, only read in a certain number of kilobytes per second (to reduce IO load), use MD5 [56] as a checksum metric instead of TIGER [1] (which is the default), and so forth; these alterations will decrease the performance of Samhain, whose speed is directly proportional to the number of files to be tested and the computational expense of the metrics to be tested against, but they help to reduce system load at the expense of speedy filesystem checking. Samhain also offers the ability to try and hide its existence by taking a number of steps, some of which are:

- Renaming all binaries
- XOR'ing all strings in the database, logfile and executable
- Steganographically hiding the configuration file within an image

Samhain also takes many overt steps to protect itself from being compromised, or to at least leave some evidence that it *has* been compromised in the event that this happens. Self-protection measures include digitally signing the Samhain executable as well as the database used to store file metrics.

2.11 Signed Executables for Linux

Information in this section is derived from [68], which we consider to be one of the seminal papers in this research area.

The paper describing this implementation of digitally signed binaries for the Linux operating system does not give it a name; for the sake of notational convenience, we shall refer to it by the moniker SEFL. SEFL, written in C, consists of a set of utility programs used to digitally sign binary files and a kernel module that tests the contents of a file from kernelspace.

Through a process of delegating testing of script files to interpreters, which are themselves verified binaries, SEFL is able to verify any executable content, whether script or binary, as valid or invalid. SEFL also uses a cache of verified signatures as a performance-boosting enhancement: binaries that have not been altered in the interim between a previous check and this one are simply allowed to execute with no testing required. According to Van Doorn *et al* [68], the performance of the system without the cache reveals a slowdown of between approximately 1.7 and 25 times; however, with the cache the slowdown is completely removed: in fact, in some instances the executable appears to load *more quickly* with SEFL+cache than without it. Our suspicion is that this effect is due to timing inaccuracies.

As quoted from [68, p. 2], van Doorn, Ballintijn and Arbaugh created SEFL for two reasons:

> In the design of our system we were primarily focused on providing the following two integrity guarantees:
>
> - Prevent the modification of authorized executables, and
> - Prevent the addition of unauthorized executables.

They succeed in both of these aims, and their design is clear and clean enough that a reimplementation should not be difficult. Van Doorn *et al* also raise issues of signature revocation, interpreters, and script redirection that few other papers deal with and that are important to consider. Despite this, we do not consider SEFL's design to be ideal. The project lacks self-protection mechanisms and relies on userspace interpreters being modified to verify scripts. This latter criticism is the most difficult to overcome since it leads to a need to change every interpreter on a system to enable them to verify scripts. Other criticisms are that SEFL may not be able to handle the runtime loading of executable code via functions such as `dlopen` (see 5.1.5) since such calls are not handled by the normal execution path of the kernel, and that the caching mechanism is incomplete in that it cannot easily handle both native binary executable ELF [66] files and scripts.

Our design must, if possible, overcome all of these limitations.

2.12 Tripwire

Information in this section is derived from [67, 30].

Tripwire is written in C++ and is configured through a policy file consisting of which directory trees or files to test and which not to. Any file may be tested.

Running Tripwire tests all the files specified in the configuration file, and is thus computationally expensive; how computationally expensive it is depends on the metrics tested against and the number of files tested. Tripwire is well-known in the domain of file integrity checking, and is included here for the sake of completeness; the features of AIDE (see 2.4) are very similar.

2.13 TrojanProof

Information in this section is derived from [75].

TrojanProof is a set of patches to the OpenBSD operating system kernel that takes its name from the ability of a real-time file integrity checker to stop trojan horses from executing. It is written in C, and consists of a set of patch files for two reference implementations, one on FreeBSD and the other on OpenBSD. The only metric tested is a cryptographic hash of file contents. Hashes are stored on a filesystem in a database which is tested to ensure that it is effectively read-only.

TrojanProof requires no configuration and the action taken by it – whether to log attempts, do nothing or deny execution – is dependent on the ***securelevel*** of the operating system. According to Williams [75], the performance of TrojanProof is not very good, with 2-40% of processing power being dedicated to calculating signatures during a series of workload tests.

Williams in [75, p. 4] poses the question: "Does the ongoing massive increases in CPU processing power and memory bandwidth mean that the cost benefit ratio of calculating and comparing a digital signature for each and every invocation of an executable or script file is acceptable", and answers it by saying in the next paragraph that "The decision must come down to the the [sic] cost of having the information that an attacker has got far enough to tamper with executable files versus the cost of not knowing".

As researchers within the security field, we would agree that maintaining system security is worth almost any price. However, we also understand that having a large performance impact makes users unwilling to use the software, and that (as 2.11 demonstrates) the performance impact is an artifact of design and not something that must simply be accepted by a user. Therefore, the most important lesson that TrojanProof provides us with is that unless our integrity checker is designed with performance in mind, it could easily become prohibitively computationally expensive.

2.14 Veriexec

Information in this section is derived from [63, 46, 43, 44, 45]. Of these, [63] seems to be the most recently-updated and reliable.

Veriexec is a NetBSD operating system kernel subsystem that takes its name from the concept of *veri*fied *exec*ution. Written in C, Veriexec tests the file integrity if the file is in a list of "fingerprinted" files. Veriexec may be configured to allow or deny execution by modifying the strictness of checks via a ***sysctl***; though its coverage is listed as "any" in Table 2.1, at the strictest level it would be listed as "all" since execution of files without fingerprints is denied.

A caching strategy is used to increase performance: if a file has not been modified since it was last tested, it is allowed to run with no verification step. Verification occurs at both a whole-file and per-page level, with a comparison fingerprint for each page being created when the entire file is being verified before execution.

The configuration file of Veriexec allows one to specify whether a file may be accessed directly, indirectly, or only in a non-executable fashion. An interesting feature of Veriexec is its ability to enforce indirect execution such that an interpreter cannot be run standalone, but must be run in order to interpret a file. As far as we are aware, Veriexec is thus far unique in its ability to enforce indirect execution.

indirect execution: interpreted execution, script execution

From Veriexec we can get a better understanding of how best to deal with interpreters and indirect execution. The project is mature; however, it still does not feature many self-protection features, which leaves it open to tampering if an intruder does gain privileges that he should not have. For example, the database is unprotected and unsigned, which means that adding to it or replacing it should not be difficult. Furthermore, the fact that files can be added to the list of authorized files at runtime is (in our opinion) a convenience that could lead eventually to a compromise: unless the strictest checking is being done, a compromised superuser account can add any executables to the valid list. Lastly, Veriexec only checks file contents and not file metadata, which means that there are integrity violations such as a file with incorrect permissions that it will not pick up.

2.15 WLF

Information within this section is derived from [9, 10].

WLF stands for "Worldwide Loadable Format", and refers to an executable format handler introduced into the Linux kernel as part of the WLF project. WLF consists of a set of utilities used to manage digital signatures embedded within ELF binaries and a set of modified binary handlers, written in C, introduced into the Linux kernel. These handlers check both scripts and files with embedded digital signatures, following on from the work done in [68] (see 2.11). Signing is done in userspace and testing of file integrity is done in kernelspace. Testing is done on-execution, when all executable files should be seen by the kernel binary handlers before execution; however, in two cases the kernel binary handlers do *not* see the file before it is executed, and in these cases the approach used in WLF fails: when executable code is invoked from a shared object using the `dlopen` system call, and when an interpreter runs code passed to it on the command line (e.g., `python myScript.py`).

WLF requires no configuration. Any files for which a special binary handler has been registered may be tested. To decrease the overhead imposed on a system, WLF uses an innovative caching system that copies files to kernel memory and binds them to a device; every access to that file is then redirected to the cached copy instead of the original. This approach allows for the caching of both remote and local files at the expense of kernel memory; all files are subject to caching. However, we do not believe that storing entire files in kernel memory is a very scalable approach to take, and we are therefore hesitant to recommend this approach.

Signature revocation (see 3.7.2) is handled through a rather involved hierarchical system of signing directories as well as files: for more detail on this, see [10]. Performance tests done by Catuogno and Visconti in [10, p. 31] show that:

> As we can see, the loading of signed executables takes twice the time needed for loading the unsigned ones. The slowdown depends on both the size of the files and the number of dynamic objects that have to be loaded together with the main program. For example, the slowdown incurred into by rpm is the smallest as rpm has no shared objects.
>
> Anyway, we believe that the overhead shown is decidedly reasonable as it only affects the start of the execution process of an executables [sic] and is amortized over longer executions of the commands: the table refers to very short executions

where only the version number of the program is requested. Moreover such overheads should be considered as upper bounds, since measurements have been done without using the cache.

Catuogno and Visconti in [10, p. 36] comment on the fact that:

> The user's perception of executables, in a UNIX-like system, actually wraps different objects. Binaries and scripts are just invoked in the same way but the system handles them quite differently and, moreover, many differences exist also among execution processes of different binary formats. Even the same binary format can be differently handled according to the design of the program. For this reason, the design of a general mechanism of verification at run-time of executables is a non trivial task and probably it is not currently possible. We think that in a "pure" model, integrity verification is simply a phase of execution. Thus, first: an entity that performs the verification has to be always present and, second: verification has to be performed on anything is handled as an "executable".

They then go on to propose changing the way that the Linux kernel execution path works to arrive at a "pure model" of execution. It is important to note that no kernelspace file integrity checker for Linux has yet managed to solve the interpreter problem as Veriexec (see 2.14) has done on the NetBSD operating system; if our design manages to do so, it will be the first of its kind. In fact, Catuogno and Visconti in [10, p. 37] go on to say that:

> Handling execution of scripts and binaries in the same way is in some cases impossible. Consider, as an example, the `source` command of the `tcsh` interpreter. For the shell, invoking `source`, is like a "dynamic code loading", but from the kernel point of view, this operation appears simply as a user-level application that opens a file, hence, involving the kernel in this operation, within the "pure model", is clearly impossible.

It is our intention to show that not only is a pure model unnecessary, but that interpreted execution can be handled quite well under Linux with the same enforceability that is provided by Veriexec on NetBSD.

2.16 Other Work

2.3 through 2.15 represent projects or papers that have had an effect on this research area in terms of their features, innovative implementation, or design. The range of file integrity checkers covered thus far examines:

- The most popular commonly-used file integrity checkers
- All known file integrity checkers being actively developed about which information has been made available
- Academic papers and research projects that have had a significant impact
- Projects that use file integrity checking primarily as a management activity instead of a security activity

In other words, a reader who makes himself or herself familiar with the above-mentioned file integrity checkers can be assured that they have a good understanding of what the current state of the art is, and of the prior work that has been done in this research area.

However, there are far more file integrity checkers than those mentioned above. Other file integrity checkers include SOFFIC [69], OODDSS [16], YAFIC [58], Nabou [35, 36], Integrit [8], Veracity [34], GFI LANGuard [41], Ionx DataSentinel [28], IBM's Assured Execution Environment [48], and Xintegrity [27]. All of these projects (and this is by no means a comprehensive list!) either have little information available on them or contain similar features to the ones discussed above, and it would add very little to go into a discussion of them as well.

2.17 Summary

Whilst Table 2.1 captures important file integrity checker similarities and differences, it fails to capture all the differences in approach, extra features and design. For example, the stealth or self-protection features of Samhain (see 2.10) are not mentioned in the table. For a more in-depth look at the flaws and strengths of a particular file integrity checker, reading the section on that file integrity checker is highly recommended. We have evaluated each file integrity checker with a view to extracting features which could prove useful in our own design, and finding those parts of file integrity design that serve as a guide to what *not* to do.

In chapter 3, we create our design for an ideal file integrity checker. We refer back to this chapter as necessary, and base our design on what we have learned through the above examination of file integrity checkers.

Chapter 3

Design

The previous chapter saw us examining different file integrity checkers. In this chapter, we build upon what we have learned to create what we consider to be the ideal file integrity checker.

We use terms and examples that are familiar to a user of the Linux operating system, as this is the platform that we have selected as being open and extensible enough to form a good test-bed environment. However, despite the examples and terminology used having a Linux focus, much of the discussion within this chapter is broadly applicable to any currently-popular operating system, whether proprietary or open; furthermore, many Linux-specific terms have been defined within Appendix A, the Glossary.

3.1 Architecture

One of the most important decisions to be made, and one of the decisions that will constrain many of the other choices made, revolves around where to test file integrity from – in other words, where to situate the file integrity testing system. The obvious places that come to mind are within userspace, possibly within a client/server setup, or within kernelspace. We discuss the merits of each of these options below.

3.1.1 Userspace Advantages

The advantages of creating a userspace integrity checker are briefly:

Language choice Working in kernelspace, one is generally restricted to creating a system in the language favoured by the kernel developers (usually C or, less frequently, C++). This restriction is removed if one chooses to create the system in userspace: one can use the most suitable language for the system instead, even if this turns out to be an interpreted language (such as Python), a functional language (such as Haskell) or a framework language (such as C# or Java). For example, AFICK (see 2.3) is written in Perl.

Extended facilities A userspace program is free to take advantage of functionality present in any or all libraries available on the system. In addition, it is able to control or be controlled by other programs, and may thus be integrated easily into a broader system. An example of such integration is the scheduling of integrity checks using the scheduler that accompanies or is part of an operating system. Extended facilities also allow the programmer to work at a higher level than would otherwise be possible, building on what already exists instead of having to code it "by hand". For example, Osiris [24] uses the OpenSSL [11] library to provide a secure cryptographic base from which to work.

Portability A userspace program does not necessarily have to be tied to one particular platform. For example, AFICK [22] and Osiris [24] (among others) are able to test files on a number of heterogenous systems.

Locally Usable Though different file integrity checkers require different privilege levels to run correctly, there is nothing inherent to the concept of a userspace file integrity checker that says that the person running it must have more privileges than an ordinary user. Of course, a user may be able to run an integrity checker with only limited functionality since access to all files may not be granted, but at least he or she is able to use the tool at will.

Ease of maintenance Working within userspace, it is easy to modify a file integrity checker without risking damage to the entire operating system and/or machine. This is because if a critical failure occurs, the operating system can simply shut the program down with no further adverse effects. The situation is far different when working at a kernelspace level, where memory protection may be minimal or nonexistent and triggering a bug, or entering an infinite loop, could be disastrous for the entire system. In a similar vein, no special development environments using user-mode kernels (such as User-mode Linux [17]) are necessary for rapid development, and no understanding of the interactions of kernel subsystems are required either.

3.1.2 Userspace Disadvantages

In [53, p. 1], I^{3}FS authors Patil *et al* make the following comments about userspace file integrity checkers:

> There are three disadvantages of any such user-mode system: (1) it can be tampered with by an intruder; (2) it has significant performance overheads during the integrity checks; and (3) it does not detect intrusions in real-time.

All of these are expanded upon below, and one further disadvantage is noted:

Insecure In a worst-case scenario, an intruder has already broken into the system and has full super-user privileges accorded to him. Given this, it is not inconceivable that he is able to tamper with the userspace integrity checker or the database of metrics in order to give the impression of false security. Some userspace integrity checkers, notably Samhain [72, 73], take measures to prevent this or make it obvious that tampering has occurred; however, most userspace integrity checkers take few such precautions, and it takes a greater effort to secure a userspace integrity checker than it does to secure a kernelspace system.

Performance A userspace file integrity checker must typically test all files listed in its configuration once it has been started. This causes a performance impact on the system that may reduce the performance of other concurrently-executing tasks. However, userspace file integrity checkers may take steps to alleviate this problem; for example, Samhain (see 2.10) is able to limit its impact on a system by limiting the speed of data reading.

Periodic checking Running as a userspace program, it is impossible to check every file before, during or after it has been opened by some entity. It is not necessary for any process to inform the integrity checker that it is about to either execute or open a file. Therefore, the best that a userspace integrity checker can do is test files periodically in order to see if any have been modified. As mentioned in 1.2.2, this gives rise to an opportunity gap that an attacker could take advantage of.

A recent addition to the Linux kernel (present in version 2.6.13, released September 2005) is a feature called "inotify" [38, 39]. This feature allows a userspace program to monitor a large number of files and be notified whenever one of them is changed. Using inotify, or a similar feature of other operating systems, it may be possible for a userspace program to do realtime integrity checking. The ability to monitor many files from userspace is not common to many operating systems; however, we feel that it is certainly something that should be looked into more seriously.

Inaction Once a modified file has been found, a userspace integrity checker can do nothing but report it and/or stop it from being executed in the future by changing its permissions, deleting it, or taking other such measures. However, even if an unauthorized change in an executable has been detected whilst it is executing, another userspace program may not be able to stop it from continuing execution; and a program that is executing with super-user privileges may be able to defend itself against being terminated by the super-user account.

It is important to note that the lack of enforceability, periodic checks, and insecurity of userspace have already led to rootkits that allow trojan horse executables to be undetectable by userspace file integrity checkers. In [25], a simple method is discussed for creating a kernel module that effectively lets userspace read and write operations test a valid executable, but executes a trojan horse executable whenever the same valid executable is to be executed. In [62], a more advanced attack that accomplishes the same thing is discussed by Sparks and Butler. Clearly, the value of a userspace file integrity checker and the validity of its results are called into question by the fact that methods are known by which such an integrity checker can easily be circumvented.

3.1.3 Client/Server

Taking advantage of the benefits of **extended facilities** mentioned in 3.1.1, certain userspace projects (such as Osiris [24] and Radmind [14]) have been able to overcome the **insecure** disadvantage of userspace software. They have done this by testing filesystem data on another machine and/or storing one or more databases of metrics on another machine. This has the added advantage of allowing one machine to be used as a monitoring server for multiple clients; on the other hand, a client/server approach leads to the disadvantages of requiring a separate machine to monitor clients and being dependent on the network infrastructure being usable when tests are due to be conducted. A further disadvantage is that network communications may be disrupted by a network-based attack, leading to an additional weak point that an attacker can exploit.

3.1.4 Kernelspace Advantages

The advantages of creating a kernelspace checker are, briefly:

Realtime checking Since all access to files must go through the kernel, it is the perfect place from which to test the integrity of files. Importantly, only those files which are accessed are tested, and (depending on the way the integrity checker is set up) not even all accessed files need to be tested.

Actionable From kernelspace a system may have the power to either grant or deny access to a file depending on an integrity check. This allows for not only the *detection* of unauthorized files on a system, but also the *prevention* of access to such files. In some cases, it may even allow remedial action to be taken: for example, if the permissions on the file are incorrect, then they can be corrected by the integrity checker. Of course, there is always an option to simply log an unauthorized file either as the sole course of action or in addition to denying access.

More secure It is more difficult for even an entity with superuser privileges to affect a kernelspace system. Enforceable security measures that could be taken include denying all access to kernel-related files on a filesystem (so that they cannot be replaced or compromised), ensuring that security modules cannot be unloaded, denying write access to kernel memory, and testing all modules before they are loaded to verify their integrity. Importantly, security measures taken within kernelspace are enforceable, whereas those that are taken within userspace may not be.

3.1.5 Kernelspace Disadvantages

Some disadvantages of a kernelspace integrity checker are:

System slowdown Assuming that file integrity is tested in realtime, each execution and/or file access attempt may be delayed as the file is being tested. This contributes directly to a perceptibly "sluggish" feeling and may make a system unusable: the performance overhead of CryptoMark (see 2.5), for example, is prohibitive. However, clever design strategies can ameliorate the incurred slowdown, as shown by SEFL, DigSig, I^{3}FS and Veriexec (see 2.11, 2.6, 2.7 and 2.14 respectively).

Dangerous It is riskier to create a system within kernelspace because of the chance that bugs within the system could wreak lasting damage upon a machine. Subtle infinite loops, unguarded critical sections and the interactions of various other kernelspace systems with one's own system need to be taken into account so that one does not deadlock a machine, cause extensive filesystem corruption, and so forth.

Fewer facilities From kernelspace, one cannot rely on functionality that exists within userspace libraries. If one requires such functionality, one option is to modify userspace libraries – assuming that one has access to the source code and is legally allowed to do so – and include their code within the kernel; another option is to reimplement the functionality required. Another drawback is that, as one is conceptually working at a low level of the system, it is difficult to integrate a kernelspace integrity checker into other systems and have it control or be controlled as is easy to do with a userspace program.

Arbitrary restrictions Depending on the operating system used to implement a kernelspace integrity checker, one could be subject to a number of restrictions that have little to do with the conceptual domain of file integrity checking. For example, one could find that C or C++ is the language that *must* be used, or that the way the overall kernel is structured may impose requirements on the system that one is developing. These restrictions are why Catuogno and Visconti in [10, pp. 36–37] have decided that handling native and interpreted executables in the same fashion is not possible.

Non-portable A kernelspace system is less portable than a userspace system; it is inherently tied to the architecture for which it is made, at least partially by the **arbitrary restrictions** mentioned previously. In fact, even successive versions of the same kernel may change inner workings to such an extent that re-working one's own code is necessary! If one wishes the effort to be portable, special precautions should be taken to not tie it too closely to any particular kernel subsystem or feature.

3.1.6 Decision

Though creating a userspace integrity checker has many advantages, the few disadvantages outweigh them by far. It is, on most systems, not possible to create a real-time userspace integrity checker, nor is it possible to create one that can always decisively act upon data even during a compromise; both of these attributes are extremely desirable for a file integrity checker, and (as noted in 1.2.3) there are many uses that a real-time integrity checker can be put to that a periodic integrity checker is simply unsuitable for. Furthermore, there are known techniques [62, 25] for subverting any userspace file integrity checker. This leaves us with the need to create a kernelspace integrity checker despite the numerous disadvantages of doing so.

However, by understanding these *a priori* disadvantages beforehand, we can ameliorate or even eliminate some of them through appropriate design. Some ways to do this from a design point of view are discussed below.

- Speed up frequent operations. Upon attempted access to a file, a record for that file must be found so that file attributes may be tested against known-good attributes. This means that a system needs to find a file record, do a comparison, and return a result. Any of these steps may be sped up:

  Finding a record Keeping records in a hash-table or similar data structure that provides good lookup times can make finding a given record efficient. A sequential search for a file record makes the entire system slower [43]. Another possibility is using a cache, as has been done by [3, 10, 68] to good effect. A cache and an efficient structure for lookups are complementary approaches.

  Doing a comparison Metrics (see 3.6) should be chosen to ensure that testing against them is not too computationally expensive, but is still sufficient.

  Returning a result The result returned should ideally be a simple variable that expresses a Grant-or-Deny decision. Returning a vast quantity of information that requires expensive processing either by the file integrity checking system or other systems is not acceptable.

- Use a "dummy" kernel. The dangers of kernel development can be eased by using a project such as User-mode Linux [17], which allows a kernel to run in userspace and has been created at least in part to address kernel development issues. A mistake made that causes the userspace kernel to become unstable has no effect on the rest of the machine. Similarly, one could use a virtual machine such as VMware to run an entire virtualized operating system without endangering the rest of the machine [51]. Both of these options make kernel development significantly less dangerous and (possibly) faster.

- Create a modular system. A system that is modular can address many portability concerns, is more extensible, and may help to circumvent arbitrary restrictions by creating abstractions that separate such restrictions away into implementation-dependent parts.

It is difficult to argue that we could create a userspace checker and ameliorate its disadvantages. The very nature of userspace means that realtime checking, and specifically on-access testing, is simply not feasible in most cases, and **inaction** can be partially remedied by granting the checking process super-user privileges – but there is still no guarantee that an intruder who has obtained such privileges himself would not be able to defend himself against such a process.

3.2 Dependencies

A firewall application takes a set of firewall rules, generally from a configuration file, and erects a firewall that reflects those rules. One of the more subtle ways of subverting a machine, then, is to supply the firewall application with a different configuration file that opens up holes within the security perimeter of a machine and, therefore, any protected network that lies behind it. A file integrity checker would not pick this up, assuming (as is a reasonable assumption) that unknown files are allowed to be read and written, but not executed (see 3.4.2).

In the above scenario we see that a file integrity checker has not managed to stop a system from being subject to a strong intrusion, though the degree to which the system has been affected is still far less than would have been possible in the absence of an integrity checker. One way to stop this from happening is to create a system of *dependencies*: files upon which other files rely. Using a specific policy (see 3.5.2), it can be made explicit that all files listed as dependencies are to be tested and read before any other file can be read. This allows us to reduce the danger of even the subtle form of strong intrusion mentioned in this section.

To continue with the above example, if we made the firewall configuration file a dependency of the firewall executable that must be read before any other file can be read, the problem disappears. Now it is impossible for the firewall to be started with any other configuration file; and, furthermore, since the integrity of the dependency can be tested as well, it is not possible to subvert the default configuration file either.

3.3 Interpreted Execution

Interpreted execution is handled differently on different platforms. Under Unix-like systems, "hash-bang" notation[1] is used to indicate the interpreter for a given file: for example, having `#!/usr/bin/zsh` as the first line of a file indicates that the interpreter of that file is the program `/usr/bin/zsh`. This gives rise to an interesting implementation problem of how to handle a chain of interpreters; for a solution to this, see 4.5.10. On other systems, the file extension may determine which interpreter is to be used; this is the case on the Microsoft® Windows® operating system. On yet other systems, such as BeOS, it is possible for heuristic analysis of the file to determine which interpreter should be used.

hash-bang: shebang

Frequently, interpreted files are treated as afterthoughts in integrity checker design; for example, the DigSig project is designed specifically to cater to native ELF files, and authors Apvrille *et al* "mainly hope to extend [their] work to protect Linux systems against malicious shell scripts" [2, p. 11]. However, interpreted files deserve more than an afterthought since they pose problems that native binary formats do not. Specifically, these problems are:

Runtime evaluation Many interpreted languages provide the ability to evaluate and execute statements provided at runtime. Allowing an interpreter read access to any file, including unlisted files, is problematic for this reason: there is a chance that the file could contain statements that would be evaluated and executed at runtime, thus defeating the purpose of integrity checks.

Standalone interpreters A *standalone* interpreter is one which is invoked without an associated script file to be interpreted. Such an interpreter would usually accept statements through its standard input stream and execute them. To allow standalone interpreters would be to allow the execution of unauthorized

[1]termed thus as it uses a "hash" sign, #, followed by an exclamation sign (called "bang" in mathematical circles), followed by an interpreter name

code, even though such allowance does not necessarily lead to a strong intrusion (see 1.2.2).

Transitive interpretation A *transitive* interpreter is one which provides an environment for other interpreters to work within. An example of a transitive interpreter on a Linux system is `/bin/env`, which executes a given program in a modified environment. Given the string "`/bin/env python`" as the interpreter for a file, an integrity checker must register `python` and not `env` as the interpreter. Given the same string on the command-line, the interpreter should be treated as being standalone.

The above problems can be solved by

- Disallowing the reading of any unknown file by any interpreter – unless an already-loaded and trusted script file is being executed. This solves the problem of runtime evaluation, since the only files which may be opened by an interpreter are known-good files.
- Ensuring that transitive interpreters somehow transmit the fact that they are being invoked as an interpreter to the real interpreter.
- Disallowing the execution of a standalone interpreter.
- Always testing the cryptographic hash of a file to be opened by an interpreter, regardless of any per-file flags (see 3.5.1) that may specify otherwise.

Our design, therefore, differentiates between files using a per-file flags (see 3.5.1), with one of the available flags being used to mark which files are to be considered interpreters. A file marked as an interpreter is subject to all of the problem-solving constraints mentioned above.

3.4 Configuration

Configuration of an integrity checker should be done in a manner that is convenient for the system administrator, and (at the same time) secure.

3.4.1 Inclusion and Exclusion Lists

A number of file integrity checkers such as Tripwire, AIDE and Radmind (see 2.12, 2.4, and 2.9 respectively) use the concept of inclusion lists, which list files or paths to be tested, and exclusion lists, which list files or paths that should not be tested. Table 2.1 shows these files as having a coverage of "Any".

The rationale behind exclusion lists is that some paths, such as the `/tmp/` directory, or certain files (such as spool files found in the `/var/spool/` hierarchy), are temporary in nature and difficult to generate baseline metrics for; therefore, these files should not be tested. We reject this rationale as being significant in our design decisions, and argue that exclusion lists only make it easy for an attacker to hide his files within an excluded path or under an excluded name.

The rationale behind inclusion lists is twofold:

1. If a certain subset of files can at least be determined to be correct – for example, core system utilities such as `ls`, `ps` and `netstat` – then, using these, the system can be restored to a "good" state after any intrusion. There is no need to check *every* file's integrity as long as we can test a core set of files.
2. Testing takes time and is computationally expensive. The fewer files there are that need to be tested, the less the impact of integrity checking on overall system performance. This is a greater consideration for userspace projects such as AFICK [22] and AIDE [33] than for kernelspace projects, unless (as used to be the case with Veriexec [43]) the number of files to be tested is a performance bottleneck.

Inclusion lists fit in well with the concept of whitelists as discussed in 1.2.1: they determine which files should be tested, and files other than those mentioned in an inclusion list should be subject to the default policy of the system. Exclusion lists exclude files from being tested; however, since our aim is to test *all* executables so that we can stop a strong intrusion, we have no need for exclusion lists in our design. Note that an exclusion list is *not* the same as a blacklist since it does not list anything that is considered "bad": it simply leaves certain files unmonitored, and therefore compromisable by an attacker.

Our design, therefore, uses inclusion lists but not exclusion lists to determine which files should be tested.

Regular Expression Matching

Regular expression matching allows for one to list the files to be tested by creating a pattern that captures variation within text. For example, the regular expression

```
/tmp/devices\.real\.[[:alnum:]]{6}$
```

would match any file within the `/tmp/` directory that began with the string `devices.real.` and contained six alphanumeric characters followed by the end of the entire string. A description of POSIX regular expressions, such as the one used as an example, may be found in POSIX 1003.1 [64], and a discussion on regular expression formats is outside the scope of this text. However, it is instructive to mention them as they are commonly used by integrity checkers – AIDE, Samhain, Osiris, and AFICK (see 2.4, 2.10, 2.8 and 2.3) use them extensively, for example – and are a powerful method of specifying many files, the exact names of which may not be known beforehand.

When used to exclude files from being tested, regular expressions open up holes in the security of a system due to this imprecision. When used to include files, regular expressions are a powerful tool – but a tool that cannot, assuming database storage, be used from the integrity checker itself (see 3.7.4). They may still be used in the generation of a configuration file, however, and it is in this role that we propose they are most useful in our design.

3.4.2 Accept or Deny?

The determination of whether to accept or deny a file's execution once it is found to be unknown or invalid should be made dependent on the purpose that the file integrity checker is used for. We foresee the most wide-spread use of the system as being a security tool used to monitor the integrity of the filesystem and prevent a strong intrusion. However, other uses have been mentioned under 1.2.2 and 1.2.3, and our design should leave enough room for any of these uses to be made a reality without substantial reengineering.

Bearing in mind that files are created and removed by various programs, creating a list of all files that could potentially exist on a system (with accompanying metrics) is a nightmarish proposition. Instead of doing this, we choose to deny the execution of unknown files but leave them readable and writable, to the extent allowed by policy (see 3.5.2).

3.5 File Attributes

This section discusses two ways to attach attributes to a file, and the semantic difference between them. Attributes are useful to describe what a file is intrinsically, and to describe its relationship to other files on the system. In 3.3 we have pointed to their use as one way to distinguish an interpreter from other files, and in [63] attributes are used to distinguish between types of files that Veriexec (see 2.14) handles.

3.5.1 Per-File Flags

Part of configuring an integrity checker is determining what should be tested for each file, and which capabilities should be allowed or denied for each file. Flags that may be set for each file provide a simple, extensible, flexible way to specify this type of information. Before dealing with the way that file flags can be specified, however, it is useful to examine *why* we might want to determine what should be tested; after all, is it not more secure to simply test all metrics for each file? The answer to this question may be best illustrated through the use of an example.

On a Linux system the file `/etc/ld.so.cache`, which holds data used by the ELF interpreter, may be recreated periodically. If we assume that the libraries on the system have not changed (and they should not have!), then the file is created each time with the same data, but different modification and change timestamps. For this particular file, then, the modification and change timestamps being altered is something that happens during the normal functioning of the system, and flagging this change whenever it happens would add little but noise to the system logs.

Therefore, to deal with files such as this, we must implement attribute ignoring: the ability to say that certain metrics on a file should simply be regarded as irrelevant, and therefore not flagged if they should change. Before a test is conducted, we should check whether the metric should be ignored or not, and act appropriately by ignoring the metric or testing it. File flags allow us to implement such attribute ignoring.

It should be noted at this point that a metric should never be ignored due to speed concerns since ignoring certain attributes has implications for the rest of the system. For example, ignoring the cryptographic hash of a file means that we can no longer be assured of what the content of the file is; therefore, a file that has its cryptographic hash ignored should not be readable by an interpreter if the problems discussed in 3.3 are to be avoided. Less critically, if the permissions of a file are ignored then the security of knowing that a given file is not executing as another user is removed. Therefore, ignoring attributes should never be done without full knowledge of what the security trade-off inherent in doing so is.

File flags may be used to add capabilities as well as to remove them. For example, in 3.8 we show how certain executables are allowed to add or remove modules, with the executables that are given this capability being specified explicitly using per-file flags.

Importantly, file flags may also be set and unset at runtime to indicate the status of a file; for an example of this, see 3.10.3.

3.5.2 Policy

Policy reflects the idea that an executable file (whether native binary or interpreted script) is not the same as a data file. An executable may, amongst other actions, read other files: in the case of an interpreter, or a daemon intended to read from certain inputs and write to logfiles, the set of files that may be read should be restricted to a known-good set. Policy is a way of expressing how a given executable file should relate to other files on the system.

In this discussion, we call the executable file the *main* file, and we say that it has *dependencies* (see 3.2). Some of the dependencies of a main file might be executable files themselves, and may therefore also have dependencies: for example, a script file could execute another script file. Nevertheless, we treat all dependencies of an executing main file simply as dependencies, and never as main files in their own right.

The following policies have been devised as being general enough to cater for the majority of main-file/dependency interactions:

deny_others This policy indicates that the main file should not be able to open any files for reading other than those listed as dependencies. It is suitable for executable files that should be restricted to a certain set of files, and be disallowed access to any others. An example of this might be a FTP server program which should not be allowed access to any files outside of the tree of files that it exists to serve to clients.

accept_after This policy indicates that, after opening all files listed as dependencies, an executable is free to open any file at all; if the file is known to the system, it is first tested. This is particularly useful for executables that read in configuration files, such as a SSH server, and should then be free to interact with any other file on the filesystem.

accept_anytime The most permissive policy, this is suitable for many applications such as `grep` and email-readers. The policy says that any file, whether known to the system or not, may be read by the executable; if the file is known, it is first tested.

deny_notfound This policy denies access to any file that is not known to the system. It is suitable for executables that might, for example, open any number of shared libraries, and ensures that those libraries that are opened will have their integrity verified.

It should be noted that only two of the policies, **deny_others** and **accept_after**, make specific reference to a file's dependencies. In all other cases, the dependencies of a file are not seen as important.

Another important fact to note is that all the policies above specify that a file should be tested if it is known to the system. Even the most permissive of policies (**accept_anytime**) will test a file if possible, but allow it anyway even if it does not. This behaviour means that making a file known through adding a signature to it or placing it in a database has the effect of ensuring that it is *always* tested before access to it is allowed.

Runtime policy assignment

None of the above policies needs to be explicitly set, in the case of many executables. Instead, a default policy may be set at runtime. What this policy is should depend on whether the file being executed is interpreted or not; see 3.3 for why interpreters must be treated specially. We recommend a policy of **accept_anytime** or **deny_others** for binary files: neither of these policies involves creating explicit dependencies for a file. For interpreted files, a default policy of **deny_others** is recommended unless one is certain that the interpreted file cannot take input from a source and interpret it at runtime.

Trust and Restriction

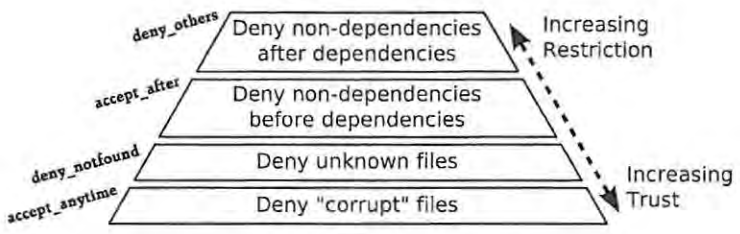


Figure 3.1: Policy, Trust and Restriction

Figure 3.1 shows the increasing restrictiveness of the policy options, with the upper layers building on the restrictions imposed by the lower layers. At the lowest layer (`accept_anytime`), only files which are in the database but fail an integrity check – in other words, those files which we are certain have been tampered with – may not be read or executed. At `deny_notfound` we add the restriction that only files we know about may be read or executed; these files are those that comprise the set $\hat{G}$ (as defined in 1.2.1). `accept_after` adds the restriction that certain files *must* be read before any other file is allowed; and `deny_others` tightens this restriction such that no file other than those explicitly listed as dependencies may be read or executed.

Looked at in terms of trust instead of restrictiveness, we can see that the files trusted to access almost anything are those to which the lower policies are applied. Conversely, the files to which the higher policies are applied are trusted to access only a known, limited subset of all files. The varying levels of trust give some indication of what a good default policy to set for a given machine is: if the machine is placed upon a vulnerable network, most files should be set to at least `deny_notfound` or `accept_after`, but if it is located in a secure area then the more trusting `accept_anytime` could make for easier administration.

3.5.3 Attribute Semantics

Both policy and per-file flags allow one to set attributes on a file that alter the way that the file is handled by the integrity checker. However, the relationship expressed by using one or the other is quite different.

Per-file flags express an "is-a" relationship that says that the file *is* an interpreter, or that certain attributes of it *are* changeable (and should therefore not be tested), or that it *is* capable of loading modules. In other words, per-file flags express some quality that is intrinsic to the nature of a particular file. Since an entity may have more than one quality, it is possible to set more than one per-file flag on each file.

Policy expresses the relationship between a single file and other files on the system. There cannot be more than one relationship between these; for example, a file cannot both be allowed to open any file (as per the **accept_anytime** policy) and denied access to files not in the database (as per the **deny_notfound** policy). For this reason, only one policy may be set upon a file. Policy does not express what a file *is*, but it does express how a file should be treated.

If in doubt as to whether to codify a file attribute as a policy or a per-file flag, consider whether the decision may vary with the system or environment that the file is within. If it may vary, then policy should be used to express the attribute; otherwise, a per-file flag should be used. For example, the `python` interpreter will always have the interpreter per-file flag set, and the `/etc/ld.so.cache` file will always have per-file flags set that indicate that its timestamps should not be tested: these attributes are inherent to what the files are, and will not vary depending on the system. However, on one system the file `/bin/test` may be given the **accept_anytime** policy, and on another it may be given the **deny_notfound** policy: both specify how it should be treated, and this varies from system to system.

In reality, the decision to codify a file attribute as policy or per-file flag may not always be as clear-cut as it has been made out to be. In cases that seem difficult to decide it is advisable to try codifying an attribute as one or the other type and, if this proves to be incorrect, to change it as necessary.

It is important to note that per-file flags and policy are complementary, and not mutually exclusive, approaches. Our design implements both.

3.6 Metrics

The metrics chosen to test a file should be both comprehensive and efficient. Given that we shall be doing realtime checking, any significant slowdown will be immediately apparent, particularly in the case of frequently-accessed files; however, it would be better to do the job well and ensure that an invalid file does not pass than to pass a file that is invalid because the tests done were insufficient. This section details metrics that are desirable in a realtime file integrity checker.

3.6.1 Hash

All systems discussed in chapter 2 use a cryptographic hash, digest or fingerprint – the terms differ, but the meaning remains the same – to ensure the validity of the contents of a file. Given the weaknesses of the MD5 [56] algorithm [71] which was previously thought to be secure, and the weaknesses that may be found in the currently-popular SHA1 [19] algorithm [70], it seems prudent to ensure that a system is able to resort to stronger cryptographic hash functions should weaknesses be found in what is currently best-of-breed. This may be done by separating the hashing function from the rest of the code and making it general enough to be replaced without too much effort; this approach has the additional benefit of making it less important to discuss the security of various hashes since one may be replaced by another with ease in the event of a problem arising.

The question of how many hashes should be computed is an interesting one as it weighs up the security afforded by having two (or more) methods of testing the file contents against the slowdown imposed by needing to calculate each additional hash. The benefit of having n hashes is that the byte-stream properties captured by each one are (presumably) not the same, and an attacker must find collisions in n hashes for the same byte-stream – a difficult proposition.

Mathematically, we can depict a ***one-way function*** that transforms an input I into a hash value v using function f as $f(I) = v$. Let us say that a given input I_{original} returns hash values v_{f_0} and v_{f_1} using hash functions f_0 and f_1 respectively. A *partial* collision is one in which, for an input I_{new}, either $f_0(I_{\text{new}}) = v_{f_0}$ or $f_1(I_{\text{new}}) = v_{f_1}$ is true; a full collision, which is what an attacker would need in order to be successful in deceiving the system, is one in which both hold true. We define the *searchspace* of a hash function f as the average number of inputs that must be tried before a collision is found, and we can depict the searchspace of f as f^s.

Using these definitions, we can see that it will (on average) take f_0^s different trial-and-error inputs to obtain I_{new0} that satisfies $f_0(I_{\text{new0}}) = f_0(I_{\text{original}})$, and that it will take f_1^s different trial-and-error inputs to obtain an I_{new1} that satisfies $f_1(I_{\text{new1}}) = f_1(I_{\text{original}})$. The security obtained from using two hashes can be seen as the probability that $I_{\text{new0}} = I_{\text{original}} = I_{\text{new1}}$, since this is the condition that must be met for a full collision to occur.

Statistically, we can say that the probability P_0 and P_1 of finding I_{new0} and I_{new1} using a brute-force method are

$$P_0 = \frac{1}{f_0^s}, P_1 = \frac{1}{f_1^s}.$$

This assumes that the P_0 and P_1 are independent variables; in other words, the assumption made is that finding I_{new0} tells us nothing that will aid us in finding I_{new1}. The probability $P_{\text{collision}}$ that $I_{\text{new0}} = I_{\text{new1}}$ is then

$$P_{\text{collision}} = P_0 \times P_1 = \frac{1}{f_0^s \times f_1^s}.$$

This can be generalized to show that the chance of finding a collision in n hashes would be

$$\frac{1}{f_0^s} \times \cdots \times \frac{1}{f_n^s} = \prod_{i=0}^{n} \frac{1}{f_i^s}$$

Therefore, using multiple hashes provides much more assurance that the contents of the file have not been compromised. However, we must ask ourselves whether much more assurance is needed.

Looking at the argument for using a single hash, we find that a popular hashing algorithm such as SHA1, assuming that [70] holds true, currently offers a searchspace of around 2^{69}, and that even if a collision is found it is not guaranteed to be in a form (such as a valid binary executable) that is useful to an attacker. This means that if we were able to test hashes at a rate of around 10^{12} per second, it would still take approximately 590,295,800 seconds (or around 18 years) to find a collision. Furthermore, once additional metadata checks are done it may not be possible to find an input in the form of a valid executable at all: for example, if we test both the file size and the file hash, then the set of possible inputs is constrained to consist exclusively of inputs that are a certain size. Within the new, smaller set of possible inputs, it is no longer certain that a collision can be found at all, and it becomes even more unlikely that any input found that causes a collision will be in a form that is useful to an attacker.

In essence, we need to consider whether the continual computational cost of using two or more hashes is greater than the one-time cost of replacing a single hash with a more secure one – if the security of the currently-used hash becomes an issue. Both options result in greater security and given the complexity added to a design by having to deal with between 1 and n hashes, as opposed to knowing that only one hash needs to be dealt with, there is reason to go with the single-hash option. As more research is done into the advisability of using multiple hashes, this question will become easier to answer; in the interim, lacking any convincing argument for either route, our design takes the faster (and simpler) option: a single hash.

3.6.2 Metadata

The file metadata that should be stored should be both system-independent and orthogonal. By "orthogonal" we mean that the metrics stored should not refer to the same property of a file more than once: it makes little sense to have two tests that devolve to returning the same semantic content – for example, a test to check the file owner ID and a test to check the file owner name. By "system-independent" we mean that those properties of a file that are unique but system-dependent should not be tested. An example of a system-dependent property on Linux is the ***inode***-number of a file, which depends on a number of system-dependent factors such as when the file was created and how much data was present on-disk at that time. Ideally, a third party should be able to supply metrics that can be used to validate a file on any system; if system-dependent metrics are used, this becomes impossible.

3.7 Storing Metrics

The question of where to store the metrics we have decided upon can be reduced to two choices: along with the file, or in a separate database. From Table 2.1 we can see that of the kernelspace file integrity checkers discussed, CryptoMark, DigSig, SEFL and WLF use same-file storage; I^{3}FS, TrojanProof and Veriexec use database storage. In other words, the ratio of kernelspace file integrity checkers that use same-file storage to those that use database storage is 4:3. This gives an indication that the question of which to use by preference is still an open one.

3.7.1 Same-file Storage Advantages

Sometimes termed "signed files", files with embedded metrics can hold more than just a digital signature. Some advantages of same-file storage are:

Efficiency Each time that a test is done, only one file needs to be opened; as the file is read, the metrics can be read in as well.

Simplicity There is no need to open a database or initialize any structures before the first file can be tested. Maintaining a list of files to be tested against is also unimportant, and therefore there needs to be no algorithm devised for searching quickly through a file-list or associating a path string from a file-list with appropriate metrics, among other routines.

Ease of distribution Data is carried with the file, making the distribution of signed files extremely easy.

3.7.2 Same-file Storage Disadvantages

Some disadvantages of same-file storage are:

File-bound Same-file storage means modifying the data that one is to test, and not all file-types are amenable to this sort of modification. ELF files [66, 42], for example, have SHT_NOTE sections that are perfect for including information that will not appear in the final executable, and many scripting languages provide facilities to comment out lines that can be used to store metrics; however plain-text documents (for example) have no equivalent. Using same-file storage, it may not be possible to sign all files without altering their semantic content, and even if such a thing were possible each file-type must be recognized to enable the metrics to be read from a particular file. In a kernelspace model, this may necessitate creating file-type recognition routines that could be misled; in any event, the complexity of the system is drastically increased by such an approach.

Backwards-incompatible Alternately, if same-file storage is to not be file-bound, with all the complexity that that option entails, then it must be backwards-incompatible; this is the choice made by WLF (see 2.15). A decision could be made to either append or prefix each file with n bytes containing metric data, which means that the checking system needs only to check these bytes in each file to get the information it needs – regardless of file-type. This approach would, for certain file-types, mean that they become unusable on other systems: the new "secure" format is not backwards-compatible with the more general format. In a certain light, this could be seen as a feature as it would now be impossible to access any invalid file on a system; however, it would certainly be an inconvenient feature to have.

If a file is made to be backwards-incompatible, another issue that arises is the necessary modification of userspace utilities to detect and compensate for this change. For example, interpreters would have to be modified to ignore the last n bytes of a file, as would graphics programs and document viewers. This could easily lead to a rippling series of changes that spread throughout the system, necessitating changes in almost everything!

Speed File metrics cannot simply be left exposed within a file, as they would then be easy to tamper with. An attacker would need only to alter a file to his specifications, calculate the correct metrics, and add them to the file to have it recognized as valid by the system. Instead, precautions have to be taken to ensure that tampering is either not possible or at least detectable. The way in which to do this, as used by CryptoMark, DigSig and WLF (see 2.5, 2.6, and 2.15 respectively), is to encrypt the metric data within the file using asymmetric encryption. To read the data, the integrity checker would decrypt it first using a public key that is either supplied upon checker start-up or compiled into the file integrity checker executable or module. Asymmetric encryption is a slow, computationally-intensive process that is orders of magnitude slower than symmetric encryption [20, p. 28], and requiring its use every time that a file needs to be tested decreases the speed of an integrity test by an appreciable amount. Apvrille *et al* in [3, p. 7], for example, state that "Results clearly indicate that the modular exponentiation routines [used for RSA decryption] are the most expensive, so this is where we should concentrate our optimization efforts for future releases".

Symmetric encryption, using keys that are guarded by an appropriate asymmetric encryption scheme, is an option that has not been used by any project known to us. The reason for this is that such an approach creates an issue of key storage and key distribution: a system *must* have access to the symmetric key in order to decrypt the data within files, and storing or transmitting this key is subject to a potential compromise. Even guarded by asymmetric encryption, a system must still have access to a public key that will allow it to read the symmetric key – and if this public key is compromised, obtaining the symmetric key is easy. Furthermore, an attacker who gains access to the symmetric key is able to create his own "valid" files; by contrast, an attacker who compromises a system such as the asymmetric one described above gains nothing but the public key, which is of no use to him if he would like to create his own set of "valid" files.

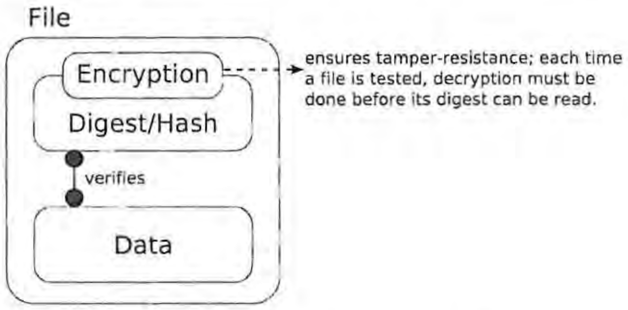

File
Encryption
Digest/Hash
verifies
Data
ensures tamper-resistance; each time a file is tested, decryption must be done before its digest can be read.

Figure 3.2: Tamper-resistant same-file storage

Figure 3.2 depicts tamper-resistant same-file storage. Each time that the file is tested, asymmetric decryption must be used in order to gain access to the hash for comparison purposes.

Revocation A signed file, X_n, can be replaced by a previous version of that file, X_{n-1}, without the replacement being detected. This attack allows an attacker to replace X_n, considered secure, with a version that contains a security flaw; it also allows an attacker to replace a configuration file or document with any previous version. This problem is inherent to the idea of same-file storage as there is no way to tell which version is the "correct" one without referring to an external source – and the point of same-file storage is to not have to refer to an external source.

One way to avoid the problem entirely is to make each signed file impossible to modify. However, this approach does not work well with the concept of network-mounted filesystems, and creates upgrade difficulties: files that are difficult to replace are difficult to upgrade. To solve this last problem it is tempting to say that only users with super-user privileges should be able to replace files, but if this is the case then anyone gaining super-user privileges can defeat the system with ease. Clearly, making files impossible to modify is not a good solution.

This problem has no real workaround that does not remove the advantages of **efficiency** and **simplicity**. One workaround to the problem is to keep a database of versioning information, updated each time that a new file is added or replaced. Another way is to keep a "revocation list" of file signatures that have been declared invalid, and test each file against this list before declaring it valid or not; this is the approach taken by DigSig and WLF (see 2.6 and 2.15 respectively). The revocation list is appended to every time that a file is replaced. Importantly, the revocation list *never* decreases in length, and can be viewed as a database of invalid files: once again, this returns us to the idea of using a separate database as well as same-file storage.

Note that **file-bound** and **backward-incompatible** are mutually exclusive disadvantages: choosing one will render the other inapplicable.

3.7.3 Database Storage Advantages

"Database storage" need not refer to an actual database management system, relational or otherwise; used in this paper, it simply means storage that is external to the file being tested. The benefits of storing metrics in a separate database are

Security A database file may be protected by special measures, only be made available across the network in a read-only fashion, written to CD or write-protected flash-media, and so forth. Additionally, the database file may be signed to make it easy to detect tampering. It is easier to protect a single file with special measures than to protect an entire filesystem, and keeping a separate database of metrics is therefore seen as more secure than keeping metrics stored in every file.

Speed The database contains references to all valid files on a system and is read during integrity checker initialization, before any integrity checking begins. This presents the opportunity to store file metrics at least partially in memory before any file is read, reducing the need to refer to the database file upon every access to any other file. We can view this as an initial start-up cost that is amortized by the long runtime of the realtime integrity checker.

Having a separate database also presents the opportunity to *indirectly* digitally sign every file on the system. In a same-file storage system, each file should be digitally signed – that is, a cryptographic digest of the contents should be encrypted with a private key – in order to detect tampering. In a database storage system, the database can be digitally signed instead, thus protecting the metrics contained within it from being tampered with; indirectly, then, we can see that signing the database is equivalent to signing each file that should be tested. The speed increase that results from not having to use asymmetric encryption during most of the runtime of the integrity checker should be appreciable.

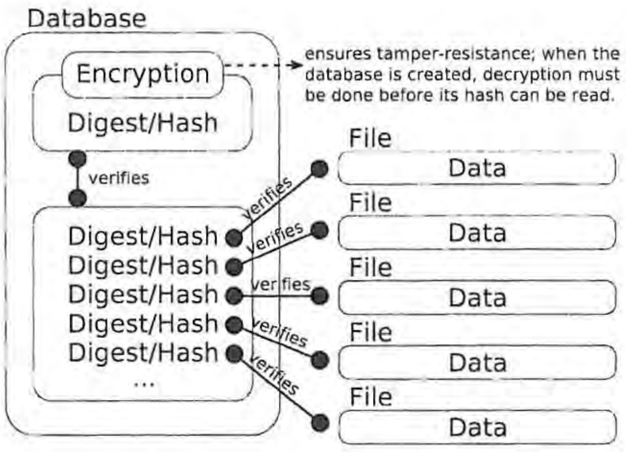


Figure 3.3: Tamper-resistant database storage

Figure 3.3 depicts tamper-resistant database storage. Upon reading the database for the first time, asymmetric decryption is used to gain access to a hash of the database contents. Using this hash, the integrity of the database is verified. Verifying the integrity of the database has the effect of verifying the integrity of all metrics *within* the database, and these metrics may now be used under the assumption that they are correct. In this fashion we are provided with the same guarantees of integrity for file metrics that same-file encrypted digests provide us with, but at a fraction of the computational cost.

For security reasons, asymmetric public keys should be compiled into the module. This is because an attacker should not be able to circumvent the tamper-resistance of the system without being able to replace the entire system. The drawback of this approach is that if the private key is compromised, or the entity responsible for signing the database is replaced, the IMon system must be recompiled with a new key-pair.

Universal application A database can contain the name of any file, irrespective of file format. This allows images, documents, binaries, scripts, and any other such files to be tested without adding complexity to the integrity checking system or breaking backwards-compatibility of file formats.

Low-impact The impact of storing file metrics in a database is small on a production system. No files need to be specially modified, and no userspace utilities need to be modified to understand changes made to files.

Version control As discussed in 3.7.2, revocation becomes a problem when using same-file storage. Given an upgrade process that only accepts an upgrade as valid if the database version is greater than the current database version, revocation is not an issue for database storage. Therefore, we can state that version control is a benefit of database storage.

3.7.4 Database Storage Disadvantages

Some disadvantages of storing metrics in a separate database are

Upgrade-unfriendly If a separate database is used, it may become inconvenient to upgrade a system. The reason for this is that an upgrade to a system would require the file metrics to be calculated and the database to be modified and possibly digitally signed once again, then re-read by the integrity checker. This constitutes an additional step in the upgrade or installation process.

It should be noted that this may be seen as an advantage instead of as a disadvantage since without explicit authorization that files are valid, a file should *not* be allowed to execute – and therefore, without a specific additional security step involved in all upgrades, the system becomes insecure. Nevertheless, it is mentioned here as a disadvantage since it increases the inconvenience of system maintenance.

Inflexibility Section 3.4.1 deals with the ability to use regular expressions to deal with, at runtime, files that match a pattern but may not have existed when the regular expression was compiled. If database storage is used, this flexibility falls away: the integrity checker needs to be initialized from the database before any files can be tested, and since it is not possible to know file metric information for files that do not yet exist, it is not possible to initialize the integrity checker with that information; hence, the flexibility promised by regular expressions falls away.

3.7.5 Alternatives to Cryptography

There are ways that could be used to protect the database from tampering that do not involve asymmetric cryptography. Keeping certain system components on read-only media is the idea favoured by AIDE [33], SEFL [68], DigSig [2] and AFICK [22] whether the component is a revocation list, a database, or a certificate.

The three main methods that could be used for tamper-protection via read-only media, and the reasons that we did not choose any of them, are

Physical media The component to be guarded could be placed on a flash stick that is made read-only, or written to CDROM. This is probably the best way in which to guard the component, but it is inconvenient. Having to burn a CDROM or write flash sticks each time that an upgrade is done is time-consuming and physically inconvenient, and would hamper the acceptance of any system, particularly in the cases of remote or hosted systems.

Read-only network-mount This approach overcomes the objections made to physical media, but introduces a new one: the network becomes a single point of failure. If it goes down, or if it is unavailable, then the system fails to operate. An example of this is a mobile worker who is sometimes out of range of any network, or who may be connecting to his home network through the network of some entity that impedes his access to the component.

A way to get around these difficulties is to keep a copy of the database for offline use; however, when no way of detecting tampering is used, it is not possible to do this and be sure that tampering does not occur. If cryptography is used, then read-only network media is an attractive option; but if it is not, then the problems of connectivity cause it to be far less attractive to us.

System-specific attributes TrojanProof (see 2.13) tests to see whether the component to be guarded is either on read-only media or made ***immutable***. Likewise, we could take advantage of certain system-specific attributes to make the guarded component unalterable. However, this would make the design of the integrity checker dependent on whether a given system supported something like the "immutable" attribute.

Given that the design proposed is meant to reside in kernelspace, it might be possible to ensure that the guarded component is made effectively immutable, whether the system supports the concept natively or not. The drawback of implementing such a thing in preference to using asymmetric cryptography is that using the latter allows the component to be given to us by a third-party – and allows for this to be verified through the digital signature – whereas the former method does not.

3.7.6 Decision

It is important to note that, unlike the discussion of architecture (see 3.1), the decision of which storage to use for file metrics is *not* exclusive. In other words, it is possible to conceive of a scheme which uses a database to store file metrics and, if a particular file is not found within the database, searches the file for encrypted metric information and uses that, if found. This approach would have all the advantages of database storage, and most of the advantages of same-file storage, with none of the disadvantages of either (assuming that revocation can be handled via the database). The only disadvantage to be seen is that of the added complexity inherent in creating one as a fall-back method for the other, and maintaining both.

Despite the advantages of implementing both approaches, we are restricted by time constraints to only implementing one. The obvious one to implement is database storage as this has fewer disadvantages to it; furthermore, the advantage of being universally applicable to both machine-code binaries and interpreted scripts is extremely convenient. However, we should bear in mind that both approaches are possible and, if we can, ensure that the code we write is modular enough to be modified in future to handle same-file storage as well.

In our design, IMon-related files such as the database are placed in a separate directory. This is so that any configuration files or other future feature enhancements that require permanent storage can be placed in an already-defined location.

3.8 Self-Protection

If a security system is itself compromised, the security provided is worse than not having any security at all: in the former case, an administrator believes that the system is protected and may not make checks that he might have made in the latter case. Therefore, limited self-protection is important for a secure system.

We say that the self-protection is "limited" in the sense that we trust the system to be in a secure state prior to the security system being made active. This can be assured through the use of a secure-boot process such as Aegis, which is described by Arbaugh, Farber and Smith in [74]. In fact, we need only trust that the operating system kernel (and, if provided separately, the file integrity checker kernel module) is in a known-good state since if this holds true, we are able to test the validity of all other components using the file integrity checker.

Upon starting up, the integrity of the database should be tested using a digital signature. If this fails, a warning should be printed and the integrity checker should fail to load – and an extremely security-conscious administrator may choose to disable access to the system at this point, pending further investigation. However, assuming that the database is in a coherent and reliable state, the integrity checker should load successfully and enact the following security measures:

- Creation, deletion, and renaming of files in the configuration directory should be denied.
- Locking the database should be denied.
- Loading unknown kernel modules should be denied.
- Unloading the file integrity checker module should be denied.
- Writing to kernel memory should be denied.
- Truncating or opening the database in a mode that allows writing to it should be denied.

These measures are easy to enforce from the operating system kernel, and help to ensure a basic level of self-protection that defeats trivial attempts to subvert the system.

3.9 Upgrading

Upgrading IMon files such as the database can be done from userspace via a utility program that ensures that the replacement file is indeed a valid upgrade for the current file. One way of ensuring this is for the userspace program to set up an authenticated, encrypted connection to a server which provides updates, and to only obtain updates through this secure channel; since the server will always provide the latest version of a file, the problem of distinguishing an older file from a newer one falls away.

The upgrade program itself would have a per-file flag set that indicates its purpose, and would therefore be accorded special privileges – such as being able to write to the directory in which IMon files are stored – that are not accorded to other processes. If we can ensure that the upgrade program works securely and correctly, and we can ensure that it has not been tampered with, then the upgrade process becomes almost as secure as if we were performing it from the kernel itself.

To ensure that the IMon system is left in a consistent state, the following steps (as modified from our work in [49]) can be followed for a sample file named `filename`:

1. The upgrade utility is started. The replacement file is locked such that no other entity can replace it with a malicious version. The integrity of the replacement file is tested using the digital signature that it contains.

2. IMon ceases integrity-checking whilst the upgrade is in progress. This stops race conditions due to IMon attempting to find file metrics in the database whilst the database is being replaced. A side-effect of this is that no execution can take place whilst the upgrade process is ongoing.

3. `filename` is renamed to `filename.old`.

4. The replacement file is copied to the IMon directory under the name `filename`. At this point, both `filename` and `filename.old` exist.

5. Synchronization is done so that any kernel buffers are flushed to disk.

6. `filename.old` is removed.

7. The upgrade utility exits. IMon, detecting this, verifies and re-reads `filename` before continuing with integrity-checking once more.

A side-effect of this upgrade process is that, on system startup, a test for `filename.old` must be done if a test for `filename` fails. By using the above procedure we ensure that the system remains in a consistent state even if any one of these steps is interrupted.

3.10 Algorithms

This section concerns itself with the selection of algorithms that are used within the file integrity checking system, based on the preceding design and specification that has been laid out throughout this chapter.

3.10.1 Minimizing Disk Access

One of the advantages of same-file storage (see 3.7.1) is that it requires only one file to be open whilst a test is done, thus saving the time spent reading data from an on-disk database. We can minimize the number of times that we have to refer to the on-disk database through a combination of two methods: on-demand loading and reference counting.

When the file integrity checker is first initialized from the database, we do not need to read in all the metrics associated with each record. Instead, we need only read in the path of the file, some bits representing the flags associated with it, and a reference to where in the database more file metrics are to be found: we call this the "stub" record, and differentiate between it and the "full" record contained in the database. Some files may never be accessed during the runtime of the system, and retaining their full metrics in memory would be a waste of resources. This partial loading of file records into memory, and the strategy of only loading the full record when we must, is on-demand loading. On-demand loading ensures that only those files which are actually being used will have their full records loaded.

When a file is accessed, a request can be made to get the full record associated with the file. If the full record is already present in memory – this happens if the same file has been opened by another process – then a reference counter for that record can be incremented, and the record can be returned. Otherwise, the record can be read from disk. When a file is closed by a process, we decrease its reference count; and when the reference count drops to zero, we free the full record (thus returning it to being merely a stub) and must reload it from file if it is required once again. This illustrates our use of reference counting.

Figure 4.4 on page 93 shows this process diagramatically. Combined, on-demand loading and reference counting help to make the best possible use of memory that is available to us whilst maintaining acceptable performance.

3.10.2 Lookup

One of the most common tasks that will be performed is searching for a matching record. Therefore, this task must be carried out with the greatest efficiency. Other implementations have in the past found this to be a bottle-neck as far as performance is concerned [46, 43]; however, we can take advantage of the fact that we create the database to ensure that it is sorted lexicographically by file path before we load partial records into memory in a way that supports random-access of elements. The reason for this sorting is to make file lookups quick and efficient, using a binary search (as described by Knuth in [31, pp. 409–414]) instead of a linear search. A binary search is an $O(\log_2 n)$ operation; the performance gain is obvious when contrasted to the $O(n)$ it would take to perform a linear search – which we would have to do had we been forced to use a sequential-access container or not been able to pre-sort records.

Another data structure and associated algorithm that could have been used is that of a hash table: it is more complex and slightly slower, but may be better-suited to modification to allow both same-file and database storage to coexist, as described in 3.7.6.

3.10.3 Caching

Caching the results of a previous integrity check, as shown by SEFL and DigSig (see 2.11 and 2.6 respectively), can dramatically increase performance. However, if done incorrectly, caching can result in a file that should be checked *not* being checked – and thereby introduce a security flaw into the system. To implement caching securely we introduce a per-file flag, which we shall refer to as the cache-flag. This flag is initially unset.

We set the cache-flag for a file after a verification has been done, if all the following conditions are met:

1. The filesystem that the file resides on is not a network filesystem.
2. The file is not memory-mapped writably.

We clear the cache-flag for a file if any of the following conditions occurs:

1. A request is made which may alter file metrics, such as writing to a file or creating a ***hard link*** to it.

2. The file is memory-mapped writably.

3. A request is made to write directly to a ***block device*** which may be a hard disk.

In the case of (3), the cache-flags of *all* files are unset as a precaution.

3.11 Evaluation

In the previous chapter other file integrity checking programs were examined, and flaws in them were identified. Now that the design of IMon has taken shape it is instructive to compare this project with those, using the same criteria.

IMon is an *I*ntegrity *Mon*itor, used to monitor the file integrity. It is written in C, and takes the form of a kernel module and utility programs that generate a database for its use. Metrics tested for are comprehensive, including a cryptographic hash of file contents, timestamps, permissions, and owner and group of a file. File metrics are stored in a database whose integrity is ensured by a digital signature. IMon protects itself at runtime by disallowing any form of write access to the database, restricting those executables which are allowed to load and/or unload modules, and denying attempts to write to kernel memory.

Configuration of IMon takes the form of setting policies and per-file flags for specific files. Any file may be tested by IMon. Tests are carried out prior to execution, before read access is granted for listed files, and before read access is granted for any file to be read by an interpreter. A caching system which stores the results of previous integrity checks is used to increase performance, and on-demand loading of records is implemented to minimize disk access. IMon is the only realtime file integrity checker on Linux that is able to handle both interpreted files and native binary executables in the same fashion. The performance impact of IMon on a production system is negligible.

3.12 Summary

In this chapter we have discussed various aspects of file integrity checker design, based on the examination of file integrity checkers done in chapter 2. We have looked at possible architectures for a file integrity checker (see 3.1), touched on issues such as interpreted execution (see 3.3) that have not been discussed by many other papers, debated open issues such as metrics and metric storage (see 3.6 and 3.7 respectively), proposed the use of implementation-independent algorithms to increase file integrity checker performance (see 3.10), demonstrated the importance of dependencies and file attributes (see 3.2 and 3.5 respectively), and laid out guidelines for configuration, self-protection, and upgrading.

In chapter 4 we shall describe our implementation of the design that has been put forward in this chapter.

Chapter 4

Implementation

In the previous chapter we discussed the design of an ideal file integrity checker. In this chapter, we shall implement that design on the Linux operating system, referring back to chapter 3 where necessary.

The full source code of IMon and associated utilities is available on a compact disc that accompanies this thesis.

The implementation of IMon is split up into several distinct areas, each of which may be discussed and understood separately. These areas, and what is discussed under them, are:

System An overview of IMon, and how all of the areas fit together to create an integrity-checking system.

Baseline Configuration file syntax, creating a baseline database to be used by the integrity-checking system, selecting per-file policy, generating keys and signing the database.

Cryptography A basic overview of RSA [57] encryption and decryption, and how it is used by IMon.

Database Access to the database, on-demand record loading, reference counting, and searching for records.

Integrity Checking Situating integrity checks, implementing policy, and acting upon failure.

4.1 System

IMon was built on a Linux system running kernel version 2.6.12, and tested for kernel versions 2.6.12 and 2.6.14 (which, as of this writing, is the latest stable version of the Linux kernel). At present, the IMon kernel module consists of ≈4250 lines of highly-commented C code. This section gives a high-level view of what the kernel module does, and references more detailed sections as necessary.

4.1.1 Hooks

Actions performed in userspace cause other actions to be performed in kernelspace, with the results of these actions usually being returned to userspace. For example, a userspace program could request that n bytes be read from a file. This would cause the kernel to read a certain number of bytes from the specified file and return n of them to the userspace program.

The kernel also provides *hooks*, which one may "hang" code from. Once a hook is reached, the code that is associated with it will be run. Hooks are implemented using function pointers which (by default) reference dummy functions that usually do nothing and return successfully. We shall be using kernel-provided hooks that are called during program execution and when a request is made to read or write to a file. The hook framework that we shall be using is the Linux Security Module (LSM) Framework [13, 12], the essentials of which are covered in 4.5.1.

4.1.2 Integrity Checking

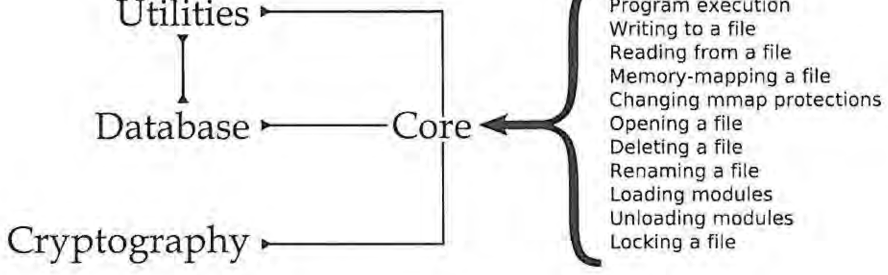


Figure 4.1: IMon Runtime Overview

IMon may be seen as four components (Core, Database, Utilities, and Cryptography) that interact at runtime to do integrity-checking. Figure 4.1 shows this diagrammatically, with the tasks affected by IMon being shown to the right as input into the Core.

The Cryptography component (see 4.3) is the simplest, and provides access to cryptographic hash functions. In reality this service is provided through separate kernel modules, with Cryptography being a set of functions that provides a convenient interface to them. This component is used by IMon during the testing of file metrics. Note that Cryptography does *not* use the big-number implementation (see Appendix C) that we created since asymmetric cryptography is only utilized during the initialization phase of IMon.

Utilities (see 4.5.7) provide functions to:

- convert a file into an absolute filename
- determine whether a file is testable or not (and, optionally, what the type of the file is)
- get a single line from a file; used during parsing of the database file
- associate a record with a process, which requires interacting with the Database component

The functions in Utilities have been created for convenience, and are used by Database as well as Core.

The Database component (see 4.4) is used by Core and uses Utilities to obtain a lines from the disk-database so that on-demand loading can occur. Core uses Database to store, request, and release records.

The Core component (see 4.5) consists of the LSM hooks described in 4.5.1, and thus forms the interface through which the rest of the kernel interacts with IMon. It utilizes the other three components to carry out its functions and receives input by way of parameters passed from any of the kernel functions that run in response to a task listed to the right of Figure 4.1. Since some of the tasks (such as "reading from a file") are very common, IMon may have its functionality invoked very often.

Since Cryptography is simply an interface to other kernel functions, and Utilities are helper functions that are not central to the purpose of IMon, we do not spend much time describing them in this chapter. Instead, we describe the implementation of Database and Core in detail since those are the two components that are central to the functioning of IMon.

4.1.3 Lifecycle

The lifecycle of IMon can be seen as comprising three phases: initialization, runtime, and shutdown. The first occurs when IMon is loaded, and the third occurs when IMon is unloaded – with the majority of time being spent in the second phase, which occurs in the interval between loading being completed and unloading beginning.

Initialization

Upon startup, the function imon_init is called. In this function we first load the public key components compiled into IMon, which means translating public-key hexadecimal strings into numbers; this is done by way of the big-number implementation created for the kernel (see Appendix C). We then proceed to verify IMon files (see 4.3.2), and (assuming that the verification is successful) release memory associated with the keys. The create_database function is called, which calls initDB (see 4.4.3) in turn to create the database used by IMon. Lastly, we register the LSM hooks that we use with the LSM framework (see 4.5.1).

If any of the above steps fail for any reason, IMon does not continue with subsequent steps. Instead, it refuses to load and prints an error message indicating why initialization failed.

Runtime

During runtime, hooks registered with the LSM framework (see 4.5.1 and 4.5.6) are called, and integrity checking takes place. Every read operation, write operation, and execution operation is now monitored by IMon. An overview of this process is given in 4.1.2, with more detail being available in 4.5.

Shutdown

When a request is made for IMon to be unloaded, the imon_fini function is called. This function first deregisters IMon with the LSM framework (see 4.5.1) and subsequently calls a function to release resources used by the database.

4.2 Baseline

This section describes how a baseline is created for the system. We require a baseline to be able to differentiate between "good" and "bad"/"unknown" files using file metrics. Creating a baseline for the system is done in 3 stages.

1. We first create a configuration file that determines which files we will be testing. After IMon is running, only these files will be allowed to execute. The configuration file also specifies specific policies for certain files if the default runtime policy assignment (see 3.5.2) is not appropriate for a particular file.
2. The configuration file is then fed into a program which generates baseline data for those files and produces a flat textfile ordered by filename: this is our database. Following each filename are the dependencies and policy for that file, and data about the file such as a hash of its contents, timestamp records and permssions.
3. Assuming that the administrator is satisfied with the resulting database, he then signs it using his private key, and loads it onto the workstation. At this point it becomes extremely difficult for an attacker to modify the database; to do so, he would require access to the administrator's private key, which he is unlikely to get.

This process is illustrated in Figure 4.2 and explained further in 4.2.1 through 4.2.3.

4.2.1 Configuration File Syntax

To make administration as easy as possible, the configuration file consists purely of filenames, comments, and policies. A filename that begins on a new line and has no space prepended to it is considered to be a *main* file, and any filenames that are indented are considered to be dependencies. Typically, we have an executable as the main file and any configuration files or libraries that it depends on as dependencies. Policy or per-file flags may be set for a main file by specifying the policy name

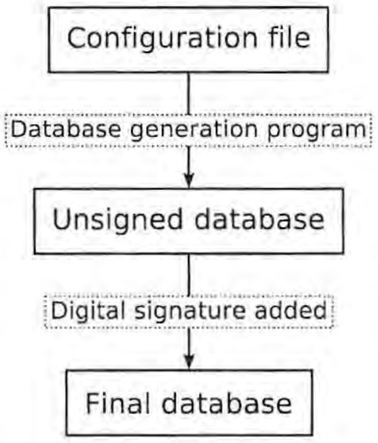


Figure 4.2: Database generation from a configuration file

(**deny_others**, **accept_after**, **accept_anytime**, or **deny_notfound**; see 3.5.2 for details) or the per-file flag (**ignore_ctime**, **module_capable**, **ignore_hash**, etc; see 3.5.1 and below for details) as a dependency, i.e. indented under the name of the main file. If the first non-blank character on a line is a hash ("#"), the line is treated as a comment.

Since a file must fall under one and only one policy, if more than one policy is specified, the last one specified is the one that is used. If no policy is specified, a default policy of **accept_anytime** or **deny_notfound** is used, depending on whether the file is interpreted or not (see 3.5.2). The same does not hold true for per-file flags, any number of which may be specified in any order. Of course, per-file flags may be repeated with no ill effects since this would simply turn the same flag on twice or more, resulting in the flag still being turned on.

An example excerpt from a configuration file is provided as Code Snippet 4.1.

Policy and Per-File Flags

Each per-file flag is mapped from configuration file syntax to a number that represents a #DEFINE statement. The flags, what they do, and which #DEFINE'd symbolic name they are synonymous with are as follows:

ignore_hash Mapped to IGN_HASH. This specifies that the file hash should not be tested; however, in line with 3.3, if the file is an interpreted one that is to be executed it is tested even if **ignore_hash** is set.

```
/usr/bin/code2color
    accept_anytime
/usr/bin/lmhttp
/home/cynic/module/tests/test5/test5_NOW_accept_after
    /lib/libc.so.6
    /lib/ld-linux.so.2
    /lib/libdl.so.2
    accept_after
    /home/cynic/module/tests/test5/libtest5_dependency.so
/usr/bin/python2.4
    interpreter
/usr/bin/smbmount
/etc/cups/certs/0
    ignore_mtime
    ignore_ctime
/etc/mtab
# stuff may get added all the time to mtab, don't worry about it.
    ignore_ctime
    ignore_mtime
    ignore_hash
    ignore_size
/etc/idmapd.conf
/etc/ld.so.conf
    ignore_ctime
    ignore_mtime
/etc/modules.devfs
```

Code Snippet 4.1: Example: Configuration file excerpt

ignore_ctime Mapped to IGN_CTIME. This specifies that the status-change timestamp of a file need not be tested.

ignore_mtime Mapped to IGN_MTIME. This specifies that the modification timestamp of a file need not be tested.

ignore_uid Mapped to IGN_UID. This specifies that the owner of a file need not be tested.

ignore_gid Mapped to IGN_GID. This specifies that the group of a file need not be tested.

ignore_links Mapped to IGN_LINKS. This specifies that the number of links to a file need not be tested.

ignore_mode Mapped to IGN_MODE. This specifies that the read-write-execute permissions of a file need not be tested.

ignore_size Mapped to IGN_SIZE. This specifies that the size of a file need not be tested.

interpreter Mapped to FLG_INTERP. This specifies that the file is an interpreter, and should be treated as such (see 3.3).

module_capable Mapped to FLG_MODCAP. This specifies that the file is a loader or unloader of kernel modules.

Three flags are set and unset at runtime:

FLG_CACHED This flag is set when a file may be cached at the end of an integrity check, and unset when the file is written to.

FLG_TESTING This flag is set when a file is being tested, and unset as soon as testing is complete.

FLG_MMAPW This flag is set when a file is memory-mapped with permissions that allow writing to it, and unset when no more references to the file are extant. It acts to inhibit the setting of FLG_CACHED for a file. For an explanation of why this flag is necessary, see 4.5.4.

These runtime flags need no representation in configuration-file syntax since they are only toggled at the behest of IMon.

4.2.2 Database Generation

The configuration file is run through a database generation program which determines file metrics given filenames, resolves symbolic links, removes duplicates, sorts the database by filename, adds data to make records easy for a computer to read, turns file references into database indices, and more. Essentially, the database generation program does most of the menial tasks that would take a great deal of time to do manually (and, most probably, imperfectly!). Eventually, the database generation program outputs a database that requires only that the administrator sign it with his private key. Figure 4.2 illustrates this process.

Code Snippet 4.2 expresses the database format in Extended Backus-Naur Form (EBNF). The "DepIdx" token in this Snippet requires some explanation. It represents the dependencies of a given record, with each dependency being represented by a particular index number into the database array. If we were to instead list the dependent filenames in each record, populating the dependencies array of a single record would require us to search through the entire database to find the correct record, then place it in the correct slot in the dependencies array – and this would be repeated for each dependency that a record has. Listing the index numbers of dependencies, which we can calculate at database compilation time, is a far easier and simpler solution that gives us a performance boost during the lookup procedure at runtime.

```
Database := Signature NumRecords MaxRecordLen Record*
Signature := HexNumber
NumRecords := Number
MaxRecordLen := Number
Record := StrLen Filename Flags Policy DepSize DepIdx* Stats
Stats := Hash UID GID Mode Links Size CTime MTime
StrLen := Number
Filename¹ := Character+
Policy := '1' | '2' | '3' | '4' | '5' | '6'
Flags := Number
DepSize := Number
DepIdx := Number
Hash := HexNumber
UID := Number
GID := Number
Mode := Number
Links := Number
Size := Number
CTime := Number
MTime := Number
Number := Digit+
HexNumber := HexDigit+
HexDigit := 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | Digit
Digit := '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'
```

[1] "Filename" is preceded by the "StrLen" token which gives its length. It consists of any characters at all; to denote this, the (undefined) token "Character" is used.

Code Snippet 4.2: Database EBNF

The first ten lines of an actual database configuration file are provided as Code Snippet 4.3.

```
36f97102ccd87d1a35fa3e1ff1781a3458d4bb598d67a18178bee60a6f6f92d518b09e344eed3c90c3f0c6783b50aba15ab4 ↵
09c4188ba6f93119c5a9fb347f18015bd61c8f020a3e699d90956bf3bab2c68c8a560556ed09f8b5b3bcd8f1d7ffe7f5f50a ↵
273e6dd4a7d851d997fd5594a578edd2a11abd2af69e81c5ff122df49ad04513b1d459457cf827b3a0f1944902d23005cc3c ↵
304f3565e6a430125dc809bd5505e212f0d881592b9a8e2dedb042c87428aa3331114c65e1aa8a16c5261c443a682f73cf72 ↵
d2d68a154780903a3e6036efde6e6387ee2eacc3cc7f6750e3b6f7d5a4127680086cb2f9ea1089cd27ff3f22ea95c068a6c5 ↵
b837ab3ba3e7da297e42991f92b7b268f6cf8da38758e4d58c34f839ede0b2cd71e769fb71e92ece6ff9f39884311d2bf059 ↵
fc89d5fa613f2caec0affcc7389fb164be9af55377d445f4c16914e16b657bf3f4f2dd96e70794d101fa099d1a5b5f972d32 ↵
90a1e3a6fae5b937c9775da28ca670c7e31e6a7bbd19d68068d9ae8b9281e51d5ded7211acf8e8820951fb1dc12b3e762c75 ↵
e4626d05ac093363d584bb7664000b778be0b888801a64cb3284201cc898c7c853a25d228f1f4e995dabee6f4123ba174dc0 ↵
ed7c0655b0280b57afce5e480d37d99acd2e67b77c6086a3a541596e410df5f5c5e3718fc8d57ced1060cdfe2fb146d10a41 ↵
63ced283b56acb99546e4351
145614 329
9 /bin/arch 0 1 0 c72cbb033f3daa310d9652f46f23bdf0c084709e 0 0 33261 1 3825 1132056955 1128583129
9 /bin/attr 0 1 0 88344696047568a3c5ccb9afd92f11cc2a7479aa 0 0 33261 1 7049 1132056955 1125842822
13 /bin/basename 0 1 0 66ee077e22c8a7c858a6759471ea237ec73e18b4 0 0 33261 1 15997 1132056975 1128583026
9 /bin/bash 0 1 0 d0e2a749f97ca7ec8111be4d3e74be3543253b2a 0 0 33261 1 873663 1132673303 1132673303
7 /bin/bb 0 1 0 fadd449f14294d50b76bdec6385bcd64525e9ff8 0 0 33261 1 1335223 1132056947 1124711503
12 /bin/busybox 0 1 0 24a61bc0340bff8a429f65cf3483bc317a14fcb2 0 0 33261 1 600472 1132056976 1124711503
11 /bin/bzdiff 0 1 0 55b21d7b5b4d0c2324c1aba1b7ca50aa218cbf8d 0 0 33261 1 2106 1121584266 1121584266
11 /bin/bzgrep 0 1 0 9e08187020a83d801c0b01a6eab06b961a2f8138 0 0 33261 1 1582 1121584266 1121584266
```

Code Snippet 4.3: Example: Database excerpt

4.2.3 Signing the Database

To detect tampering with the database, it is digitally signed. We use the OpenSSL [11, pp. 31–32] set of tools to generate keys since it has been peer-reviewed and is considered to be a good enough key-generator for our purposes. Generating a key need only be done once, and is accomplished via the command `openssl genrsa -out` *`filename numbits`*: this command generates a key of length *`numbits`* and places it in *`filename`*. The length of the key can prioritize security above speed since the database is verified only once, during initialization. Key components are extracted using a custom script to make the keys easily usable during the signing of the database.

The actual signing of the database is done via two utilities: `sha1sum` [18] and `yyyRSA` [65]. The former is used to create a cryptographic digest for the database, and the latter applies the RSA [57] algorithm to that digest, using the keys generated by OpenSSL, to create a digital signature. We do not use OpenSSL for generating the signature because `yyyRSA` creates a signature that contains no padding and is thus easier to decode. This signature is then prepended to the start of the database as the very first line.

It is intentional that we have separated the step of creating the database and signing it so that signing may be done after a manual inspection (if necessary), and so that it may be done offline.

4.3 Cryptography

The security provided by a file integrity checker depends on the strength of the cryptography used. If an attacker is, for example, able to find a collision in the cryptographic hash that represents the file contents, then the assurance provided by the integrity checker about the file contents being correct is lessened. Likewise, if the database is not guarded from tampering by asymmetric cryptography, the assurance that *any* content within it is valid is decreased.

Support for the manipulation of "big" numbers, hundreds or thousands of bits long, does not currently exist within the Linux kernel. The security of asymmetric encryption lies within the evaluation and manipulation of such numbers. Our options upon discovering this were to either "port" routines for big-number manipulation to the Linux kernel as has been done by SEFL and DigSig (see 2.11 and 2.6 respectively), or create the routines ourselves. The former has the advantage of being faster and more tested, and the latter has the advantage of being smaller. Hesitant about increasing the memory footprint of our kernel module, and secure in the knowledge that we only needed enough code to do decryption and that decryption would not have to be done very often, we chose to create the necessary routines.

As it turned out, the saving in terms of code size was not as great as we expected it to be, and the memory saved was offset by the memory required to store partial records after a successful database initialization (see 3.10.1). On the positive side, the big-number implementation that we created is, to the best of our knowledge, the first aimed specifically at the Linux kernel!

This big-number implementation is discussed in Appendix C since it is not central to the purpose of the file integrity checker, and it was felt that including it in this chapter would distract the reader from the more important discussions of how the database and core components of IMon have been implemented. For the rest of this section, we shall describe the basics of asymmetric cryptography and how it is used by IMon.

4.3.1 Asymmetric Cryptography

All cryptography relies on some "key" that is used to unlock a secret, transforming it from encrypted to decrypted form. Asymmetric cryptography is termed such because the key used to encrypt is not the same as the key used to decrypt. Generally, one key is kept private, and the other is made public. RSA encryption is used in our digital signature on the database; a full explanation of how the entire system works is to be found in [20, pp. 229–235] or [57], and we shall not go into the intricacies of it in detail here. However, it is necessary to understand the mathematical basis of how RSA works in order to understand an optimization that we have made.

RSA works using a ***trapdoor function***, and the security provided depends on the difficulty of finding the trapdoor value in a reasonable amount of time. The trapdoor value in RSA is the public key (if a message has been encrypted with the private key) or the private key (if a message has been encrypted with the public key).

The difficulty of finding the trapdoor value in a reasonable amount of time depends, in turn, on the number of bits of the key used: hence the need for manipulating big numbers, hundreds or thousands of bits in length. A message m may be encrypted to give the ciphertext c using the formula $c = m^x \mathbf{~mod~} n$, where the tuple (n, x) is the encryption key; the same message may be decrypted using $m = c^e \mathbf{~mod~} n$. x, e and n have a relationship that ensures that it is difficult to substitute other numbers in their place.

A small optimization that has been made is to select the exponent e used for decryption as being a small value, as described by Ferguson and Schneier in [20, p. 231]:

> Choosing a short public exponent makes RSA more efficient, as fewer computations are needed to raise a number to the power e. We therefore try to choose a small value for e. [. . .] Choosing fixed values for e simplifies the system and also gives predictable performance.

Currently, we use a 4096-bit RSA key as generated by OpenSSL [11] to secure the database from tampering. The e that has been chosen is the fixed value 10001_{16} (65537_{10}). A 4096-bit key is thought to be large enough to make a brute-force attempt at finding the correct trapdoor (that is, the private key) infeasible.

4.3.2 Verifying IMon Files

IMon needs to know the public key – specifically, the e and n components – to be able to decrypt the digital signature placed on any IMon files, such as the database. This key is compiled into IMon as a hexadecimal string, and translated into a structure usable by our big-number implementation (see C.1) at runtime.

To verify a file called `filename`, we first obtain the internal kernel data structure associated with the IMon directory. We then test to see whether either `filename` or `filename.old` exist, in that order – see 3.9 for why this is done. If neither exist, we assume that the system has been compromised and return an error. Otherwise, we verify that the file has not been tampered with by validating the digital signature within it, returning successfully if the digital signature is correct and returning an error if it is not.

4.4 Database

The database component of IMon initializes an array of records (which is our in-memory database) from a database file, stores stub records and performs on-demand loading (see 3.10.1), handles reference counting transparently, and releases memory associated with the database when IMon is shutting down. By making extensive use of the C preprocessor, the database component is both compilable as part of IMon and as a standalone program that simulates requests for records and tests the database lookup process.

In this section we describe the implementation of the database. We begin by describing the data types that represent a stub and full record, and then describe the interface by which the database is used by the Core component discussed in 4.1.2. We then describe the workings of core functions that initialize the database, release memory associated with it, and search for a record. Lastly, we discuss the locking strategy used by the database to ensure that race conditions do not occur.

From this point on, we shall use the following terminology:

record A single set of metrics for a file. The exact data structure that holds these is noted in 4.4.1. A record may either be **full** or **stub**, depending on whether it is populated with all the metric data of the file or just some of it.

disk-database The file that contains the complete records for each file.

database The in-memory array of records.

4.4.1 Records

The record data structure must be small so as not to waste memory, but flexible enough to be modified should one wish to test a new metric (such as a secondary file digest or hash). We have achieved this by making the stub data structure (imon_security) very small, and placing all data that is not strictly necessary into a second data structure (imon_data) that the stub holds a pointer to.

```
001 struct imon_data {
002   unsigned char hash[HASH_LEN]; /* non-NULL-terminated SHA1 160-bit hash */
003   mode_t perms; /* read-write-execute bits */
004   unsigned int uid, gid; /* File user and group */
005   int links; /* number of links */
006   size_t size; /* size in bytes */
007   time_t ctime, mtime; /* creation/change time, modification time */
008   char policy;
009   atomic_t refs; /* for database: number of references extant */
010   unsigned int num_deps; /* number of dependencies */
011   struct imon_security **dep; /* array of pointers to dependencies */
012 };
013 struct imon_security {
014   char *filepath; /* absolute path */
015   unsigned short flags; /* per-file flags */
016   unsigned int line_no; /* Which line is it on in the DB file? */
017   struct imon_data *full; /* what is the ideal? */
018 };
```

Code Snippet 4.4: Database data structures

The core data structures for the database are depicted in Figure 4.3, which reflects Code Snippet 4.4. In this Figure the overall database structure is abstracted such that each record is represented by an "R", with the index number of the record listed below it. If the record is full, then a full circle is shown next to it; if it is a stub, then the circle is empty. All stub records have NULL imon_data pointers, as shown by the lack of "..." in the boxes that they point to. Populating the pointer-to-imon_data within an imon_security structure results in a full record. A dependencies array, which is part of the imon_data structure, is shown pointing to various other records in the database array.

imon_security, shown on lines 13–18 of Code Snippet 4.4, consists of the absolute path of the file (filepath), per-file flags (flags), a number indicating the byte within the database file at which the record data begins (line_no), and a pointer to more

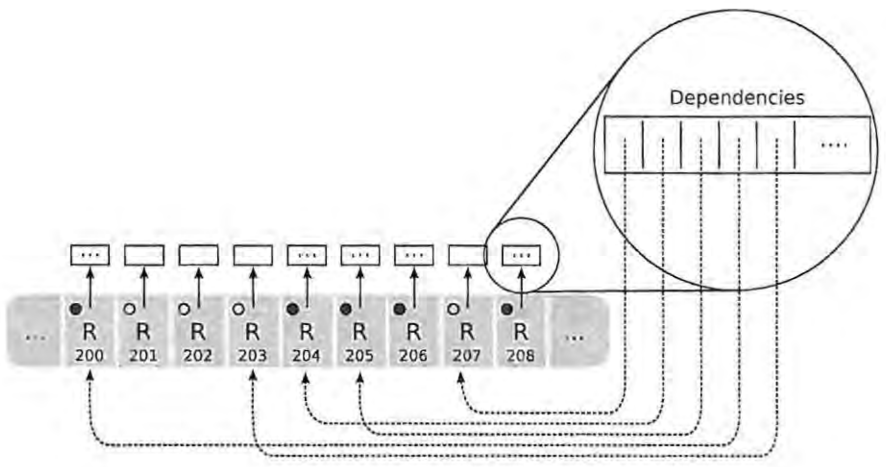


Figure 4.3: IMon Record Database Structure

data (full). All data in the stub, with the exception of the full member, are set at initialization time.

The creation of a stub set of records with initialized dependencies is made easier by Step 2 of the baseline creation process (see 4.2). We are able to generate the total number of records, sort them, and include much information that removes the need to ever handle more than a single record at a time. For example, each filename string is preceded with its length, making it easy to read in a set number of bytes and call it the filename without having to resort to using "special" delimiter characters to show where a filename begins and ends. This can be seen in Code Snippet 4.3.

It is important to note that in Step 2 of the baseline creation process we order records by filename, as mentioned in 4.2. Since we read each record in-order, we end up with an initialized database that is sorted. This fact is used to greatly increase record lookup speed, as described in 3.10.2.

imon_data, shown on lines 1–12 of Code Snippet 4.4, holds baseline and non-essential data and is initialized on-demand. Importantly, this structure can be added to and extended by simply adding fields and creating the appropriate functions to test those fields: the rest of the program need not change at all. Due to the on-demand loading and unloading, the size of imon_data can be increased without affecting the memory impact of IMon overmuch. The current set of metrics that IMon tests, and the lines of Code Snippet 4.4 that they are shown on, are:

hash, line 2 A SHA1 160-bit hash (currently). HASH_LEN refers to the size of the hash in bytes. By changing two lines of the source code, not shown above, the hash used can be altered to be any crytographic hash supported by the Linux kernel. As of kernel version 2.6.12, this list consists of four algorithms at various digest lengths that have been created specifically as hash functions: MD4, MD5, SHA1, SHA256, SHA384, SHA512, Whirlpool-256, Whirlpool-384, Whirlpool-512, Tiger-128, Tiger-160, and Tiger-192. Any of these may be easily substituted for SHA1.

perms, line 3 The mode of the file, expressed in terms of read-write-execute bits for owner-group-other entities.

uid, line 4 Numeric representation of file owner identity.

gid, line 4 Numeric representation of file group identity.

links, line 5 Number of hard links made to a file.

size, line 6 File size in bytes.

ctime, line 7 Time of last status change.

mtime, line 7 Time of last modification.

Non-metric-related fields of imon_data are:

policy, line 8 The policy (see 3.5.2) associated with this file.

refs, line 9 A reference counter used to reduce the number of times the disk-database needs to be accessed (see 3.10.1).

num_deps, line 10 The number of dependencies (see 3.2) that this file has.

deps, line 11 An array of pointers-to-dependencies.

4.4.2 Database Interface

The database component of IMon presents an interface through the following four functions, which are discussed in detail in subsequent sections:

int initDB(struct file*) This function initializes the database from a specified disk-database. It allocates memory for storing database stub records, reads record stubs into the database, and tests the database for consistency before returning successfully.

void freeDB() This function releases all memory related to the database.

struct imon_security* get_record(const char*) This function returns a specified record, or NULL if no record is found. It will act to load a full record if only a stub exists in the database, and increments the record's reference count in any case.

void put_record(struct imon_security*) This function releases a record by decrementing its reference count and, if the reference count reaches zero, freeing the memory that differentiates a full record from a record stub.

4.4.3 initDB

The initDB function initializes the database from the disk-database. It is called during IMon's initialization phase (see 4.1.3) and can be seen as having three phases:

1. Setup: preparation is made to read in records efficiently
2. Initializing: records are read in
3. Testing: the database is validated for consistency

Setup

By the time that initDB has been called, the database has already been declared either valid or not (see 4.1.3); if invalid, then initDB is never called. Therefore, the first thing that we do in the function is read past the digital signature at the top of the file. The second line of the database consists of two numbers: the first specifies the total number of records in the file, and the second specifies the maximum length of a record. This makes reading records exceptionally easy, as we have merely to loop until all records have been read in, and can read each record into a holding buffer that we know will not overflow as we know the maximum size that a record could possibly be.

Allocating memory for the record buffer is easy: we shall only need a few hundred bytes, and this is easily given to us by the kmalloc function of the kernel. However, allocating memory to the database is not as easy to do since kmalloc returns a contiguous memory block, and contiguous memory is difficult to find as the amount requested grows larger – resulting in kmalloc returning an out-of-memory error. Three options present themselves at this point:

Minimize imon_security size imon_security currently holds members that could be reduced in size, at the cost of limiting their range. An example is the policy and flags member variables: since flags is simply a bitset, it could be set to use only as many bits as there are flags; and since policy may be encoded in only three bits (assuming that more policy-types are not implemented), the two could be combined to form a single int-type variable. Alternately, a C bit-field specification[1] could be used to achieve the same effect. So far, we would have gained space by removing bits that would never have been used; however, if we are willing to remove bits that could be used the potential savings increase. For example, if we are willing to reduce the maximum number of references that can be extant, or the maximum number of dependencies, then these members can be changed from the int type to the short int type.

The primary drawback of this approach is that we will be presented with exactly the same problem at a later date, as more and more records are added: the approach does not scale. In addition, the system becomes increasingly inflexible as we reduce the range of member variables to what we believe to be reasonable – which may not be what another user of the system considers to be reasonable at all.

Allocate in smaller blocks Instead of allocating a single list, we could allocate several smaller lists and sort each one. Alternately, we could make imon_data consist of nothing but the filepath, and create a new structure, perhaps called imon_stub, which contains stub data. This approach would result in a greater total amount of memory being allocated since the size of each pointer-to-imon_stub would have to be factored in; however, the size of each *individual* allocation would be smaller, and thus more likely to succeed in terms of kalloc requirements.

The primary drawback of this is that it is a workaround that actually *increases* the amount of memory allocated, and makes accessing members of imon_security more difficult: one would now have to go through another level of pointer indirection to get to needed data. Another disadvantage is that, once again, this simply puts off the problem until a later date, when allocating an array of n objects – no matter how small – will become problematic.

[1]such as "unsigned int policy:3;"

Use a different allocation function kmalloc, and functions that interface with it, are not the only memory allocation functions that we could use. Another function that fits our needs is vmalloc, which allocates *virtually*-contiguous memory space; in other words, the memory that is presented to us is not necessarily contiguous, but is made to seem so through use of the hardware ***paging unit***.

The main drawback of this approach is that memory access to areas allocated with vmalloc is generally slower.

Since the last option is the only one that scales well and allows us to maintain code simplicity and readability, we choose to use vmalloc to allocate the database array.

The last thing that we do as part of preparing to read in records is to initialize the database to be in a defined "uninitialized" state. When we test for consistency later on, this makes it easier to detect if a certain element has not been initialized by seeing if it is still in the defined "uninitialized" state.

Initializing

Using a for-loop, we iterate through the disk-database and initialize each element in turn. For each element, the filepath, per-file flags (flags), and the line number (line_no) at which further metrics may be found are set.

Testing

Testing is done by iterating through each record in the database and testing to see whether the filepath member is uninitialized. Since it is set to be uninitialized during the setup phase, this test should be able to catch any errors in the database file. Whilst testing could be made more extensive and comprehensive we rely on the assumption that the creator of the database would have checked it before signing it, and that the digital signature is a stamp of approval as well as a verification of authenticity. Therefore, we only test for completeness and consistency rather than checking for a deliberate mangling of the database.

If the database passes all tests, we set a global variable called imon_db_usable to be equal to one. This variable is used in IMon to determine when it is safe to obtain a record; when it is set to zero, IMon will not attempt to get a record from the database.

4.4.4 freeDB, get_record and put_record

These three functions are grouped together as they are not complex to explain, nor is much discussion of their workings necessary.

freeDB

freeDB goes through the database sequentially, releasing all memory associated with each record; at the end, the memory allocated to the database itself is released. After a call to freeDB, the database refuses to return any records until a successful call to initDB has been made.

get_record

get_record takes a const char* as the pathname to find. It does a binary search (as described by Knuth in [31, pp. 409–414]) of the database to find the correct record. A binary search can be used because, as mentioned in 3.10.2, the database file is sorted lexicographically by file path. If no record is found, NULL is returned. If a matching record is found, the reference count is incremented and a check is performed to see if the record is a stub: if it is, we read in the full record using a function called read_imon_record and return it, otherwise we simply return the record as-is.

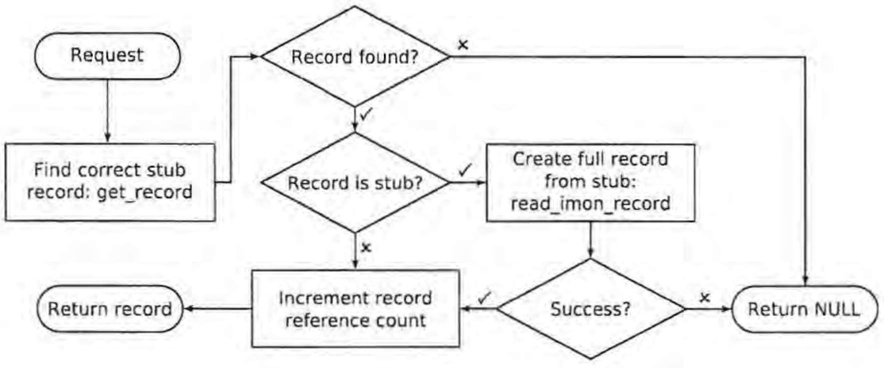

Request
Record found?
×
✓
Find correct stub record: get_record
Record is stub?
✓
Create full record from stub: read_imon_record
×
Return record
Increment record reference count
✓
Success?
×
Return NULL

Figure 4.4: Requesting a record

Figure 4.4 shows the request process, *sans* locking (see 4.4.5), diagramatically.

put_record

put_record takes a pointer-to-imon_security as an argument and decrements its reference count. If the reference count is zero, the FLG_MMAPW flag (see 4.5.4) on the record is unset and the record is turned into a stub by releasing the memory associated with the full record.

4.4.5 Locking Strategy

If compiled for a system capable of symmetric multi-processing (SMP), or if compiled with support for preemption, the Linux kernel is effectively multi-threaded. This means that shared data structures such as the database must be protected against concurrent modification, and necessitates the use of appropriate kernel synchronization methods. In this discussion of database locking strategy, we assume that the reader is familiar with such concepts as concurrency, deadlock, and critical sections; such concepts can be reviewed by reading through [40, pp. 119–130].

As described by Love in [40, pp. 119–156], the synchronization primitives provided by the Linux kernel are extensive, covering ***atomic*** bit-operations through memory barriers and per-CPU preemption-disabling to spinlocks and semaphores. A full explanation of all of these (and more) is provided in the aforementioned reference work. In this section, however, we shall content ourselves with discussing only those synchronization primitives that are most pertinent to our specific synchronization problems, and ignore topics such as per-CPU preemption disabling and condition variables.

Spin Locks A spin lock is an extremely lightweight lock which may be both acquired and released with very little overhead. Upon finding the critical section unlocked, a spin lock claims the lock and enters. Upon finding the critical section locked, the spin lock will *spin* – that is, enter a tight loop during which the lock status is tested repeatedly, and during which no other task may run[2] – until the critical section may be entered once more; at this point, the lock is

[2]This is true on SMP machines; on uniprocessor machines, a spin lock simply disables preemption so that the lock may never be contended.

claimed and the section is entered. Locking a spin lock more than once results in deadlock.

Since a spin lock takes up processor time whilst spinning, it is not good to hold a spin lock for a lengthy period of time. There are certain functions that may block until a resource is available: these functions are said to be able to *sleep*. Calling a function that may sleep whilst holding a spin lock is not advisable since it may be necessary for another task to release resources in order for the sleeping function to gain them – and since a spin lock effectively disables preemption, the sleeping function will never be able to obtain the necessary resources and a system freeze will occur.

Semaphores A spin lock busy-waits until the critical section may be entered; a semaphore sleeps instead, periodically waking up to see whether the section may be entered. This is the fundamental distinction between the two, and one of the side effects of it is that semaphores may be used whilst sleeping functions are called. The overhead incurred by sleeping, waiting, and waking up means that semaphores impose a far greater synchronization penalty than do spin locks, and should therefore only be used either when spin locks cannot be used or when the lock is to be held for a long time [40, pp. 143–144].

The locking strategy employed does not use semaphores, but it could conceivably have done so. Semaphores are described here so that one may compare them to spin locks and thereby understand why the spin lock synchronization primitive has been chosen as the speedier alternative.

Atomic Operations Atomic operations ensure that a read and write to a memory location either occurs as an atomic transaction or does not occur at all. For example, they ensure that a given integer, if incremented concurrently by two threads of execution, will have the correct value. Atomic operations are carried out on variables of type atomic_t, which are integers that are "wrapped" by structs to ensure that it is difficult to accidentally modify them non-atomically. An example of an atomic integer that has already been seen is the refs member variable shown on line 9 of Code Snippet 4.4.

Both semaphores and spin locks come in default and reader-writer varieties. The default versions allow only one thread of execution to be within a critical section at a time, as is expected of synchronization primitives. Reader-writer versions allow one to gain a reader-lock or a writer-lock; any number of threads may obtain reader-locks at the same time, but only one thread of execution at a time may hold a writer-lock, and no reader-locks may be held whilst a writer-lock is held. Reader-writer locks therefore allow for a safer form of concurrency and improved performance.

We perform no locking during the database initialization phase, reasoning that at this point the database is not yet in use – and, therefore, there is no danger of concurrent operations causing corruption within it during initialization. It is during get_record, read_imon_record, put_record and freeDB that care must be taken.

Locking Primitives

There are two primitives that we use in our locking strategy: a reader-writer spin lock (imon_db_lock) and an atomic_t variable called imon_db_usable. When imon_db_usable is set to zero, it is assumed that the database is being freed and is therefore not usable; after initialization, imon_db_usable is set to 1, and it is only in the freeDB method that it is set to zero. In most places where a lock is obtained, the first thing done is a check to see whether imon_db_usable is unset, though this is usually not mentioned explicitly in what follows for reasons of brevity and clarity.

imon_db_lock is used to protect the consistency of the database during most operations, and will be referred to the most in the following subsections.

Locking: Reading in a Record

During the binary search for a matching record, a reader-lock is held. If the search is successful and the record already exists, the lock is dropped and the correct record is returned. If the record is not found, the lock is dropped and NULL is returned. These are the simple cases; the complex case occurs when the record is found, but has not yet been populated.

In the complex case, the reader-lock is dropped, and the read_imon_record function is entered. A writer-lock is now obtained. At this point, we check to see if the record has already been populated by another thread of execution, and exit successfully if so. Otherwise, a copy is made of the stub record to be obtained, and the writer-lock is dropped. The spin-lock cannot be held whilst the record is being populated as this involves allocating memory and reading from disk, which are both operations that may sleep. The imon_data field of the copy is now filled in, and a writer-lock is again obtained. Once again, we test to see if the record is already populated, returning successfully if so. Otherwise, we populate the real record entry with data from the copy, drop the writer-lock, and exit successfully. In this fashion we never hold the writer-lock for any operation lengthier than a couple of tests and an assignment.

Note that since a semaphore can be held whilst a function that may sleep is called, the use of semaphores instead of spin locks could have reduced the complexity of locking in read_imon_record, but would have incurred a greater performance penalty.

Locking: Releasing a Record

In put_record, we obtain a writer-lock, release memory (if necessary), and drop the writer-lock. The lock is never held for more than the length of a couple of tests and the time taken to release non-stub memory.

Locking: Freeing the Database

As mentioned above, the first thing that freeDB does is set imon_db_usable to zero. After this, all memory associated with all records is freed in a tight loop that traverses the database and, for each record, obtains a writer-lock, frees any non-stub memory associated with that record, then drops the writer-lock. Finally, a writer-lock is obtained and all remaining memory – the filepaths of the records, and the memory used to create the database array – is freed. At the end, the writer-lock is dropped.

4.5 Integrity Checking

This section deals with the part of the system that actually performs integrity checks, and the logic that underlies those checks. Consequently, it is the largest section in this chapter. Before discussing this, however, it is necessary to understand some aspects of the Linux Security Module (LSM) [13, 12] framework and the processes of execution and file access on a Linux system. Only as much detail as is necessary to understand the workings of IMon is presented. After examining the LSM framework, we explain how the Linux kernel handles a request to execute a program, and how the Linux kernel handles a request to read from or write to a file. This is necessary so that the reader can understand where in these respective processes we are able to intervene using an LSM-provided hook.

Once we have covered the background necessary to understand how IMon fits into the existing processes of the Linux kernel, we move on to discussing problems related to memory-mapping and incorrect file permissions that have arisen during the implementation and our solutions to them. After this, we briefly describe a per-process data structure that we have found necessary, and the reasons why it is necessary in our implementation.

At this point we are ready to examine the functions that IMon registers to be called when certain LSM hooks are reached. Each of these functions is described in detail, with flowchart diagrams being provided whenever necessary to clarify the underlying algorithm behind the code and description. Utility or helper functions used by IMon are then described.

The testing functions of IMon first apply policy and then test files; the policy applied depends on the currently-executing executable. We describe these functions next, followed by a section on a function that takes action once an integrity check has been failed. Lastly, we discuss transitive interpretation (see 3.3) and trace through an example that demonstrates how IMon handles it.

For reasons of brevity and clarity, whenever IMon code is discussed certain elementary checks – for example, testing that the pointer to a structure passed is non-NULL – have been omitted from the descriptions below. Note further that whenever an operation is denied, the function immediately returns without considering any further checks; again, for reasons of brevity and clarity, this may be not explicitly mentioned in the text.

4.5.1 Linux Security Module Framework

The Linux Security Module (LSM) framework, explained more fully in [13, 12], consists of a series of "hooks" in the Linux kernel: places at which security functions, initially dummy functions, may be run. Also provided are `void*` pointers within certain structures that allow data related to security to be saved there. Hooks are placed strategically at places where information is available to make an informed decision about whether to allow or deny a given operation, and in this fashion LSM provides an excellent way to support a number of different security models.

Implementation-wise, the hooks are provided as a structure of function pointers and functions that call them at the correct point. This is best illustrated through an example.

```
001 struct security_operations {
002 /*...*/
003     int (*bprm_alloc_security) (struct linux_binprm * bprm);
004 /*...*/
005 };
006 extern struct security_operations *security_ops;
007 static inline int security_bprm_alloc (struct linux_binprm *bprm)
008 {
009     return security_ops→bprm_alloc_security (bprm);
010 }
```

Code Snippet 4.5: LSM Example, `security.h`

Code Snippet 4.5 consists of excerpted lines from the file `security.h` in the Linux kernel tree. This file defines a structure called security_operations which contains the function pointer seen on line 3, bprm_alloc_security, among other function pointers not included in this example. Line 6 indicates the existence of a global variable called security_ops, which is of type security_operations*; this global variable points to the table of security-related functions that will be called. Lines 7–10 show the function security_bprm_alloc, which serves as a "wrapper" for calling the function pointed to by security_ops→bprm_alloc_security. The wrapping process is shown in Figure 4.5.

In context, this is called by two functions which are responsible for executing files: we shall examine one of them[3], do_execve, here. Sections of this function are shown in Code Snippet 4.6 in a simplified form.

```
int do_execve(/*args*/)
{
/*...*/
    retval = security_bprm_alloc(bprm);
    if (retval)
        goto out;
/*...*/
out:
```

[3]The other function, compat_do_execve in `compat.c`, is a backwards-compatible version of do_execve that handles 32-bit userspace-supplied pointers.

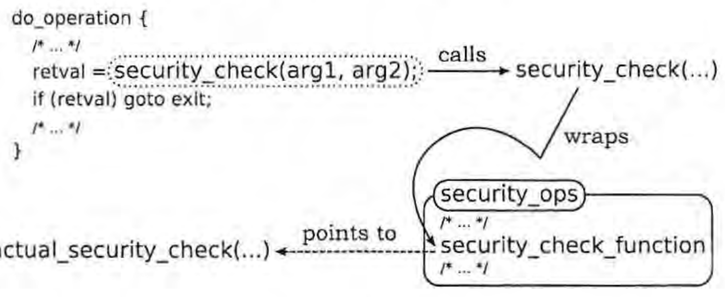
```
009     return retval;
010 }
```

Code Snippet 4.6: LSM Example, exec.c

Line 3 replaces a large portion of the error-checking code that the do_execve function contains. By the time that the flow of control reaches line 4, where security_bprm_alloc is called, basic sanity-checks have already been passed. security_bprm_alloc, as we have seen, is simply a wrapper that calls the appropriate function pointer and returns the result. On lines 5–6, we see that if a non-zero result is returned by this function pointer, we leave the function immediately, returning the error value returned by the security hook. Thus the security hook acts to deny an operation.

The paradigm of doing a security check after basic error-checks is repeated throughout the Linux kernel in much the same fashion as shown above. In fact, a security module may not get to see all attempts made to perform an operation, since some attempts could be deemed invalid on grounds other than those of security – for example, a file that one tries to access could be a symbolic link whose target does not exist. If a sanity check is failed, no further checking is done and the function is exited as soon as possible.

By default, when no security module is registered with the system the security function that is actually called does nothing but return a default value – generally, this is a value that indicates success and allows the operation to proceed. "Registering" a security module therefore means setting selected function pointers in the security_ops global variable to point to one's own functions; a utility function, register_security, is provided by LSM to do this for us.



```
do_operation {
  /* ... */
  retval = security_check(arg1, arg2);
  if (retval) goto exit;
  /* ... */
}
```

Figure 4.5: Linux Security Module Framework

Figure 4.5 shows this process graphically. Note that, in reality, actual_security_check may return its value directly to the do_operation function; in the diagram, the value passes through the wrapper function, security_check, first. However, since the wrapper function simply consists of a return statement, it is possible that an optimising compiler could replace all calls to it with a call to the function pointer instead. In any case, including such details serves only to make the diagram less clear, and they have therefore been omitted.

Summary

To summarise, there are two things which should be understood about LSM in order to see how the framework is used by IMon.

1. LSM provides void* pointers in various structures that provide a way to attach security fields to those structures.
2. LSM provides hooks that allow a security function to deny what would otherwise be permitted.

Another point to keep in mind as being very important in the explanations that follow is that the currently-executing process may always be obtained by IMon through the current variable; this is restated for emphasis in 4.5.2.

The exact workings of the hooks, as explained earlier, are important to know so that one can understand why registering with the LSM framework is necessary, why IMon might not get to see all file accesses, and so forth.

IMon security functions

The most important LSM-provided function pointers that are registered for file integrity use by IMon are:

bprm_alloc_security This is called at the beginning of program execution, before the search for a binary handler – a function that understands the file format – has started.

bprm_free_security This is called unconditionally if the search for a binary handler has ended successfully.

bprm_check_security This is called at the point before various binary handlers are tested to see if they understand the binary format. It is passed a linux_binprm structure which represents the binary that is to be executed, and data (such as environment and command-line arguments) that is associated with that binary.

file_alloc_security This is called when memory allocation is being done for a file structure, before it is to be used.

file_free_security This is called when memory is being released for a given file structure.

file_mmap This is called when a file is memory-mapped. Linux uses memory-mapping to load pages from binary executables, scripts, and shared libraries, making this an excellent point at which to ensure that the file that is being memory-mapped is valid. One of the arguments passed to it is the file that is to be mapped, which may be NULL to indicate that the memory-map is not backed by any file at all.

file_mprotect This is called when the protection applied to a memory-mapped region is changed. The protection of a region refers to the operations – read, write, and execute – that may be performed on that region.

file_permission This is called by functions that read or write to an open file. It is passed the file that the operation is requested for, and a bitmask that indicates the read, write and execute permissions requested for that file.

task_alloc_security This is called when a process is created.

task_free_security This is called when a process is destroyed.

capable This is called by functions that wish to request a process's ***capability set***.

Less important function pointers are used for self-protection (see 3.8) and occasionally to unset the cache-flag on a file. These are:

inode_create This is called whenever a regular file is to be created.

inode_unlink This is called whenever a regular file is to be deleted.

inode_rename This is called whenever a regular file is to be renamed.

inode_permission This is whenever a request is made to open a file for reading or writing.

inode_link This is called whenever a hard link is to be created.

inode_setattr This is called when certain attributes of a file are changed.

file_lock This is called whenever a request is made to lock a file; that is, to deny access to a file to any other process.

The less important function pointers are not discussed in detail. It suffices to say that they forbid certain operations (such as truncation, deletion, and locking) to be performed on the IMon database or within the directory used to store IMon-related files, and that they may unset FLG_CACHED as necessary.

All function pointers that are registered for use by IMon are set to point to functions named using the scheme imon_<funcptr_name>; for example, file_mmap_security points to imon_file_mmap_security. IMon functions that are not called by security hooks are not subject to this naming policy.

4.5.2 Executing a Program

Linux uses the following simplified process when executing a program:

1. Allocates memory for a linux_binprm structure, and initializes said structure. linux_binprm will hold data necessary for execution to occur such as the interpreter of the executable, environment variables, and arguments passed. As part of the initialization process, the bprm_alloc_security hook is called.

2. Calls the search_binary_handler function, which attempts to find the correct way in which to load the file format. The "correct way" may involve invoking another file as an interpreter for this one, in the case of special formats or script files. The bprm_check_security hook is called from this function.

   Each known executable file format can be registered with the kernel, and is then associated with a structure that contains, among other things, function pointers to functions that can handle that format[4]. Therefore, searching for a binary handler becomes a case of cycling through each known format and

[4]It is instructive to compare this approach to that detailed in 4.5.1; the same technique for runtime insertion of code is used there.

calling the load_binary function pointer until a handler returns success or all handlers fail.

3. The functionality of load_binary does not vary too much across machine-code file formats, each of which resorts to calling do_mmap in order to load the file into memory – and do_mmap invokes the file_mmap hook.

   Interpreted script files are handled slightly differently: the script handler invokes the correct interpreter, which is loaded with the sh_bang member of its associated linux_binprm structure set to a value greater than zero: this indicates that the program is being loaded as an interpreter. Note the implicit recursion of this approach: at some point, a native-code interpreter *must* be found to interpret a file, which may in turn interpret another file, and so forth. As soon as such an interpreter is found, the recursion terminates. The machine-code interpreter, upon being successfully loaded into the current process space, is passed the filename to be interpreted as if it were a command-line argument.

4. If the search for a binary handler is successful, the process currently running is replaced by an image of the new process, after which the bprm_free_security hook is called.

5. If the search for a binary handler is *not* successful, the bprm_free_security hook is called only if the security member of the linux_binprm structure has been set by bprm_alloc_security.

The above is derived from the do_execve, search_binary_handler and the various load_binary functions in the Linux kernel tree (source files `exec.c` and `binfmt_*.c`), and is documented in more detail by Love in [7, pp. 678–683]. A more detailed and expansive look at the execution process is described in 4.5.10 which includes the explanatory Figure 4.10.

The process that is currently executing may always be obtained by IMon through the current variable. Furthermore, Linux does not load the entire file into memory at once, but only maps certain sections that are being used. As more of the executable is needed, it is mapped using the do_mmap function, which calls the file_mmap hook; this holds true for all types of executable files.

4.5.3 File Access

To the Linux kernel, a "file" is an entirely temporary abstraction created to model the interaction between a process and a filesystem object; it is created when the object is opened, and destroyed once the object is closed. In the interim, it is represented by a file structure. Files as they exist on a filesystem are represented by inode structures, and linked to directories through dentry structures. file structures contain data that links them to the relevant inode, which stores actual data (such as file size, mode, and owner) about the file, and to the enclosing dentry, which can be followed to obtain the full path of the file.

Linux supports multiple filesystems which are unified through a layer known as the Virtual Filesystem (VFS). This abstraction layer does basic management of data related to filesystems and is the layer that LSM hooks are called from. If a security check is failed at this layer, it is possible that the underlying filesystem-specific non-VFS code never even sees the request.

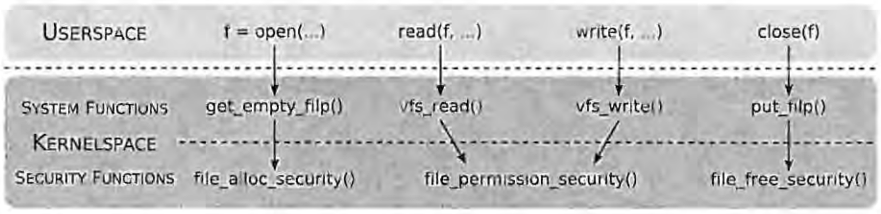


Figure 4.6: File access

The simplified life-cycle of a file structure can be seen as follows:

1. A request is made to open a file: the get_empty_filp function in `file_table.c` is called. Among other things, this function calls the file_alloc_security hook. get_empty_filp allocates memory for a file structure, and returns it.

2. During the lifetime of a file, read, write, seek, and other operations may be performed upon it. We are concerned only with read and write operations, which occur as follows:

   (a) When a read request is made, vfs_read in `read_write.c` is called. In this function, the file_permission_security hook is called with the permission mask set to MAY_READ.

(b) When a write request is made, vfs_write in `read_write.c` is called. In this function, the file_permission_security hook is called with the permission mask set to MAY_WRITE.

3. When the process closes the file, put_filp in `file_table.c` is called. As part of releasing memory associated with the file, this function calls the file_free_security hook.

Figure 4.6 depicts the abovementioned life-cycle. It is split up into two main sections: userspace and kernelspace. Kernelspace is further categorized into system and security functions, with the security functions being the LSM hooks that are called. When a user uses the `open` system call, `get_empty_filp` is called, which as part of its normal workings calls `file_alloc_security`; this is point (1) in the enumerated list above. (2) and (3) are similarly depicted, with the same security function being called by both `read` and `write` calls.

This process as well as other VFS processes are documented more fully by Love in [7, pp. 374–420].

4.5.4 Memory-Mapping and File Permissions

A file may be written to in two ways: by opening it in the appropriate mode and writing to it, or by memory-mapping it with the appropriate protection and writing to it. Whilst the first method invokes the LSM file_permission hook, the second does not; and, therefore, we have no way of telling whether a file is being written to once it has been memory-mapped writably.

For a writable memory-map to be successful, a file's permissions must allow it to be written to by the user who wishes to memory-map it. However, the super-user account (`root`) is allowed to write to a file no matter what the permissions of the file are – and whilst we can stop this from the file_permission hook for direct writes to a file, we have no way of even detecting it from within the LSM framework for a memory-mapped file. Fortunately, we can reduce the capabilities of the super-user account from the kernel (see 4.5.6) so that this is no longer a problem.

Another facet of the issue is that many Linux systems are by default set up with file permissions that grant the super-user account to write to shared libraries. This is not necessary and a well-configured, secure Linux system that follows the principle of least privilege[5] has shared libraries set with permissions that deny writing to them by anyone.

We are faced with the following two alternatives for dealing with writably memory-mapped files:

Patch the kernel source It might be possible to patch the kernel source code to add in a call to the correct LSM hook at the appropriate places. This removes the problem entirely, but involves stepping outside of the LSM framework and risking having to maintain a number of different patches, one for each kernel version or variation.

Don't cache If we refuse to cache file integrity checks for writably memory-mapped files, we will always test them. Therefore, it does not matter whether they are written to and we cannot detect the write, since a subsequent test will be able to detect any changes. This increases the overhead imposed by IMon on the system since there are certain files which may be tested repeatedly, whether they have really been updated or not, on the mere suspicion that updates might have occurred.

Neither of these options is attractive. We have chosen to implement the second option as we are not certain that the first is practical: there may be race conditions and side-effects (which could change from one kernel version to the next!) that we are not aware of that would affect the logic of IMon in unknown ways; furthermore, we may not have all the information required to make an informed judgment about allowing or denying an operation at the point that a memory region is written back to disk.

To implement the second option, we set a runtime flag (FLG_MMAPW) which indicates that the file is mapped writably. This flag inhibits caching for the file, and is only unset once no further references to the file exist.

4.5.5 Per-process Data

To deal with transitive interpretation (see 4.5.10) and policies such as **accept_after** (see 3.5.2) that require keeping track of which files have been opened by a specific process – as opposed to which files have been opened by *any* process – we

[5]A well-known security principle, this states that an entity should be assigned the lowest privilege-level that still makes it possible for a task to be accomplished.

attach a small amount of per-process data to each process. This is done via the imon_task_alloc_security and imon_task_free_security functions.

```
struct imon_procsec {
  const char* script;
  u8* deps_checked; /* bitmask */
};
```

Code Snippet 4.7: Per-process data structure

The per-process data structure is shown in Code Snippet 4.7, and consists of only two member variables. The first, script, is set to point to the full path of the script being interpreted, or NULL if there is no script being interpreted by this process. The second, deps_checked, indicates how many dependencies have been checked by this process; it is only used by the **accept_after** policy at present.

The deps_checked array of 8-bit characters requires some explanation. If a dependency is opened then it should be listed as having been checked by the process. However, we cannot keep a simple count of the number of dependencies that have been checked: if the same dependency is opened *twice*, the count would be artificially increased. Therefore, we must also retain information about *which* dependencies have been checked. We can do this by setting bits in an array; for example, if the third dependency on a file's dependency list has been opened, then we can signify this by setting the third bit of the first character in the array. Similarly, if the twelfth dependency has been opened, then we can signify this by setting the fourth bit of the second character in the array – which is equivalent to the twelfth bit of the array. Opening a single dependency twice now simply sets the same bit twice, and does not contribute to the number of dependencies checked. Finding the total number of dependencies set is as easy as counting the number of bits in the entire array that have been set.

4.5.6 Integrity-Checking Hooks

This section looks at the functions that are called via hooks mentioned in the course of the processes detailed in 4.5.2 and 4.5.3, and the logic followed by each hook. To decrease the amount of repeated code, any sufficiently complex code-path that is used by more than one function has been turned into a utility function (see 4.5.7); to increase the maintainability of the code, only *hook-specific* testing is carried out in each hook, and the application of policy and testing metrics has been centralized into two separate functions (see 4.5.8). This approach has led to the code within each hook being short and readable, and has also made each hook quite simple to describe.

imon_bprm_alloc_security

This function is passed a pointer-to-linux_binprm as an argument, arg_0. It obtains the name of the file that is to be loaded from this argument's file member, then attaches a database record to the file's f_security member. If no database record exists, an error is returned (in which case execution is denied). By the end of this function, arg_0→file→f_security is either NULL (if no matching record is found) or initialized to point to the correct record.

Note that we set arg_0→file→f_security, and *not* arg_0→security. Accordingly, by the process detailed under 4.5.2, if a binary handler is not found then imon_bprm_free_security will not be called; in this case, the security field is released by a call to imon_file_free_security made when the kernel releases the file member of arg_0.

imon_bprm_free_security

This function is passed a pointer-to-linux_binprm as an argument, arg_0. We first allocate a per-process data structure (if necessary), and alias current→security to psec. If the arg_0→sh_bang is non-zero and psec→script is NULL, we can assume that the current process will be replaced with an interpreter for the current file (see Step 3 of 4.5.2); if this is the case, then we set psec→script to be the current filename, as pointed to by arg_0→security, thereby indicating that it is either an interpreter or a transitive interpreter.

Note that we need not release arg_0→file→f_security in this function since it will be released when the file structure arg_0→file is destroyed.

imon_bprm_check_security

This function is passed a pointer-to-linux_binprm as an argument, arg_0. Its workings are graphically represented in Figure 4.7.

If arg_0→file→f_security is NULL, one more attempt is made to populate it; if this

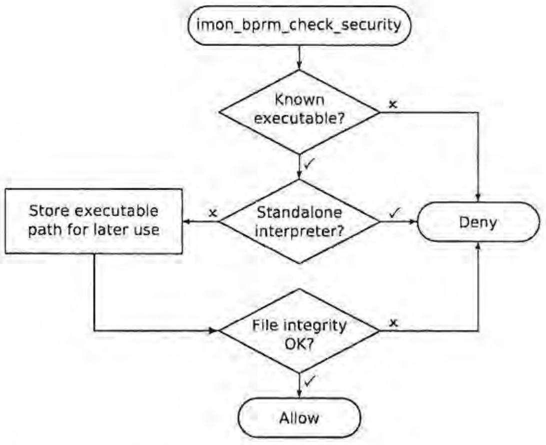


Figure 4.7: imon_bprm_check_security

fails, then execution is denied on the grounds that the executable is unknown to IMon. If the flags associated with arg_0→file indicate that the file is an interpreter, and neither arg_0→sh_bang nor current→security→script are set, then we deduce that the file is neither being executed as an interpreter nor is it part of a chain of transitive interpretation; on the grounds that it must, therefore, be a standalone interpreter, we deny execution.

If arg_0→security is NULL and it is not registered as an interpreter (current→security→script is NULL), then this might be an interpreted file. Until imon_bprm_free_security is called, we simply cannot know; therefore, to be on the safe side, we modify the value of arg_0→security to point to the path of the current file, as obtained from the database record's filepath member.

The last thing that we do is test the file to see if its metrics are correct. If they are not, then execution is denied; otherwise, execution is allowed.

imon_file_alloc_security

This function is passed a pointer-to-file as an argument, arg_0. If the the name associated with the file arg_0 lies within the directory reserved for IMon's files, the function returns successfully; none of the files within this directory are accessible from userspace, and the directory itself is not writable. Otherwise, the imon_file_alloc_security requests a record for the file, and places the result in arg_0→f_security. By the end of this function, arg_0→f_security holds either a record or NULL.

Note that, unlike imon_bprm_alloc_security above, this function always returns successfully. This is because not having a database record for a given file is not critical, whereas not having a database record for a file that is being executed is critical. For example, even if `/tmp/myfile` has no record associated with it, we don't mind if it is read – unless the process that is doing the reading happens to be an interpreter, in which case the matter is dealt with in the test function described in 4.5.8.

imon_file_free_security

This function is passed a pointer-to-file as an argument, arg_0. If $\mathsf{arg}_0 \rightarrow$f_security is not NULL, the entry is released back to the database.

imon_file_mmap, imon_file_mprotect

imon_file_mmap is passed four arguments, one of which is a pointer-to-file which we shall call arg_0 and another of which is a bitmask, arg_1, that indicates the protection to be applied to this memory region. imon_file_mprotect is passed three arguments, one of which is a pointer-to-vm_area_struct which we shall call arg'_0 and another of which is a bitmask that indicates the protection to be applied to this memory region. One may obtain the file (if any) that is being mapped by arg'_0 by looking at the $\mathsf{arg}'_0 \rightarrow$vm_file member.

The workings of both functions are almost identical (see Figure 4.8). For this reason, a separate function has been created to handle the common parts which deal with the setting of FLG_MMAPW and FLG_CACHED; this function is described following this subsection.

If arg_0 – or $\mathsf{arg}'_0 \rightarrow$vm_file, in the case of imon_file_mprotect – is not testable, then we allow the operation: the memory region is not backed by any file that we are able to test. We call _mmap_common (described next) to do the bulk of the work necessary. Afterwards, imon_file_mprotect returns successfully; imon_file_mmap tests the file against the database metrics that reflect what its state should be and returns a value that allows or denies access based on this.

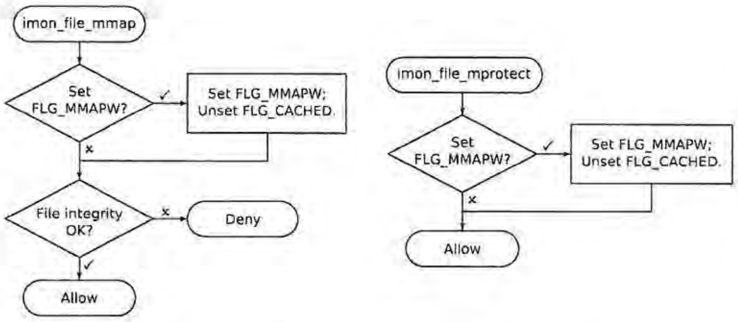


Figure 4.8: imon_file_mmap and imon_file_mprotect

_mmap_common

This function populates (if necessary) the f_security member of the pointer-to-file that it is passed, and tests to see whether the pointer-to-file and protection bitmask passed should result in FLG_MMAPW being set. It is called from both imon_file_mmap and imon_file_mprotect. If the bitmask indicates the memory region is to be writable, and the file passed has a record associated with it, and the file content matters (as determined by whether IGN_HASH is set or not), and writing to the file is allowed by its permissions – then FLG_MMAPW is set, and FLG_CACHED is unset.

imon_file_permission

This function is passed two arguments: a pointer-to-file (arg_0) and a bitmask (arg_1). The bitmask contains the value(s) MAY_READ or MAY_WRITE, indicating the question being asked of the security module. imon_file_permission is the most complex of the hook functions and consists mostly of a series of simple checks, as shown in Figure 4.9. This figure should be examined closely in conjunction with a reading of the text as it resolves any ambiguities that may exist within the textual description.

We first test to see if arg_0 is a regular file. If not, and it is a block device that could hold filesystems that is being written to, we remove FLG_CACHED on all files as a security precaution, thereby nullifying all cached results. Likewise, if it is a character device that could affect writable memory maps, we also unset FLG_CACHED for all files. As a self-protection measure, writing to kernel memory using the kmem character device is not allowed; if we detect that this is the character device being written to, we deny the operation. If arg_0 is any special file other the kmem device, we allow the operation; otherwise, we are sure that it is a regular file,

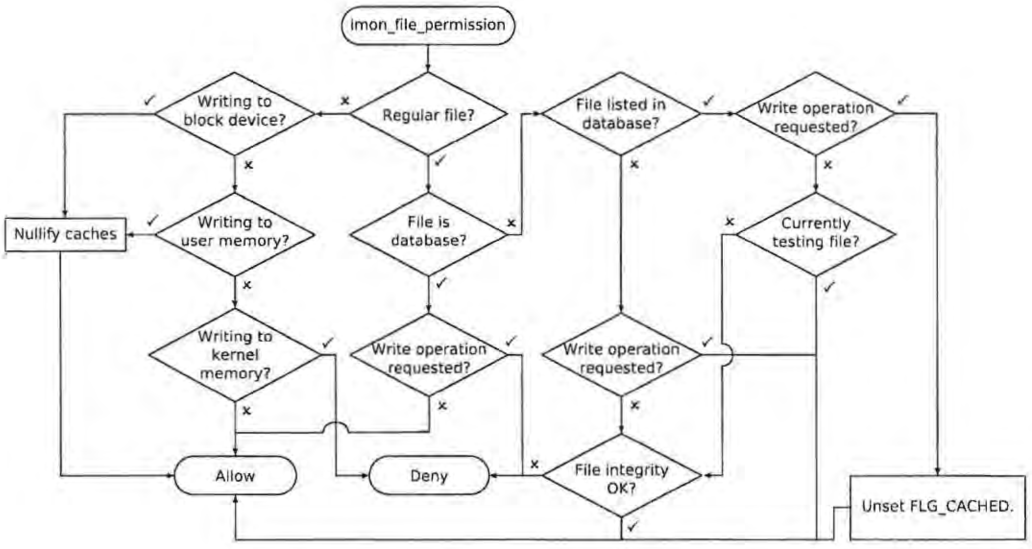


Figure 4.9: imon_file_permission

and we continue with the function logic.

We next check whether arg_0 is the database file. If so, and we are attempting to write to it, the operation is denied; if we are requesting read permission for the database file the operation is allowed.

If $arg_0 \rightarrow$f_security is not NULL, then we know that it is listed in the database. If a write operation is being requested, then we unset FLG_CACHED and allow the operation. This has the effect of forcing the next test of the file to revalidate all its metrics.

If FLG_TESTING is set, we allow the operation: the file is currently being tested, and we allow permission to read the file so that it can be tested. Failure to do this results in a subtle infinite loop:

1. The file is read during a test (for example, to calculate a hash of file contents).
2. Reading the file causes imon_file_permission to be called.
3. imon_file_permission requests that file metrics be tested.
4. GOTO (1).

If the operation requested is not a read, we allow the operation. If no record is associated with the file as yet, we try to associate a record with the file and proceed to test it, returning a status of allowed or denied depending on the result of the test.

imon_task_alloc_security, imon_task_free_security

These functions take an argument, arg_0, of type task_struct. imon_task_alloc_security allocates an imon_procsec structure (see 4.5.5) to $arg_0 \rightarrow$security, and imon_task_free_security releases all the memory associated with $arg_0 \rightarrow$security.

imon_capable

An LSM hook that is used equally for self-protection and during integrity checking is imon_capable, which is called to determine whether a process has the capability to perform a given operation.

Under Linux, the *root* or super-user account can do many things that ordinary users simply cannot do. One of these things is to bypass the usual permissions checks when reading from, writing to or executing a file, so that the super-user is always able to write to any file; this problem is described in more detail in 4.5.4. Whilst this is not a problem for IMon to handle, it does impose a performance penalty to not be able to cache any file that is opened by a super-user process – as must be done to ensure that a malicious process with root privileges is unable to modify any file, no matter what the permissions on the file might be. This performance penalty can be removed by reducing the privileges of the super-user account slightly. Specifically, the CAP_DAC_OVERRIDE capability (which allows the aforementioned bypassing of checks) is disallowed.

Module loading and unloading requires a process to have the CAP_SYS_MODULE capability. For self-protection purposes, only certain executables are allowed to load or remove modules; for these executables, the FLG_MODCAP flag must be set in the IMon configuration file. Given this restriction, an attacker is unable to use a flaw in a program executed with super-user privileges to execute code that could remove IMon from memory; in addition, if only module-loading utilities are given the FLG_MODCAP flag, removing IMon or otherwise corrupting or disabling its activities once it is loaded becomes much more difficult to do. As a secondary benefit, the same security benefit is extended to any other loaded modules. If said module-loading utility is given the policy of deny_others and a selection of allowed modules, it also becomes a non-trivial task for an attacker to load *any* untrusted code into the kernel!

The imon_capable function is passed two arguments: arg_0, which is a pointer-to-task_struct indicating which process is requesting the capability, and arg_1, which is an integer representing the capability asked. If arg_1 is CAP_SYS_MODULE, a record is associated with arg_0 and the operation is denied unless FLG_MODCAP is set in arg_0→mm→mmap→vm_file→f_security→flags.

If arg_1 is CAP_DAC_OVERRIDE, permission is denied; otherwise, permission is granted or denied according to the capability set accorded to arg_0.

4.5.7 Utility Functions

A number of utility functions have been created to make the hooks mentioned easier to implement. Since these are not fundamental to the workings of IMon, only a brief description of how each is implemented is given here.

fullpath Passed a pointer-to-file, this function traces up the directory tree, storing names as it goes, to obtain the full path of a file.

fullpath_dentry This performs the same functions as fullpath, but is passed a pointer-to-dentry instead.

get_task_security Given arg_0 as a pointer-to-task (which is a structure used to represent an executing process), this function requests a record for the file used to create the process, and places the record in the arg_0→mm→mmap→vm_file→ f_security field, if this field is not already filled in. get_task_security also allocates a per-process data structure to arg_0→security, if one does not already exist.

isTestable This function tests to see whether the pointer-to-file passed refers to a file that can be tested. It is possible that the file passed is a ***swapfile***, or a directory, or a block device, or a symbolic link, or exists on a virtual filesystem, and so forth; this function returns non-zero if the file is none of these, and is therefore testable. Optionally, isTestable also fills in a parameter that indicates the type of the file.

fgets This takes similar parameters to the Standard C Library function fgets, and performs a similar function. The kernel only provides basic file-reading functions, and fgets was built on top of these.

4.5.8 Testing

Testing of a file is done by two functions, one of which applies policy and the other which tests the metrics of a single file. This split of functionality makes adding new policies and new metrics to be tested extremely easy.

test

The test function takes as a parameter a single pointer-to-file, which we can call arg_0, and applies a policy to it. The function makes two assumptions:

1. A best-effort has been made before test was called to associate a record with arg_0→f_security, and
2. arg_0 is a regular, testable file.

It first obtains a filename for arg_0, placing it in the variable filename. Then a record, sec, is associated with the currently-executing process; an error is returned if the executable that started the process has no record associated with it, since no record being associated with it means that the executable is unknown to the system.

Next, which executable to test is determined. In the case of a native binary executable, sec is the correct file to use for policy purposes. However, in the case of an interpreted file, it is the script itself which should be used to determine policy and not the script interpreter. We create a pointer-to-imon_security called testfile and use it to point to the correct file: if current→security→script is not NULL then we assign the script's record to testfile; otherwise, we assign sec to testfile.

If the policy testfile→full→policy is currently unset, we perform runtime policy assignment to set testfile→full→policy as follows:

- If testfile is a script, we set the policy to DEFAULT_INTERPRETER (which is currently an alias for DENY_NOTFOUND).
- If testfile is a native binary executable, we set the policy to DEFAULT_BINARY (which is currently an alias for ACCEPT_ANYTIME).

Policy is now applied based on the value of testfile→full→policy:

deny_others For each dependency of testfile, we compare filename to the dependency's filepath member. If a match occurs, then we return either success or failure depending on the result of testing the integrity of arg_0. If no match is found, we return an error.

deny_notfound We return either success or failure depending on the result of testing the integrity of arg_0.

accept_after We first create a pointer-to-imon_procsec, proc, to refer to the currently-executing per-process security field; this is done for ease of reference. If proc→deps_checked is NULL, we allocate an array that is large enough to store bits for each dependency (see 4.5.5). We test to see if all dependencies have been checked by counting the bits set in proc→deps_checked; if all dependencies have been checked, then we either return a result based on testing arg_0, or (if arg_0 has no associated record) return successfully.

If all dependencies have *not* been checked, then for each dependency of testfile, we compare filename to the dependency's filepath member. If a match occurs, then we test arg_0 and set the correct bit in proc→deps_checked if the test is successful; we return either success or failure depending on the result of the integrity check. If no match occurs, we return an error.

dependency, accept_anytime If arg_0 has an associated record, we return either success or failure depending on the result of testing the integrity of arg_0. Otherwise, we simply return success.

Note that in the application of the above policies, a file is always implicitly assumed to be a dependency of itself. This means that, for example, if file `/bin/test` is set to have the policy deny_others and has no dependencies, but wishes to read from `/bin/test`, the operation is allowed.

test_file

This function takes two arguments: arg_0, which is a pointer-to-file, and arg_1, which is a boolean variable that determines whether a check of the file hash should be forced. test_file is usually not called from an LSM hook, but via test instead; this ensures that policy of the currently-executing executable is always checked before a file's metrics are tested, and that a file's metrics are never needlessly tested. The exception to this is when test_file is called from bprm_check_security, as the only important factor during the execution of a file is that the file is a valid executable; it is for this reason that when test_file is called from bprm_check_security, arg_1 is set.

test_file first tests to see whether $arg_0 \rightarrow$f_security is NULL; if so, it returns failure. This is because a nonexistent file does not pass any integrity checks. Next, it tests to see whether the FLG_CACHED bit of the associated record is set and arg_1 is unset, and returns successfully if it is. This is because the results of the previous integrity check were successful and are still valid. FLG_TESTING is set (see 4.5.6 for the details of why this is necessary). From this point on, no matter what the outcome of the test, FLG_TESTING is unset before the function returns; for clarity, this is not stated explicitly in what follows. The FLG_CACHED flag is also unset, pending the result of the tests.

For each attribute, if the corresponding ignore-flag is set or the actual attribute matches the ideal attribute, a pass is recorded. If an actual attribute fails a check, the function immediately returns an error without proceeding to the next test. "Cheap" tests that only involve looking up values within internal kernel data structures are done first, with the more time-consuming calculation of a hash of file contents being done last. The hash check is always done, even if the IGN_HASH flag is set for arg_0's associated record, if current$\rightarrow$security$\rightarrow$script indicates that it is an interpreter or if arg_1 is set.

At the end of the tests, if arg_0 is not on a network-mounted filesystem (as tested by checking whether it has a block device associated with it), and if FLG_MMAPW is not set for it, then FLG_CACHED is set. The function finally returns successfully.

4.5.9 Taking action

If a file failure is detected, some action must be taken. The exact action to be taken should be easy to change, and for this purpose the _action function has been created. Its function prototype is effectively

int _action(**const char*** arg_0, **const int** arg_1, **const struct** file *arg_2, **const char** *arg_3);

_action is usually called using a line such as

returnValue = action(test(f), f, "Lack of turnips");

The first thing to notice about this line is that action (note the lack of a leading underscore!) is a macro, defined as

```
#DEFINE ACTION(X,Y,Z) _ACTION(__FUNC__,X,Y,Z)
```

It adds the function name of where it is being called from as the arg_0 of _action. arg_1 (test(f) in the example) provides a return code, and arg_2 provides a pointer-to-file (if any) that indicates the file that has played a primary role in the failure. The last parameter, arg_3, indicates extra information and is useful for mentioning why the failure has occurred.

The _action function is in this fashion provided with enough information to provide useful log messages and take appropriate action. Currently, if arg_1 is zero, the function returns zero. However, if it is non-zero, a message is sent to the system log and arg_1 is returned. Note that since the return code of _action typically determines the return value of IMon, it is quite possible to handle file failure by doing nothing at all – as might be done on a honeypot system, for example.

```
Oct 12 11:31:04 [kernel] [imon] [Warning] test_file: No database entry
for
'/src/read_write.c' exists.

Oct 12 11:31:05 [kernel] [imon] [Warning] test: '/usr/bin/perl5.8.7'
is
interpreter. Denying access to '/src/read_write.c'.

Oct 12 11:31:05 [kernel] [imon] [Action] _action: Failure occurred in
'imon_file_permission', task '/usr/bin/perl5.8.7' and file
'/src/read_write.c'; extra info: File has failed an integrity check.
See previous message.
```

Code Snippet 4.8: Example: log file excerpts

Both test and test_file functions output complementary information to the system log, which leads to very informative log messages that indicate which file failed a check, why the check was failed, which process was running at the time, and so forth. Example log messages are shown in Code Snippet 4.8. From this Code Snippet we can clearly see the chain of events that has led to _action being called.

4.5.10 Transitive Interpretation

This section gives a broad picture of how transitive interpretation works, and follows a single example through from beginning to end using a diagram and trace table to ease understanding. Transitive interpretation relies on small code sections implemented in various parts of IMon. This section summarizes how and why it works in a step-by-step form. It may be regarded as a more expanded and explained version of the steps found in 4.5.2 and is suitable for understanding transitive interpretation in totality.

Through this entire section we shall use the example of a file, `/tmp/ccc`, that has the string `#!/bin/aaa /bin/bbb` at the top of it. This string causes `/bin/aaa` to be executed as the interpreter and passed the filename `/tmp/ccc` as an argument; `/bin/aaa` subsequently executes `/bin/bbb` and passes it `/tmp/ccc`, from userspace, as an argument. Thus, the kernel does see `/bin/bbb` as the interpreter, and treats the case as though `/bin/bbb` were invoked from the commandline as `/bin/bbb /tmp/ccc`. Transitive interpretation allows the kernel to understand that `/bin/bbb` is the actual interpreter and that it has *not* been invoked standalone.

Figure 4.10 shows the logic underlying transitive interpretation. Grey blocks indicate those instructions that are carried out within the ambit of IMon, as opposed to in userspace or in the rest of the kernel. Words in square brackets indicate that a particular variable has been allocated at that step and will be called by the name in brackets for the rest of the explanation. For example, "allocate bprm→file→security [record]" means that "record" will be used as another name for "bprm→file→security", which has been allocated in this step. Since transitive interpretation relies to a great degree on state changes, Table 4.1 has been provided to trace through steps that alter significant variables: once again, it is based on the example mentioned previously.

After going through each step, we should be able to understand how transitive interpretation works for this example, and should also be able to use Figure 4.10 to generalize from this example to any others that might come to mind.

0. This represents the userspace command to fork-and-execute `/tmp/ccc`, the interpreted file. We assume that this is done by some shell process, named `sh` in Table 4.1.

1. This step shows part of the initialization of the new process as imon_task_alloc_security is called, and by the end of this step the process creation is complete.

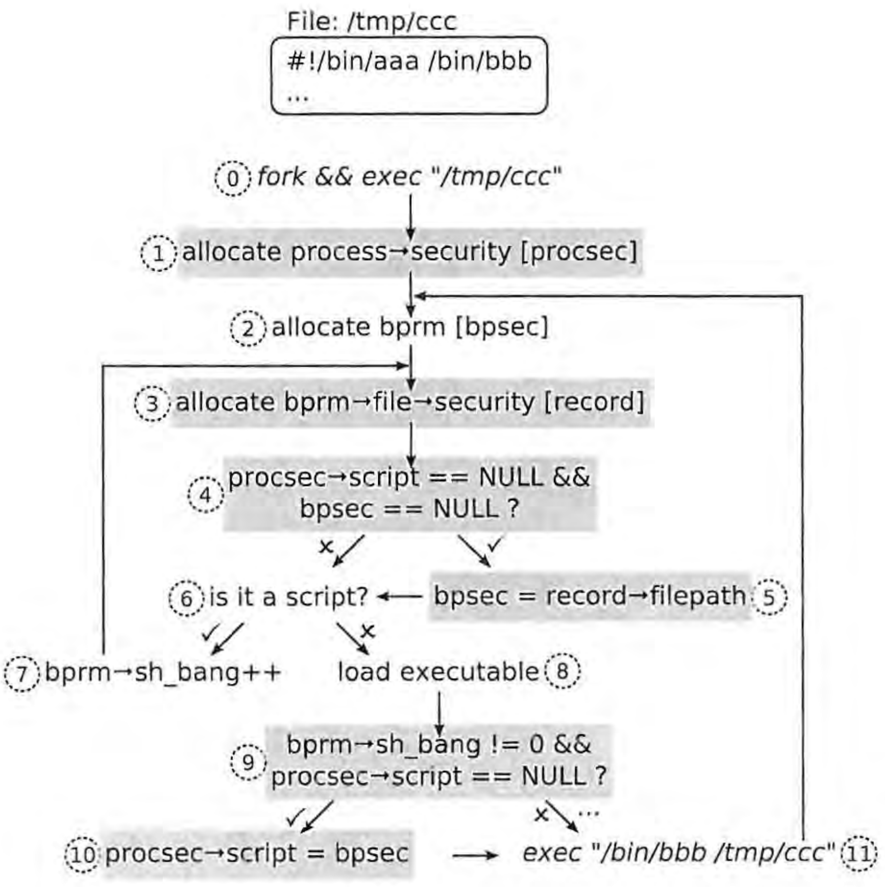

File: /tmp/ccc
#!/bin/aaa /bin/bbb
...
0 fork && exec "/tmp/ccc"
1 allocate process→security [procsec]
2 allocate bprm [bpsec]
3 allocate bprm→file→security [record]
4 procsec→script == NULL && bpsec == NULL ?
5 bpsec = record→filepath
6 is it a script?
7 bprm→sh_bang++
8 load executable
9 bprm→sh_bang != 0 && procsec→script == NULL ?
10 procsec→script = bpsec
11 exec "/bin/bbb /tmp/ccc"

Figure 4.10: Transitive interpretation

imon_task_alloc_security creates and initializes the process's security field, aliased to procsec in this explanation. All members of procsec are zeroed.

2. This represents the creation of the bprm structure that will be used during the execution; we shall call the security part of this structure bpsec. bpsec is initialized to NULL.

3. In imon_bprm_alloc_security, we allocate the imon_security structure that gives the correct metrics of bprm, and alias it to record.

4. When both of these conditions are true, then this is the very beginning of what might be a chain of transitive interpretation. Referring to Table 4.1, we see that procsec→script will be NULL until (8), and that bpsec is NULL once we reach (2) again; therefore, neither of these conditions alone can establish that this is the beginning of a possible chain.

5. If we have established that this is the very first file in a possible execution

Step	procsec	bprm		record	process name
	script	bpsec	sh_bang	filepath	
0	-	-	-	-	sh
1	NULL	-	-	-	sh
2	NULL	NULL	0	-	sh
3	NULL	NULL	0	/tmp/ccc	sh
5	NULL	/tmp/ccc	0	/tmp/ccc	sh
7	NULL	/tmp/ccc	1	/tmp/ccc	sh
3	NULL	/tmp/ccc	1	/tmp/aaa	sh
8	NULL	/tmp/ccc	1	/tmp/aaa	aaa
10	/tmp/ccc	/tmp/ccc	1	-	aaa
11	/tmp/ccc	-	-	-	aaa
2	/tmp/ccc	NULL	0	-	aaa
3	/tmp/ccc	NULL	0	/tmp/bbb	aaa
8	/tmp/ccc	NULL	0	/tmp/bbb	bbb

Table 4.1: Partial trace: transitive interpretation

chain, we set bpsec to point to record→filepath; from this piece of data we can resurrect the entire record at will by requesting it from the database.

6. The kernel decides whether the executable is a script.
7. If so, then bprm→sh_bang is incremented and the same bprm structure has certain members filled with new values; however, bpsec is *not* altered at all. After this, we end up at (3) again.
8. If not, then an executable is loaded.
9. In this step, which occurs within bprm_free_security, we can finally test that this is an interpreter by examining bprm→sh_bang; we can also be certain that we have reached the end of an execution, and test that is the original script file in the interpreted chain (since procsec→script is NULL).
10. We assign bpsec to procsec→script, recalling that bpsec was set (5).
11. This step shows the return to userspace; the userspace program may opt to execute another interpreter, and this takes us back to (2). Table 4.1 traces through this second visit to (2).

4.6 Summary

In this section we have described in detail the implementation of IMon on Linux. We have given an overview of the system, illustrated the process used to create a baseline, touched upon cryptography, spent some time understanding the workings of the IMon database and delved deeply into the inner workings of the IMon file integrity checking code. Implementation problems have at every stage been explained and solved, with possible solutions being explored where necessary. The design that was conceived in chapter 3 has been faithfully followed throughout.

At present we can safely claim that IMon is the most complete and comprehensive realtime integrity checker for Linux. It protects itself from tampering, tests both scripts and native executables and is the very first integrity checker to accomplish this on Linux, uses multiple metrics, and is the only realtime integrity checker we know of that, via policy, understands files not only as individual entities but also as dependencies of each other.

The reader now has an understanding of the IMon codebase, and is prepared to understand what occurs "behind the scenes" during each test. In the following chapter we evaluate the implementation described in this chapter.

Chapter 5

Evaluation

At this point we have seen the prior work in the field of file integrity checking, created a best-of-breed design, described a concrete and usable implementation, and can now proceed to evaluate such an implementation. This chapter evaluates IMon in terms of security provided, performance impact, and applicability.

5.1 Security

In this section we examine the security that IMon provides to a system, and expand on how each security feature has been tested to ensure that it works correctly. Note that due to the fact that shared executables are linked to system libraries such as `/lib/libc.so`, there are more dependencies added for certain executables than have been mentioned below. To make explanations simpler, system dependencies have not been listed.

5.1.1 Binaries

IMon stops binary files from being executed via the execve system call unless those binary files are listed in the IMon database. We have tested this by creating both statically-linked executables (`a.static` and `b.static`) and dynamically-linked binary executables (`a.dynamic` and `b.dynamic`), and placing the `a.*` executables in the database whilst leaving the `b.*` executables out. As expected, `a.*` could be executed once IMon was running, and `b.*` could not.

5.1.2 Shared Libraries

Depending on the policy set for a binary file, IMon may or may not allow access to a shared library requested by the executing binary.

Policy	Listed	Unlisted	Conditions
accept_anytime	✓	✓	
accept_after	✓	✓	If listed as dependency, or if all dependencies have been tested
deny_others	✓	✗	If listed as dependency
deny_notfound	✓	✗	

Table 5.1: Policy-based shared-library loading

Table 5.1 shows whether access would be granted to a shared library for each policy, and the conditions that must be met before access is granted. Using this table, we can determine whether a particular shared library would be allowed to load given the policy associated with the file that wishes to load it. For example, if we have a file `/bin/test` that wants to load `/lib/library.so`, and the policy **deny_notfound** is associated with `/bin/test`, then access to `/lib/library.so` would only be allowed once it has been found in the database (and, hence, tested). However, if `/bin/test` had the policy **accept_anytime** associated with it, access to `/lib/library.so` would be granted whether it has been found in the database and tested or not.

Testing

To test whether the security of policy-setting works, we created four executable files: $\mathtt{e}_{\mathtt{deny\_others}}$, $\mathtt{e}_{\mathtt{deny\_notfound}}$, $\mathtt{e}_{\mathtt{accept\_after}}$ and $\mathtt{e}_{\mathtt{accept\_anytime}}$. These were assigned appropriate policies. We also created three libraries:

- $\mathtt{l}_{\mathtt{dep}}$, a dependency of all four executables.
- $\mathtt{l}_{\mathtt{non\_dep}}$, listed in the database but a dependency of none of the executables.
- $\mathtt{l}_{\mathtt{unknown}}$, unlisted and therefore unknown to IMon.

We then ran an exhaustive series of tests during which we linked each library to each executable such that each executable was at some point linked to every possible combination of libraries. The results of these tests bear out the expectations of Table

5.1 exactly. One caveat to be aware of is that the order in which libraries are loaded can be crucial: for example, if $\texttt{e}_{\texttt{accept\_after}}$ attempts to load $\texttt{l}_{\texttt{non\_dep}}$ or $\texttt{l}_{\texttt{unknown}}$ before $\texttt{l}_{\texttt{dep}}$, it will abort execution before ever loading $\texttt{l}_{\texttt{dep}}$; however, if $\texttt{l}_{\texttt{dep}}$ is loaded first, the other two libraries will be loaded successfully as well. Using dlopen (see 5.1.5) instead of linking directly to a library avoids this pitfall.

5.1.3 Scripts

IMon stops interpreted files from being executed unless those files are listed in the IMon database; in addition, interpreters are not allowed to read unlisted files, and the standalone execution of an interpreter is forbidden. We test that this is so by running the tests listed in 5.2. As this table shows, the expected result was obtained in each case.

Test Description	Result	Expected
Executing the interpreter `/usr/bin/python` standalone via the system Standard C function.	[failure]	[failure]
Executing the interpreter `/usr/bin/python` standalone via tha execve function.	[failure]	[failure]
Executing a test-script listed in the database.	[success]	[success]
Executing a test-script not listed in the database.	[failure]	[failure]
Reading an unlisted file via a listed script.	[failure]	[failure]
Executing a listed test-script which is subject to a two-stage chain of transitive interpretation.	[success]	[success]
Executing a listed test-script which is subject to a three-stage chain of transitive interpretation.	[success]	[success]
Executing an unlisted test-script which is subject to a three-stage chain of transitive interpretation.	[failure]	[failure]
Executing the command-line: `python listed_script.py`.	[failure]	[failure]
Executing the command-line: `python unlisted_script.py`.	[failure]	[failure]
Executing a pipe-line involving a standalone interpreter as one of the stages.	[failure]	[failure]
Executing the `source` command from a tcsh script to read a file that is not in the database.	[failure]	[failure]

Table 5.2: Script security

5.1.4 Raw Device Writes

Writing to a raw device can allow an attacker to modify the content of files on a system without altering the metadata associated with those files, since the modification is carried out at a level lower than that of the filesystem. This is especially dangerous in the case of a caching file integrity checker since a cached file will not have its content revalidated unless a write to the file is detected at the level of the filesystem – and writing directly to a raw device allows an attacker to bypass this detection.

IMon uncaches all files once a write to a block device that could affect a filesystem is done. We test for this by modifying IMon slightly to print a message to the system log just after beginning a test of a file. This message is usually only printed once since the cached result is used subsequent to the first test of the file; however, once a write is made to a block device and the file is read again, the message is printed once more. This indicates that the file is no longer cached, and another test is being done.

5.1.5 Dynamic Linking Loader

The function dlopen allows a program to open a shared binary object at runtime and obtain a function pointer that can then be used to execute code from the shared object. Semantically, this is akin to loading functionality from a shared library at runtime, and the same rules that apply to loading code from shared libraries should apply to the use of dlopen. Given this similarity, we tested dlopen using the same methodology used in 5.1.2, and arrived at the same results noted in that section.

5.1.6 Memory-maps

Memory-mapping a file is an alternative way of gaining read and/or write access to it. Importantly, this method bypasses some of the LSM hooks that would ordinarily be called for a read or write operation, such as the file_permission hook which determines whether a request to read or write to a file should be granted (see 4.5.4). A conceivable attack that uses memory-mapping to bypass a caching integrity checker is:

1. The attacker memory-maps `file` writably. This causes an integrity checker to

uncache any entry for `file`.

2. The attacker now reads `file`. This causes the integrity checker to cache the file.
3. The attacker modifies the memory-mapped file, and then unmaps it, causing the changes to be written to disk. This is not detected by the integrity checker.
4. A user opens `file`. The integrity checker uses the cached (and now invalid!) result of the previous check, and allows access to `file`.

In (4), the correct action for the integrity checker to take is to test the file again and deny access to it if necessary. To test IMon's handling of memory-mapping, we have implemented the above attack and found that, once IMon is running, the attack is no longer possible. We have also tested the above style of attack as it may be applied to code execution:

1. The attacker memory-maps `library` writably. This causes an integrity checker to uncache any entry for `library`.
2. The attacker uses dlopen to execute a function from `library`. This causes the integrity checker to cache the file.
3. The attacker modifies the memory-mapped file, and then unmaps it, causing the changes to be written to disk. This is not detected by the integrity checker.
4. A user accesses a function from `library`. The integrity checker uses the cached (and now invalid!) result of the previous check, and a modified version of the function is executed.

Once again, IMon detects this attack and stops it from occurring.

5.1.7 Deletion, Truncation, and Replacement

The simplest attack that can be carried out on a caching integrity-checking system is as follows:

1. The attacker reads `file`. This causes the file to be cached.

2. The attacker now removes `file` and replaces it with a custom executable. Alternately, he may truncate file, or copy another file in its place.

3. A user accesses `file`. This access is allowed since the file is cached.

The correct action to take is to uncache the file at (2), so that it is tested once again at (3). IMon does this, and testing a number of simple attacks of this sort have all resulted in the attacks being defeated.

5.1.8 Policy

Policy aids an administrator in setting up programs that may only ever access certain files, or may only access files after a configuration has been read, and so forth. By setting up the minimal file access permissions that a program needs, an administrator can increase security by "locking down" a given application to using only certain files and denying access to all other files.

To test whether the created policies allow and deny access as intended, we created four executable files ($\mathtt{e}_{\mathtt{deny\_others}}$, $\mathtt{e}_{\mathtt{accept\_after}}$, $\mathtt{e}_{\mathtt{accept\_anytime}}$ and $\mathtt{e}_{\mathtt{deny\_notfound}}$) and assigned each of them a certain policy. Each executable `e` is identical, and merely opens files passed on the command-line in sequence and attempts to read each one; for example, running the command $\mathtt{e}_{\mathtt{accept\_anytime}}$ `aye bee see` would cause $\mathtt{e}_{\mathtt{accept\_anytime}}$ to attempt to open and read first `aye`, then `bee`, and lastly `see`.

We then created three files:

- $\mathtt{f}_{\mathtt{dep}}$, a dependency of all the `e` executables.
- $\mathtt{f}_{\mathtt{non\_dep}}$, a file that is listed in the database but is not a dependency of any of the `e` executables.
- $\mathtt{f}_{\mathtt{unknown}}$, a file that is not listed in the database and is therefore unknown to IMon.

The following command-lines were then executed, with expected results being given after each command-line:

- $\mathtt{e}_{\mathtt{deny\_others}}$ $\mathtt{f}_{\mathtt{non\_dep}}$ $\mathtt{f}_{\mathtt{dep}}$ $\mathtt{f}_{\mathtt{unknown}}$

  Expected: only $\mathtt{f}_{\mathtt{dep}}$ should be readable.

- $\mathtt{e}_{\mathtt{deny\_notfound}}$ $\mathtt{f}_{\mathtt{non\_dep}}$ $\mathtt{f}_{\mathtt{dep}}$ $\mathtt{f}_{\mathtt{unknown}}$

  Expected: $\mathtt{f}_{\mathtt{dep}}$ and $\mathtt{f}_{\mathtt{non\_dep}}$ should be readable.

- $\mathtt{e}_{\mathtt{accept\_anytime}}$ $\mathtt{f}_{\mathtt{non\_dep}}$ $\mathtt{f}_{\mathtt{dep}}$ $\mathtt{f}_{\mathtt{unknown}}$

  Expected: all files should be readable.

- $\mathtt{e}_{\mathtt{accept\_after}}$ $\mathtt{f}_{\mathtt{non\_dep}}$ $\mathtt{f}_{\mathtt{unknown}}$ $\mathtt{f}_{\mathtt{dep}}$ $\mathtt{f}_{\mathtt{non\_dep}}$ $\mathtt{f}_{\mathtt{unknown}}$

  Expected: $\mathtt{f}_{\mathtt{non\_dep}}$ and $\mathtt{f}_{\mathtt{unknown}}$ shold not be readable initially. After $\mathtt{f}_{\mathtt{dep}}$ is read successfully, they should be readable.

In each case, the expected results were obtained. We then ran the same tests using interpreted files instead of binary executables, and obtained exactly the same results.

5.1.9 Modules

Modules loading and unloading can be restricted to a known-good set of modules, and only certain executables can be allowed to load or unload modules. To test this, we made copies of the `rmmod` and `insmod` executables which are used to remove or insert a module respectively. We then changed the database entry for the real `rmmod` and `insmod` to allow them to load and unload modules, and placed both of them under the **deny_other** policy after providing suitable modules as dependencies.

As expected, running $\mathtt{rmmod}_{\mathtt{copy}}$ and $\mathtt{insmod}_{\mathtt{copy}}$ resulted in no change to the modules loaded; in addition, attempting to insert a module that was not on the dependency-list of allowed modules failed. This demonstrates that the functionality to secure module loading and unloading is effective.

5.1.10 Corrupted Files

IMon stops corrupted files from being accessed, providing that metrics that would indicate corruption are not specifically ignored. To test this, we modified files used in previous tests that *could* be accessed, and noted the results. As expected, IMon denied access to such files no matter what policy was attached to them.

5.1.11 Attack Vectors

As with any complex system, IMon may be susceptible to attacks that we have not considered as yet. In this section we consider two possible attack vectors.

Ignoring attributes

In 3.5.1 we discuss the necessity of ignoring changes to certain attributes. The most dangerous attribute to ignore, in our estimation, is the cryptographic hash of the file: ignoring it opens the way for an attacker to make undetected changes. IMon attempts to ensure that this does not become a problem by forcing the hash of any file being executed to be examined before it is executed, no matter what the per-file flags of the file may be. We are, however, uncertain as to whether every possible avenue that could exploit attribute ignoring has been found.

For this reason, we recommend that attributes never be ignored unless it is absolutely necessary to do so. It is worth emphasising that, as recommended in 3.5.1, attributes should not be ignored in order to increase the speed of the system.

Caching

Testing whether the implementation of caching may introduce security holes is a difficult proposition since all possible combinations of operations that may cause a file to be cached and uncached are difficult to work out. In the implementation of IMon we have chosen to set the cache-flag in only one place: after a test has been successfully completed. We have also ensured that it can be overridden within the test_file function, should the caller so desire; and we have uncached a file whenever there might be a possibility that a metric could be changed. Despite these precautions, some combination of operations that leaves a file erroneously cached may exist, and we therefore consider caching to leave open a possible attack vector.

For performance reasons caching has been left in the IMon system. Should it prove to be a security risk that cannot be surmounted, it can be removed entirely from the system by commenting out a single line of the source code.

Database size	Average number of comparisons
10	3
100	6
1,000	9
100,000	16
500,000	18
1,000,000	19
10,000,000	23
10,000,000,000	33

Table 5.3: Database size vs. average comparisons

5.2 Performance

As mentioned in 1.3.2, a file integrity checker must be efficient, and a realtime file integrity checker must be very efficient. In this section we discuss the performance impact of IMon on a running system.

5.2.1 Scalability

IMon has been tested with a database of over 280,000 files, with no perceptible slowdown in either lookup speed or verification. Since lookup is an $O(\log_2 n)$ operation (see 3.10.2), we can see the impact on the lookup algorithm of increasing the database size by examining Table 5.3.

Assuming that there are 500,000 files in the database[1], it would take only 18 comparisons to find the correct record. As can be seen from Table 5.3, the lookup of records is not a scalability issue, nor is it likely to become one: even with ten billion files in the database, only 33 comparisons (on average) are required to find the correct one. Note that both Table 5.3 and the statements made in this paragraph are based on the work of Knuth in [31, pp. 409–414].

A more pressing scalability concern is the amount of memory used by the IMon database: at ≈ 63 bytes used per entry (including memory used by filepath members), a database with 500,000 files would take up $500000 \times 63 = 31500000\text{bytes} \approx 30\text{MiB}$ of memory whilst IMon is running. Due to the on-demand loading process, this memory will not increase overmuch whilst IMon is active; however, it is still a significant amount of memory to be taken up. Modern systems, fortunately, have

[1]We estimate that there are $\approx$320,000 files, including non-executable files, on an average Linux desktop machine

more than enough physical memory to make the impact of IMon negligible and this is therefore not as great a scalability concern as it might once have been. Nevertheless, we discuss ways in which the memory impact of IMon could be reduced in 6.2.1, and leave it as future work.

5.2.2 Efficiency

The efficiency of IMon can be discussed either in terms of the amount of overhead that IMon adds to the execution of the Linux kernel, or in terms of the performance impact experienced by a user whilst IMon is running. We have chosen to evaluate the efficiency of IMon in both these spheres.

Kernel Overhead

To test the overhead of IMon on a running kernel, we added simplistic benchmarking variables and tested the duration for which each non-trivial function ran. We also tracked how many times each function was called, and how long the entire execution of IMon took from initialization to shutdown. All durations were measured using the kernel-provided jiffies variable which is incremented by a timer interrupt every millisecond. Therefore, our measurements have a granularity of one millisecond. By measuring these two items, we were able to calculate the performance impact of IMon in terms of milliseconds/call for each function. Adding together all of the milliseconds/call times gives us a rough idea of how much overhead is imposed by IMon on a running kernel.

We then created a short test program that executed itself a configurable number of times. Bytes were added to the end of the test executable to change its size; this was done to simulate the execution of files of various sizes. Finally, we set the number of processes to be created to be 20,000, and collected benchmark data. Figure 5.1 shows the results of this test.

The jagged nature of the lines shown in 5.1 can be explained through a combination of the limited granularity of the Linux timer and the interruption of IMon processing by other system operations (which, on a preemptive kernel, may interrupt IMon at almost any stage). There is also a possibility that certain measurements were lost due to the fact that no locking mechanism was used to guard against race conditions involving the global benchmarking variables. Even taking these method-

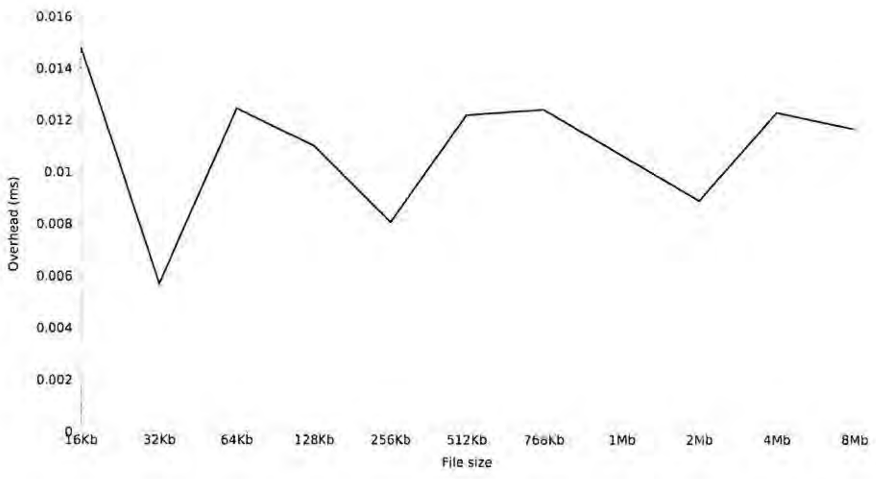


Figure 5.1: Kernel overhead of IMon

ological limitations into account, however, it is clear that the impact of IMon is low enough to be imperceptible – and would still be low enough to be imperceptible if they were three or four times as great.

We also created other performance tests to read in files of different sizes byte-by-byte, and to memory-map and unmap files of different sizes. These tests revealed a performance impact that was less than that shown in 5.1. This is intuitively correct since starting a program tests both file-reading and memory-mapping, and would therefore impose a greater overhead on a running kernel than doing either operation alone.

System Impact

Compilation of a large program results in files being read, files being memory-mapped, programs being executed, data being piped from one process to another, and so on: such a workload is ideal for testing whether IMon's impact on a system is perceptible. We first compiled a freshly-uncompressed 2.6.14 kernel source tree without the IMon module running, and timed the entire process using the `time` command. We then repeated the same procedure with IMon running. Before each test, the test machine was cold-booted to ensure that OS and CPU caches did not unduly skew results. The results can be seen in Table 5.4.

Table 5.4 presents three measurements. The first, "Real time", is the amount of objective time that has passed; that is, the amount of time as could be measured

Metric	With IMon (s)	Without IMon (s)
Real time	486.782	489.320
Userspace time	439.231	441.092
System time	33.378	29.062

Table 5.4: System impact of IMon

from start to end using a wall-clock or stopwatch. "Userspace time" and "System time" refer to the time spent in userspace and kernelspace respectively. As expected, with IMon running the amount of time spent in kernelspace is higher; however, the increased time is almost unnoticeable when compared to the total running time of the entire compilation. Indeed, the real time of the compilation without IMon running is higher – quite possibly due to other processes running on the system at the same time.

Based on the results obtained, we can conclude that the system impact of IMon is negligible.

5.3 Applicability

In this section we discuss where IMon may be most suitably deployed. The section is broken into unsuitable deployments, possibly troublesome deployments, and suitable deployments; at the end of the section, a partial list of security problems that IMon does *not* address is presented. This section acts only as a guide, and should not be mistaken as a comprehensive list of what is or is not a suitable use for IMon.

5.3.1 Unsuitable Deployments

IMon is not suitable for the following systems:

Badly-configured systems A system that has shared libraries whose permissions allow them to be written to is one that may take an enormous performance hit if IMon is used. As described in 4.5.4, such shared libraries cannot be cached if security is to be maintained. Since the functionality within shared libraries is often invoked, being unable to cache the results of testing them could impose a large performance overhead on the system.

Fortunately, this problem is easily solved by modifying shared library permissions to disallow writing to them.

Development systems A system that is used for developing applications is unsuitable for use with IMon. This is because newly-compiled executables will not be able to run unless they have been placed in the database – or unless they are specified in the database with an appropriate combination of per-file flags such as [IGN_HASH, IGN_SIZE, IGN_CTIME, IGN_MTIME, IGN_MODE]. However, this "solution" to the problem is in reality a security hole: an attacker is now able to replace a file with such per-file flags with any other file of his choosing, and is able to execute that file without IMon being aware that anything is wrong.

5.3.2 Troublesome Deployments

Deploying IMon on a laptop could prove troublesome due to the additional hard-disk accesses that it necessitates. If the laptop is to function on battery power for an extended period of time, such activity could increase its power consumption drastically. However, no tests have been conducted to determine how serious the effect of IMon on a laptop system is; therefore, the hypothesis that it is unsuitable for a laptop could prove unfounded. The same arguments that apply to a laptop apply equally well to any other battery-powered device that IMon may be executed on.

5.3.3 Suitable Deployments

IMon is especially suited to the following environments:

Stable-configuration machines Routers, firewalls, web servers and mail servers are examples of machine configurations that are not expected to execute new code very often, and are expected to be secure. IMon is suitable for use in any configuration that is expected to be stable or unchanging, and adds a level of security to the configuration.

Special-purpose machines Laboratory machines used in schools, office machines used for work, and internet cafés are examples of special-purpose machines that are expected to run a limited number of very specific applications, and nothing else. For these machines, IMon is suitable for restricting the number of applications to those allowed by a system administrator, and for decreasing the chance that a strong compromise of the machine will occur either maliciously or accidentally.

Corporate desktops As described in 1.2.3, a medium or large organization could find it beneficial to maintain a single set of executables and libraries, with multiple defined execution profiles. As an additional advantage, organizations that use IMon also derive the benefits of licence compliance (as described in 1.2.3.

Honeypots A honeypot, especially one that is intended to capture and analyze foreign executables, is a well-suited environment for IMon. With some modification, IMon can be made to hide its existence and store foreign executables in a safe location instead of denying their execution outright. This would be a useful ability to have on such a honeypot.

IMon is suitable for more than these, of course; any situation that is not an unsuitable one (see 5.3.1) is a possible situation in which the added protection provided by IMon can be used.

5.3.4 Completeness

It should be noted at this point that IMon is *not* a complete security solution. IMon addresses certain attack vectors, most notably those posed by unauthorized code, and prevents (to the best of its ability) strong intrusions from occurring. If a more complete (but much more complex) security solution is required on the Linux platform, a project such as SELinux [37] is recommended. However, it should also be kept in mind that IMon provides security guarantees that SELinux does not: for example, IMon can guarantee that an executable is uncorrupted, whereas SELinux tries to ensure that even a corrupted executable is unable to act in an untoward fashion.

Specifically, the following two attack vectors are not handled by IMon:

Buffer overflows Buffer overflows and similar methods of tricking a program into executing unauthorized code are not dealt with by IMon. If the file from which the executing process was started has a more restrictive policy than

accept_anytime attached to it, the damage done by an attacker who manages to corrupt the process image might be reduced; however, IMon makes no attempt to stop the attack from occurring in the first place.

Macros It is possible for macros embedded within documents to act in a malicious fashion. IMon makes no attempt to stop such macros, which are effectively interpreted by the document viewer, from executing; indeed, IMon has no way of telling whether the document viewer is executing a macro at a given time or not.

Despite the fact that IMon does not block the above attack vectors, it may still prevent either of the above attack vectors from resulting in a strong intrusion.

5.4 Features

The performance of IMon is comparable to that of DigSig (see 2.6), which is currently the most actively-developed kernelspace file integrity checker for the Linux operating system. However, the feature set provided by IMon is far greater than that provided by DigSig; in fact, certain features are (to the best of our knowledge) not to be found in any other file integrity checking system for the Linux operating system. Selected innovative features of IMon are:

Policy and per-file flags The ability to attach file attributes to a file to specify what it is and how it should be treated is a feature not found in any other kernelspace file integrity checker. The fact that dependencies can be specified as part of policy is also something not found in any other realtime file integrity checker, though userspace systems such as Radmind (see 2.9) do implement a form of file attribute assignment.

Transitive interpretation This feature is not found in any other realtime kernelspace file integrity checker for the Linux operating system (and is not even possible for a userspace file integrity checker to achieve). SEFL (see 2.11) attempts to address the problem of interpreters via userspace modifications of interpreters; however, that approach is flawed, as pointed out in 2.11. IMon provides a much cleaner solution that is, as far as we are able to ascertain, able to cover all cases of transitive interpretation.

Stopping standalone interpreters IMon is the only system for Linux that allows interpreters to execute only when interpreting allowed files, and disallows their standalone execution.

Binary and script equality IMon allows policy to be assigned to both binary and script executables, and allows both to be treated equally. This feature is not only not found in any other realtime integrity checker made for Linux, but has been described by Catuogno and Visconti in [10, p. 36] as "a non trivial task and probably [...] not currently possible".

5.5 Summary

In this chapter we have evaluated IMon in terms of security, performance, and applicability. We have discussed possible future attack vectors and the comprehensiveness of the security that IMon provides. In 5.4 we detail features that set IMon apart from other file integrity checkers.

Our conclusion is that IMon is a system that has a very low performance impact on a running system, and that the security it provides can prevent a strong intrusion from taking place. In addition, the system of policies that IMon provides is not found in any other realtime integrity checker, and provides a flexible way to handle the relationships between files.

Chapter 6

Conclusion

In this chapter we look back on what we have achieved during the course of this investigation, propose future research directions, and present our final conclusions.

6.1 Goals

In 1.3.2 we discussed several worthy goals and ranked them by importance. Looking back, we ask ourselves to what extent these goals have been achieved.

Comprehensive checks IMon tests every system-independent aspect of a file that it can. The checks made are comprehensive and complete, and test the meta-information of a file as well as the data within it. IMon also tests every executable file, whether binary or interpreted. This goal has been met.

Realtime checking IMon detects untrusted or unknown executables before they have been executed, and thereby helps to prevent a strong intrusion. Whilst IMon is running, no opportunity gap exists for an attacker to execute a file between scans of the filesystem. This goal has been met.

Efficiency IMon is extremely efficient and imposes no noticeable slowdown on a system. It uses a good design for looking up records, implements a caching system, and minimizes disk access to arrive at this level of efficiency. The efficiency of IMon makes a system that is running it indistinguishable from a system that is not running it, as far as a user can tell, thus achieving the secondary goal of transparency. This goal has been met.

Automation IMon does not depend on being run periodically, and takes action automatically to log and stop any possible intrusion. It enforces policy without any intervention, and has a number of policies that can be used for greater flexibility. This goal has been met.

Relevance IMon provides output only when a problem has occurred, and the output is sufficiently detailed and relevant for an administrator to be able to identify exactly what has occurred, and why it has occurred. Furthermore, all IMon system-log messages are prefixed with the string "[imon]", making it easy for an automated tool to identify any entries made by IMon. This goal has been met.

Self-protection Once running, IMon protects itself by disallowing operations that could compromise its database and stopping unknown modules from being loaded. It also makes it difficult to unload modules by restricting the capability to unload modules to one or more selected programs – and makes it impossible to unload modules if no such unloading programs are specified. IMon also verifies its database using public-key encryption before using it. This goal has been met.

Maintainability The configuration file syntax of IMon is simple and easy to use, and databases can be created by a third party and transferred to a system easily. However, no structured upgrade process has yet been implemented, though one has been designed. This goal has only partially been met: more work needs to be done before IMon can be said to be easily maintainable.

6.2 Future Work

In this section we discuss possible future work on the IMon file integrity checker. The system, as implemented in chapter 4, is both modular and highly-commented; therefore, modification should not be difficult once the core implementation has been understood.

One possible extension that we would like to see is another implementation of the design laid out in chapter 3. Ideally, this would occur on another platform such as Windows®, OS X, or FreeBSD, and thus validate our claim that the design is reasonably platform-agnostic.

The rest of this section is split into two subsections: improvements and extensions. The former discusses improvements to the current way that IMon works, and the latter discusses changing the way that IMon works to implement a new feature, or to extend a feature already present.

6.2.1 Improvements

Separation The big-number routines are only used during IMon initialization whilst verifying the integrity of the database, and are used only through a defined interface; looking at 4.1, one can see that they are not used during runtime. It is therefore possible to separate the big-number portion of IMon into a separate kernel module that may be loaded and unloaded as the need arises. This would have the benefit of a cleaner code-base.

Memory-saving measures As mentioned in 5.2.1, the memory requirements of IMon may become a scalability problem in future or on smaller systems. These memory requirements can be reduced by a number of measures, two of which are:

- Splitting the filepath member variable. To illustrate, if we have the records `/usr/bin/test` and `/usr/bin/arch` we can split them into the components `usr/bin`, `test`, and `arch`. We could then save memory by having the pathname of both records point to the same string in memory: `usr/bin`.
- Noting that each pathname begins with the '/' character, we could opt to remove this character from start of the filepath variable. This would save one char of memory per record, which does not sound like a lot until one considers that it is a saving of 500,000 chars (or half a megabyte, assuming that a char takes up one byte of memory) when 500,000 records are created.

Of course, more complex memory-saving measures – such as compressing the database within memory – are possible; these are simply two of the easier measures to implement.

Lookup cache If we assume that a record that is requested and released may be requested again within a short span of time, then we can keep a small cache of recently-requested full records instead of immediately releasing the memory associated with a record. When the cache is full, we could evict the least-recently-requested cache item and release memory associated with it, returning it to being a stub record. A cache hit would mean that no disk access needs to be made to read in a record.

Compiling a program generally involves multiple sequential invocations of the compilation tool-chain: a compiler front-end, an assembler, a linker, and so forth. Such an execution sequence could be expected to give us cache hits quite often, and the performance impact (if any) of the cache across different workloads would be interesting to examine.

Slab allocation There are certain objects that are frequently allocated and deallocated by IMon, such as imon_data structures. The Linux kernel provides a slab allocator [40, pp. 194–201] that can make such operations much more efficient. A possible improvement to IMon, then, is to alter all components that deallocate and allocate objects frequently to use the slab allocator. For example, imon_security and imon_procsec data structures are frequently created and destroyed: using the slab cache would make the functions in which such creation and destruction takes place more efficient.

6.2.2 Extensions

Upgrade process Currently, all features mentioned in chapter 3 have been implemented, with the exception of an upgrade utility and an upgrade process. These are important for IMon to be easily maintainable, and the infrastructure to support the upgrade process is already available in the IMon verify routine as well as the IMon database routines. The main difficulty in implementing this is the creation of a secure userspace upgrade utility.

De-modularize IMon may currently only be loaded as a module since its initialization phase requires a number of pieces of kernel infrastructure to be available. By shifting those pieces to a separate function and initializing at the latest possible moment, IMon can be built into the Linux kernel directly.

Password security A cryptographic digest of a password supplied as a module parameter could be used to provide additional security for unloading modules. An administrator would enter a password (via sysfs [40, pp. 291–305] or another mechanism), which would then act to allow a single module to be unloaded. Without both the password being input and the executing file having the FLG_MODCAP flag set, a module may not be unloaded.

Information provision Statistics on frequently-accessed files, memory usage, average time spent on integrity testing, and so on could be provided through the virtual filesystems procfs [40, pp. 87–88] or sysfs [40, pp. 291–305]. This information could be used by a system administrator to assess the impact of IMon, given the specific workload of a certain machine. Administrators may also be able to tune their systems by, for example, removing files that they notice have not been used for a very long time.

Same-file storage As mentioned in 3.7.6, using a database and using same-file storage are *not* mutually-exclusive design decisions. Attempting to find a same-file signature if a database lookup fails could be implemented in the get_record and put_record functions of IMon.

Additional policies The policy mechanism that IMon implements is flexible, efficient, and extensible. The policies mentioned thus far can be seen as simple examples of what could be done with it. It is left as an exercise for interested parties to experiment with creating new policies that enhance the flexibility of IMon and the security of a system.

Additional uses In 1.2.3 we discuss uses that a realtime integrity checker could be put to. With a minimal amount of work, IMon could be modified to be used for any of these purposes.

Remedial action In 3.1.4 it is briefly suggested under the **Actionable** point that remedial action could be taken in the case of certain integrity checks failing. This has not been implemented by any realtime integrity checker that we are aware of, but there seems to be no logical or technical reason that it could not be implemented. Such an extension would make the integrity checker more resistant to reversible file metadata changes such incorrect file permissions, or the incorrect file owner.

6.3 Conclusion

In this thesis we have examined projects and research related to file integrity checking, laid out the design of a best-of-breed file integrity checker, and implemented a proof-of-concept that illustrates this design. The implemented design meets almost all of the goals set out at the start of this thesis. Importantly, there are innovative features that arise from this design that, to the best of our knowledge, are not found in *any* currently-available integrity checking system that exists for the Linux operating system today.

The original problem of unauthorized code execution via a strong intrusion, as described in 1.1.1, is solved in its entirety by IMon. The security of the system has been evaluated and attacked using all cases that we considered and, whilst we do not claim that we have thought of all possible cases, we do claim that IMon is not trivially circumvented. Furthermore, performance tests carried out reveal that IMon imposes almost no performance impact on a running system. Lastly, there are several innovative features described in 5.4 that mark IMon as the best currently-available file integrity checker for the Linux operating system.

We submit that all significant goals laid out in the introductory chapter of this thesis have been met, and that IMon represents a best-of-breed file integrity checker that builds upon the work of others to produce an innovative, well-considered design and a modular, extensible implementation; we also submit that this thesis has added to the body of knowledge in the field of file integrity checking by examining crucial theoretical issues such as blacklisting and whitelisting (see 1.2.1) and exploring open issues such as metric storage (see 3.7) to arrive at conclusions that are well-supported, reasonable and coherent.

Appendix A

Glossary

A.1 Format

This glossary is in alphabetical order, with the page on which the term first occurs referenced after the term itself. Example: "***quux***", first seen on page i, would appear here as

quux, i This text defines what "quux" means.

The first time a term is referred to, it appears in ***boldface italics*** as a visual indicator that it may be looked up in future in this glossary.

A.2 Terms

atomic, 94 An atomic operation is one that either succeeds completely or does not occur at all. An atomic variable is one used in atomic operations.

baseline, 4 A baseline is a known-good set; "baseline metrics" are therefore considered to be known-good. Similarly, a "baseline file" is a file that is known-good, and a "baseline system" is one that only contains known-good files. An example of a baseline system is a newly-installed operating system.

blacklist, 3 A list of that which is denied; anything that is not on the blacklist is, by default, allowed. Blacklisting implies a default-allow security stance. See also: whitelist.

block device, 72 A device on which data can be accessed using a block number (as opposed to only being accessible via a byte offset). A block device is a raw device.

capability set, 102 The privileges accorded to a process are called its capability set.

daemon, 2 A process designed to run in the background. For example, a SSH daemon listens for incoming SSH connections and handles them appropriately.

default-allow, 5 This security stance classifies that which is unknown as being allowed.

default-deny, 5 This security stance classifies that which is unknown as being denied.

ELF, 26 ELF stands for Executable and Linkable Format, and is the default binary format of executables as well as shared and static libraries on the Linux platform.

execution profile, 11 a set of executables that is recognized as valid by the real-time file integrity checker

hard link, 72 Each file can be separated into "name" and "data". A hard link is a name by which a file's data can be known.

hash, 2 A hash is used to describe both the result of a one-way function and the function itself. A one-way function is a function $f : A \mapsto B$ for which it is difficult or impossible to create $g : B \mapsto A$. A hash function, furthermore, maps A to B in as uniformly-distributed (and therefore collision-resistant) a fashion as possible. Syn: digest, fingerprint.

honeypot, 10 a system that exists to be attacked, and to log and/or perform analyses of attacks as they are in progress.

inode, 59 Each file on a Unix-style filesystem has an *i*ndex *node* or inode associated with it. This internal kernel data structure may be uniquely identified by its inode number, and stores file metadata such as file size, owner, and timestamps.

ioctl, 28 An ioctl allows one to control a device by sending data to a file that represents that device. However, it is quite possible for the device to be virtual and have no real-world analogue; for example, the `null` device simply receives whatever data one sends to it and discards it. An ioctl can therefore be seen quite simply as a way to communicate with the kernel.

immutable, 66 In BSD parlance, this refers to an attribute of a file that makes it unmodifiable unless the securelevel is low enough *and* the user attempting to change it to not be immutable is the super-user.

kernelspace, 23 Kernelspace refers to the memory area used by the operating system kernel, device drivers, and kernel extensions. This memory may not be swapped to disk. Kernelspace processing is done from a more privileged perspective than userspace processing due to the fact that anything running in userspace is able to be affected by any processing done in kernelspace, whilst the reverse does not hold true.

malware, 7 Malware is "any software program developed for the purpose of causing harm to a computer system"[1].

one-way function, 57 A function f that takes an input and produces an output can effectively be reversed if one can find a second function, f', which takes (as input) the output of f and produces (as output) the input of f. A one-way function is a function f for which f' is difficult or impossible to find.

opportunity gap, 3 That space of time between a file being altered and the file being tested. During the opportunity gap, an invalid file is accepted as being valid.

page, 28 Memory may logically be divided into discrete units called pages, with each page being a fixed size (such as four kilobytes, or two megabytes). Each page can be granted different permissions and thus be made read-only, or execute-only, or read-write, and so forth. Storing a file or portions thereof in memory may be done by allocating an appropriate number of pages and placing the file contents into those pages. Entire pages may also be "swapped" to disk, in which case they are reloaded from disk whenever they are needed once again. The term "page" is synonymous with the term "frame", as found in Operating Systems literature.

paging unit, 92 The hardware paging unit translates linear addresses, which are logical in nature, to physical addresses, which refer to the correct location in memory.

[1] *Wikipedia*, `http://en.wikipedia.org/wiki/Malware`, 5 July 2005 14:56

permissions, 3 A traditional Unix model for filesystem security uses bits that specify which users are allowed to perform which operations on a file. These bits are called permissions, or one may refer to them as the mode of a file. Traditionally, read/write/execute permissions may be specified separately for the file owner, for members of the file group, and for all other entities.

raw device, 29 A file resides on a filesystem; in turn, a filesystem typically resides on a hard drive, flash stick, memory, or other such storage. This storage that a filesystem resides on is called a raw device, and modifications to a raw device occur beneath the level of the filesystem – and thus beneath the level at which the filesystem can detect them.

rootkit, 1 This is a specific type of trojan horse that replaces important system files (usually executables or kernel modules of some sort) with malware designed to hide suspicious activity. Rootkits can open up "backdoors" (unauthorized gateways for an attacker into the system) or attack other systems, among other behaviours. It is usually difficult to detect the presence of a rootkit as they are designed to make the system act as though it is functioning perfectly normally – or at least report that it is functioning normally when it is not.

securelevel, 34 In BSD parlance, the securelevel of the operating system refers to an increment-only variable that determines how security policies on a system should be enforced.

spyware, 1 A program that collects and sends information about the user across a medium (usually a network), without the user being aware of it.

swapfile, 115 A file used to simulate a greater amount of primary memory than actually exists on a system.

sysctl, 35 A sysctl is used to configure kernel parameters at runtime. It is a way of modifying a predefined variable that exists within the kernel, after appropriate kernel-level checks have been done.

trapdoor function, 85 A function or computation that is difficult to reverse without knowing a certain value, but that is easy to reverse once this value is known.

trojan horse, 1 This is malware disguised as a useful program. A trojan horse performs unauthorized (and usually unwanted) actions in addition to performing a useful function; these actions usually include creating a "back-door" into the system for an attacker to use.

unauthorized code, 13 Executable content (whether native or interpreted) that an authorized entity (such as an organization, system administrator or user) has not specifically allowed the execution of.

userspace, 23 Userspace refers to the memory area used by all user mode applications. This memory may be swapped to disk at any time. Userspace applications are less privileged than kernelspace applications due to the fact that they are treated as processes under the control of the kernel.

virus, 1 Executable code that attaches itself to a host (such as an executable binary, shared library or script), either appending itself to the host or overwriting portions (sometimes the entirety!) of the host. Usually infects other hosts either by its own initiative or by being unwittingly propagated by a user's action.

whitelist, 3 A list of that which is allowed; anything that is not on the whitelist is, by default, denied. Whitelisting implies a default-deny security stance. See also: blacklist.

worm, 15 Similar to a virus, but does not require a host. A standalone program that infects other machines; usually alters the system to ensure that it is executed across reboots. Worms generally feature active replication across a network.

Appendix B

Set Notation

In this Appendix we briefly describe standard set notation for those readers who are not as familiar with mathematics as they are with computer science.

A set of elements or items is usually denoted by a capital letter such as A, G, or P. This capital letter may be decorated, as in $\hat{A}$ or $\hat{Z}$. The symbol $\emptyset$ denotes the *empty set*, which is the set that contains no elements.

$A \cap B$ produces the *intersection* of sets A and B; that is, the set comprising those elements that are common to both sets.

$A \cup B$ produces the *union* of sets A and B; that is, the set comprising those elements that are in either or both sets.

$A = B$ indicates that sets A and B are equal, and therefore contain the same elements.

$A \neq B$ indicates that sets A and B are not equal, and therefore at least one element of them differs.

$A \cap B \Leftrightarrow C \cup D$ indicates that $A \cap B$ implies and is implied by $C \cup D$; viewed another way, it shows that both sides of the $\Leftrightarrow$ are a way of saying one thing.

$A \setminus B$ produces the set which contains all elements of A that are not also elements of B.

$A \subset B$ indicates that all elements of A are also elements of B.

$A \nsubseteq B$ indicates that there are some elements of A that are not elements of B.

Appendix C

Big-Number Implementation

This appendix looks at the big-number implementation that was developed. The implementation is, through extensive use of the C preprocessor, compilable both as a standalone program that tests algorithms and as part of the IMon kernel module. Code Snippets presented in this appendix are *not* complete, and are generally presented in a form that ignores memory management and error handling. The addition, subtraction, and multiplication algorithms are based on those described by Knuth in [32, pp. 265–270].

C.1 Data Structure

```
#DEFINE INT_TYPE UNSIGNED LONG
typedef struct {
    signed int used; /* Index of last cell used */
    int allocated; /* Number of cells available */
    INT_TYPE *cell; /* the cells themselves */
} bignum;
```

Code Snippet C.1: Big-number data structure

Code Snippet C.1 shows the basic data structure that represents a big-number: a bignum. Each bignum is allocated a certain number of "cells", which may be altered during the course of operations; cell is an array of unsigned long integers, each of which is half-filled with data. In other words, if an unsigned integer is 32 bits long, 16 of those bits is used in each of the elements of the cell array. This is done to make multiplication (see C.5) easier.

C.1.1 Allocation

Before a bignum, X, is used, it must be passed as an argument to the bn_init function. This function allocates CELLSIZE elements (where CELLSIZE is some constant, currently 128) to X.cell, and changes X.allocated to reflect this. It then zeroes all elements of X.cell.

All allocations of memory occur in blocks of CELLSIZE elements. If CELLSIZE is too small, then many reallocations will be done to increase the amount of memory allocated to a bignum in the course of its lifetime; if it is too large, then the excess memory will be wasted. 128 was chosen as a trade-off between the two.

C.1.2 Deallocation

Once a bignum is no longer required, it must be passed to the bn_fini function, which deallocates the memory allocated to it and sets the allocated member to be zero. After deallocation, any attempt to use the bignum without passing it to bn_init beforehand leads to undefined behaviour!

C.2 Utility Functions

Certain common operations to be performed using bignums are not supported in a syntactically-friendly fashion by C. Functions needed to be created to

- Set a bignum, or a particular section of a bignum, to zero: bn_clear(bignum*), bn_clear_from(bignum*, int), bn_clear_until(bignum*, int).
- Assign a bignum to another bignum: bn_assign(bignum* to, bignum* from).
- Output a bignum in hexadecimal: show_num_hex(bignum*).
- Append a value to the end of a bignum: bn_append(bignum*, unsigned long).
- Read in a bignum in hexadecimal: bn_get_num(bignum*, const char*).
- Compare two bignums, returning a signed integral result indicating whether the first operand was less than, equal to, or greater than the second: bn_cmp(bignum*, bignum*).

Creating these functions is an interesting but elementary exercise; they shall not be discussed here.

C.3 Addition

```
int bn_add(bignum *res, const bignum *a, const bignum *b) {
  int ret;
  long end = bn_max(a→used, b→used), carry = 0;
  bn_clear(res);
  res→used = end;
  for (long j = 0; j <= end; ++j) {
    res→cell[j] = a→cell[j] + b→cell[j] + carry;
    carry = (res→cell[j] & (1 << 16)) >> 16;
    if (carry) {
      res→cell[j] &= 0xffff; /* remove 'carry' bit */
    }
  }
  if (carry) {
    if (res→used+1 >= res→allocated) bn_alloc_more(res);
    res→cell[++res→used] = 1;
  }
  return 0;
}
```

Code Snippet C.2: Addition

This code assigns the result of adding a and b to res. We start by zeroing res, then add each cell from a and each cell from b together; if there is an overflow, we add the "carry" to the next cell. Once all additions have been done, we check for a final "carry", then return successfully.

C.4 Subtraction

```
int bn_sub(bignum *res, const bignum *a, const bignum *b) {
  long end = a→used;
  bn_clear(res);
  res→used = end;
  short carry = 0;
  while (res→allocated <= end) bn_alloc_more(res);
  for (long j = 0; j <= end; ++j) { /* Going from least to most significant */
```

```
    long tmp = (0x10000 + a→cell[j]) – carry – b→cell[j];
    res→cell[j] = 0xffff & tmp;
    carry = tmp & 0xffff0000 ? 0 : 1;
  }
  for (int i = bn_min(res→allocated–1,end); i && !res→cell[i]; ––i) res→used = i–1;
  return 0;
}
```

Code Snippet C.3: Subtraction

This function subtracts b from a, assuming that the former is smaller than the latter, and assigns the result to res. The logic follows that used in C.2: we start at one end of the array, subtract cell from cell, and compensate for borrows as we go along. Once all subtractions have been done, we set res.allocated appropriately and return successfully.

C.5 Multiplication

```
001 int bn_mul(bignum *res, const bignum *a, const bignum *b) {
002   const long n = a→used, m = b→used;
003   bn_clear_until(res,m+n+2);
004   a→cell[n+1] = 0;/* Necessary, in case a→used+1 has junk in it */
005   for (long j = 0; j <= m; ++j) {
006     unsigned long carry = 0;
007     for (long i = 0; i <= n+1; ++i) {
008       unsigned long temp = a→cell[i] * b→cell[j] + res→cell[i+j] + carry;
009       res→cell[i+j] = temp & 0xffff;
010       carry = temp >> 16;
011     }
012     res→cell[j] &= 0xffff;
013   }
014   res→used = m+n–1 >= 0 ? m+n–1 : 0;
015   int end = bn_min(res→used+3, res→allocated);
016   for (int i = 0; i < end; i++) {
017     if (res→cell[i]) res→used = i;
018   }
019   return 0;
020 }
```

Code Snippet C.4: Multiplication

Multiplication is implemented naïvely through a series of cumulative additions, and ends up with the product of a and b placed into res. We start on line 3 by zeroing the result until index m+n+2, which is the maximum size of the product of the two operands. We begin two for loops that do cumulative addition on line 5, using the res.cell array to store partial results; and, noting that we go to n+1 on line 7 to deal with overflow from a previous loop iteration, we set that position to zero on line 4.

The outer loop merely contains the inner; and the inner loop body should be familiar if C.2 has been studied as it follows much the same idea of doing an operation, then compensating for overflow. Lines 15–18 determine where the end of res lies, and on line 19 we return successfully. Note that multiplying two unsigned integral values, each of which is n bits long, will give a result that is a maximum of $2 \cdot n$ bits in length; this is why it is important to store only $\frac{n}{2}$ bits in each n-bitlength cell (see C.1). If we did not store only $\frac{n}{2}$ bits in each cell, part of the product from a given multiplication done on line 8 might be lost, leading to an incorrect result!

Similar to the above, there is also a bn_mul_ui function that multiplies a bignum by an unsigned long. This function does not require an inner loop, as each cell is simply multiplied by a single fixed number to return the product: this is the only real difference between the above code and bn_mul_ui.

C.6 Division

```
001 void bn_div(bignum *quotient, bignum *remainder, const bignum *a, const bignum *b) {
002   bignum working_dividend, result, dividend, divisor;
003   bn_assign(&dividend, a); bn_assign(&divisor, b);
004   const unsigned long d = 0x10000/(divisor.cell[divisor.used]+1);
005   int temp = −1, m = 0;
006   unsigned long q_hat;
007   bn_mul_ui(&divisor, &divisor, d);
008   bn_mul_ui(&dividend, &dividend, d);
009   for (long j = dividend.used; j >= 0;) {
010     while (bn_cmp(&working_dividend, &divisor) < 0) {
011       bn_append(&working_dividend, dividend.cell[j]);
012       quotient→cell[j] = 0;
013       j—;
014       if (j < 0) break;
015     }
016     if (bn_cmp(&working_dividend, &divisor) < 0) break;
017     if (temp == −1) { /* i.e., this is the first number to be placed on */
018       temp++;
```

```
            m = j+1 >= 0 ? j+1 : 0;
        }
        if (working_dividend.cell[working_dividend.used] == divisor.cell[divisor.used]) {
            q_hat = 1;
        } else {
            q_hat = ((working_dividend.cell[working_dividend.used] << 16) |
              (working_dividend.used > 0 ? working_dividend.cell[working_dividend.used−1] : 0))
              / divisor.cell[divisor.used];
            if (q_hat & 0xffff0000) q_hat >>= 16;
        }
        if (q_hat == 1) {
            bn_mul_ui(&result, &divisor, 0xffff);
            if (bn_cmp(&working_dividend, &result) >= 0) {
                q_hat = 0xffff;
                goto qhat_correct; /* optimization */
            } else { /* Passed? Well, try this one. */
                bn_mul_ui(&result, &divisor, 0xfffe);
                if (bn_cmp(&working_dividend, &result) >= 0) {
                    q_hat = 0xfffe;
                    goto qhat_correct; /* optimization */
                }
            }
        }
        bn_mul_ui(&result, &divisor, q_hat);
        if (bn_cmp(&working_dividend, &result) < 0) {
            bn_mul_ui(&result, &divisor, —q_hat);
            if (bn_cmp(&working_dividend, &result) < 0) {
                bn_mul_ui(&result, &divisor, —q_hat);
            }
        }
qhat_correct:
        quotient→cell[j+1] = q_hat;
        bn_sub(&working_dividend, &working_dividend, &result);
    }
    quotient→used = m;
    bn_div_ui(remainder, NULL, &working_dividend, d);
}
```

Code Snippet C.5: Division

Division provides us with an interesting problem: whilst addition, subtraction, and multiplication could all be done in a "schoolbook" fashion – much as one would do them by hand, on paper – division requires one to divide in order to divide. A quick reflection on how division is done by-hand shows that this is true: to obtain a digit of the quotient, one must first find the minimum number of significant dividend figures that form a number greater than (or equal to) the quotient, and then divide one by the other to find the quotient digit in question.

To accomplish division, we use part of the algorithm found in [32, pp. 271–274], and assume in the code above that the divisor is smaller than or equal to the dividend. The first step in this algorithm is to normalize divisor and dividend. We do this by multiplying each one by d, defined on line 4 to be

$$\frac{\text{cell\_maximum\_value} + 1}{\text{most\_significant\_cell}}$$

assuming a 32-bit architecture. This normalization assures us that the process of finding a good estimate for a quotient digit[1] (lines 21–28) works, and finds a digit that is at most 2 greater than the correct digit. After normalization, the steps taken diverge somewhat from those of the algorithm, and are as follows:

Bring-down, 10–15 A working dividend, consisting of significant digits from the dividend proper, is created as being just larger than the divisor. Each time that the working dividend is *not* greater, we add a zero into the correct position in the quotient. If the dividend proper ends whilst the working dividend is less than the divisor (line 14) then we break out of the loop.

Exit-test and set-length, 16–20 Line 16 tests if working_dividend is the remainder; if so, we break out of the loop. Lines 17–20 are only executed once, using temp as a flag variable, to set the final length of the quotient so that we don't have to calculate it at the end.

Calculating $\hat{q}$, 21–28 q_hat, or $\hat{q}$, is the trial quotient value that we arrive at. We calculate this by Knuth's method, by assigning $\hat{q} = 1$ if the most significant digits of the working dividend and the divisor match, and calculating $\hat{q}$ based on the two most significant digits of the working dividend and the most significant digit of the divisor; if this results in overflow, we shift to the right to get the correct answer.

Estimation correction, 29–48 As noted above, the estimate made may be 1 or 2 greater than it should be. Lines 29–41 test to see whether the estimate should "wrap around" to become 0xffff or 0xfffe instead of 1; lines 42–48 test all other

[1]Note that when we say "digit" in this context, we are referring to an entire 16-bit number, a full cell of the quotient answer.

cases. Between 1 and 3 multiplications by an unsigned integer will occur in this entire step. By the end of this step, result holds the product of q̂ and divisor, and q̂ is correct.

Quotient digit, subtraction, 50–51 The quotient digit is assigned and subtraction of result from working_dividend occurs to give us the new working dividend that should be appended to.

Remainder, 54 At this point, after the loop has ended and all digits of the quotient have been calculated, the remainder is found by calling bn_div_ui to unnormalize it. This function follows the much same steps that bn_div does, save for the **calculating q̂** and **estimation correction** steps; this is because since our divisor is only the length of a single bignum.cell element, we can divide without needing to do these steps at all.

C.7 Power-Modulus

```
001 int _bn_powm(bignum *a, const bignum *b, const bignum *c, const bignum *d) {
002   int ret = 0;
003   bignum bPower[4], temp;
004   bn_clear_until(a,1);
005   a→cell[0] = 1;
006   if (c→used == 0 && c→cell[0] == 0) {
007     if (d→used == 0 && d→cell[0] == 1)
008       a→cell[0] = 0;
009     goto exit; /* 'a' is fine. */
010   }
011   if (b→used == 0 && b→cell[0] == 0) {
012     a→cell[0] = 0;
013     goto exit; /* Always zero */
014   }
015   bPower[0].cell[0] = 1; /* bP[0] = 1 */
016   bn_assign(&bPower[1], b); /* bP[1] = b */
017   bn_mul(&bPower[2], b, b); /* bP[2] = b*b */
018   bn_mul(&bPower[3], &bPower[2], b); /* bP[3] = b*b*b */
019   bn_div(NULL, &bPower[1], b, d);
020   bn_div(NULL, &bPower[2], &bPower[2], d);
021   bn_div(NULL, &bPower[3], &bPower[3], d);
022   int i = c→used;
023   INT_TYPE ci = c→cell[i];
024   int ciBits = 16;
```

```
  while (!(ci & 0x8000)) { ci <<= 1; ciBits—; }
  if (ciBits % 2) ci >>= 1;
  short j;
  do {
    j = 0;
    do {
      bn_mul(a, a, a);
      bn_mul(a, a, a);
      bn_div(NULL, a, a, d);
      bn_mul(&temp, a, &bPower[(0xC000 & ci) >> 14]);
      bn_div(&temp, a, &temp, d);
      ci <<= 2;
      j += 2;
    } while (j < ciBits);
    ciBits = 16;
    i -= 1;
    if (i >= 0) ci = c→cell[i];
  } while (i >= 0);
exit:
  return ret;
}
```

Code Snippet C.6: Power-Modulus

Power-Modulus is the operation a^b *modulus* c, as used in RSA [57] decryption. We build upon the functionality discussed above to create the above function. The mathematics underlying this function are beyond the scope of this discussion, and are described in [59].

Bibliography

[1] Ross J. Anderson and Eli Biham. TIGER: A Fast New Hash Function. In Dieter Gollmann, editor, *Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 1996.

[2] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. DigSig: Run-time authentication of binaries at kernel level. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, November 2004. Pre-print.

[3] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. The DigSig project. Technical report, March 2004. Partly published in Linux World, vol 2(1), December 22, 2003.

[4] William A. Arbaugh. A Patch in Nine Saves Time? *Computer*, 37(6), June 2004.

[5] Boris Balacheff, Liqun Chen, David Plaquin, and Graeme Proudler. *Trusted Computing Platforms*. Prentice Hall PTR, July 2002.

[6] Steven M. Beattie, Andrew P. Black, Crispin Cowan, Calton Pu, and Lateef P. Yang. CryptoMark: Locking the Stable door ahead of the Trojan Horse. Technical report, WireX Communications Inc., July 2000.

[7] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel.* O'Reilly, 1005 Gravenstein Highway North, Sebastopol, California 95472, United States of America, 2nd edition, February 2003.

[8] Ed L. Cashin. integrit file verification system. Available: `http://integrit.sourceforge.net/`, accessed 2005/Jul/11.

[9] Luigi Catuogno and Ivan Visconti. A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code. In *Proceedings of the Third*

Conference on Security in Communication Networks. Dipartimento di Informatica ed Applicazioni, Universita di Salerno, September 2002. Available: `http://libeccio.dia.unisa.it/wlf/scn02/index.html`, accessed 2005/Nov/12.

[10] Luigi Catuogno and Ivan Visconti. An Architecture for Kernel-Level Verification of Executables at Run Time. *The Computer Journal*, 47(5), September 2004.

[11] Pravir Chandra, Matt Messier, and John Viega. *Network Security with OpenSSL*. O'Reilly, June 2002.

[12] Crispin Cowan, Chris Wright, James Morris, Greg Kroah-Hartman, and Stephen Smalley. Linux security module framework. In *Ottawa Linux Symposium 2002*, Ottawa, Canada, June 2002.

[13] Crispin Cowan, Chris Wright, James Morris, Greg Kroah-Hartman, and Stephen Smalley. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, United States of America, August 2002.

[14] Wesley D Craig and Patrick M McNeal. Radmind: The integration of filesystem integrity checking with filesystem management. In *Proceedings of the 17th USENIX Large Installation System Administration Conference (LISA 2003)*, October 2003.

[15] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD (Orange Book).

[16] Guilherme Herrmann Destefani. Verificação Oportunista de Assinaturas Digitais Para Programas e Bibliotecas em Sistemas Operacionais Paginados. Master's thesis, Centro Federal de Educação Tecnológica do Paraná, March 2005. Thesis is in Portuguese; source code was used as documentation.

[17] Jeff Dike. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, November 2001.

[18] Ulrich Drepper and Scott Miller. *sha1sum(1)*. Free Software Foundation Inc., 2005. system manpage.

[19] D Eastlake and P Jones. RFC 3174 - US Secure Hash Algorithm 1 (SHA1). Technical report, Motorola; Cisco Systems, September 2001. Available: `http://www.faqs.org/rfcs/rfc3174.html`, accessed 2005/Nov/12.

[20] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, 2003.

[21] David Geer. Malicious bots threaten network security. *Computer*, 38(1):18–20, January 2005.

[22] Eric Gerbier. Another file integrity checker, June 2005. Available: `http://afick.sourceforge.net/`, accessed 2005/Jul/11.

[23] Tim Grant. Ten tips to mitigate software license risks. *IT Audit*, 7, April 2004.

[24] The Shmoo Group. Osiris user handbook, 2005. Available: `http://www.hostintegrity.com/osiris/handbook.html`, accessed 2005/Jul/11.

[25] halflife. Bypassing Integrity Checking Systems. *Phrack*, 7(51), September 1997.

[26] Microsoft Inc. Authenticode overviews and tutorials (internet explorer), 2005. Available: `http://msdn.microsoft.com/workshop/security/authcode/authenticode_ovw_e%ntry.asp`, accessed 2005/Nov/12.

[27] Global Data Integrity. Xintegrity home. - data integrity assurance, network security and host based intrusion detection for windows. Available: `http://www.xintegrity.com/`, accessed 2005/Jul/11.

[28] Ionx. Ionx - data sentinel. Available: `http://www.ionx.co.uk/html/products/data_sentinel/`, accessed 2005/Jul/11.

[29] Aditya Kashyap, Jay Dave, Harikesavan Krishnan, Charles P Wright, Mohammed Nayyer Zubair, and Erez Zadok. Using berkeley db in the linux kernel, 2004. Available: `http://www.cs.sunysb.edu/~csgsc/grc2004/abstracts/filesys/kbdb.pdf`, accessed 2005/Nov/12.

[30] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[31] Donald E Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[32] Donald E Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 2nd edition, 1998.

[33] Rami Lehti. The aide manual. Available: `http://www.cs.tut.fi/%7Erammer/aide/manual.html`, accessed 2005/Jul/11.

[34] Rocksoft Limited. Rocksoft.com. Available: http://www.rocksoft.com/rocksoft/veracity/, accessed 2005/Jul/11.

[35] Thomas Linden. www.daemon.de - nabou, August 2004. Available: http://www.daemon.de/Nabou, accessed 2005/Jul/11.

[36] Thomas Linden. www.daemon.de - nabouconfig, August 2004. Available: http://www.daemon.de/NabouConfig, accessed 2005/Jul/11.

[37] P Loscocco and S Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[38] Robert Love. Why Not dnotify and Why inotify, September 2004. Available: http://www.kernel.org/pub/linux/kernel/people/rml/inotify/README, accessed 2005/Nov/12.

[39] Robert Love. *inotify_init(2)*, July 2005. online manpage.

[40] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, June 2005.

[41] GFI Software Ltd. Intrusion detection for windows 2000. Available: http://www.gfi.com/lansim/, accessed 2005/Jul/11.

[42] Hongjiu Lu. Elf: From the programmer's perspective. Technical report, May 1995. Available: citeseer.ist.psu.edu/lu95elf.html.

[43] Brett Lymn. Verified exec - extending the security perimeter. Technical report, December 2003. Presented at linux.conf.au 2004.

[44] Brett Lymn. Verified executables for netbsd. Technical report, July 2003. Available: http://www.users.on.net/%7Eblymn/veriexec/, accessed 2005/Jul/11.

[45] Brett Lymn. heads up: major changes to verified exec. NetBSD 'current-users' mailing-list post, April 2005. Available: http://mail-index.netbsd.org/current-users/2005/04/20/0007.html, accessed 2005/Nov/12.

[46] Brett Lymn and Jay Fink. Preventing the unauthorised binary. Technical report, January 2000. Presented at the 2000 Australian Unix Users Group summer conference.

[47] M Slaviero and M S Olivier. Attacking Signed Binaries. In *Southern African Telecommunication Networks and Applications Conference (SATNAC) 2004 Proceedings*, June 2005.

[48] Robert McMillan. Ibm researchers take axe to computer security. InfoWorld news article, October 2005. Available: `http://www.infoworld.com/article/05/10/28/HNibmaxe_1.html`, accessed 2005/Nov/12.

[49] Yusuf M Motara and Barry V Irwin. File integrity checkers: State of the art and best practices. In *Proceedings of the Information Security South Africa (ISSA) 2005 Conference*, June/July 2005.

[50] Yusuf M Motara and Barry V Irwin. In-kernel cryptographic executable verification. In *Proceedings of the IFIP WG 11.9 First International Conference on Digital Forensics*, February 2005.

[51] Jason Nieh and Ozgur Can Leonard. Examining vmware. *Dr. Dobb's Journal*, August 2000.

[52] Bill Nowicki. RFC 1094 - NFS: Network File System Protocol Specification. Technical report, Sun Microsystems Inc., March 1989. Available: `http://www.faqs.org/rfcs/rfc1094.html`, accessed 2005/Nov/12.

[53] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. Stony Brook University.

[54] Peter Pikosz and Johan Malmqvist. A comparative study of engineering change management in three swedish engineering companies. In *Proceedings of DETC98 ASME Design Engineering Technical Conference*, Atlanta, Georgia, United States of America, September 1998.

[55] H Zero Seven 'Risk'. Antiexploit documentation — h zero seven. Available: `http://www.h07.org/?q=node/37`, accessed 2005/Jul/11.

[56] Ron Rivest. RFC 1321: The MD5 message-digest algorithm. Technical report, RSA Data Security Inc., April 1992. Available: `http://www.faqs.org/rfcs/rfc1321.html`, accessed 2005/Nov/12.

[57] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[58] Allan Saddi. Yafic homepage. Available: `http://www.saddi.com/software/yafic/`, accessed 2005/Jul/11.

[59] Anuj Seth. Data encryption page - [www.anujseth.com]. Webpage, 2005. Available: `http://www.anujseth.com/crypto/bignumbers.php`, accessed 2005/Nov/12.

[60] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC 3010 - NFS version 4 Protocol. Technical report, Sun Microsystems Inc.; Hummingbird Ltd.; Zambeel Inc.; Network Appliance Inc., December 2000. Available: `http://www.faqs.org/rfcs/rfc3010.html`, accessed 2005/Nov/12.

[61] Marc Singer. *bsign(1)*. The Debian Project, January 2001. system manpage for version 0.4.5.

[62] Sherri Sparks and Jamie Butler. Raising the Bar for Windows Rootkit Detection. *Phrack*, 11(63), July 2005.

[63] The NetBSD Developers. *Chapter 19. NetBSD Veriexec subsystem.* The NetBSD Foundation, September 2005. Available: `http://www.netbsd.org/guide/en/chap-veriexec.html`, accessed 2005/Nov/12.

[64] The Open Group. IEEE Standard 1003.1: Single Unix Specification. Specification, IEEE; The Open Group, 2004. Version 3, 2004 Edition.

[65] Erik Thiele. yyyrsa, 2003. Available: `http://www.erikyyy.de/yyyRSA/`, accessed 2005/Nov/12.

[66] Tool Interface Standards Committee. Executable and Linkable Format (ELF). Specification, Unix System Laboratories, 2001.

[67] Tripwire Inc. Tripwire for Servers Datasheet. Technical report, Tripwire Inc., 2005. Available: `http://www.tripwire.com/files/literature/product_info/Tripwire_for_Serv%ers.pdf`, accessed 2005/Nov/12.

[68] Leendert van Doorn, Gerco Ballintijn, and William A. Arbaugh. Signed Executables for Linux. Technical Report CS-TR-4259, University of Maryland, June 2001.

[69] Vinicius da Silveira Serafim and Raul Fernando Weber. The SOFFIC Project. Technical report, UFRGS Security Group - GSeg. Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil, 2002.

[70] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on sha1. Technical report, Shandong University, February 2005. Research note.

[71] Xiaoyun Wang and Hongbu Yu. How to Break MD5 and Other Hash Functions. 2005. To appear.

[72] Rainer Wichmann. samhain file system integrity checker. Available: `http://la-samnha.de/samhain/index.html`, accessed 2005/Aug/30.

[73] Rainer Wichmann. Samhain labs | file integrity checkers, October 2004. Available: `http://la-samhna.de/library/scanners.html`, accessed 2005/Nov/12.

[74] David J Farber William A Arbaugh and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[75] Michael A. Williams. Anti-Trojan and Trojan Detection with In-Kernel Digital Signature testing of Executables. Technical report, Security Software Engineering: NetXSecure NZ Limited, April 2002.