

ESTABLISHING TRUST IN ENCRYPTED PROGRAMS

A Thesis
Presented to
The Academic Faculty

by

Ying H. Xia

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2008

Copyright 2008 by Ying H. Xia

ESTABLISHING TRUST IN ENCRYPTED SOFTWARE

Approved by:

Dr. Henry Owen, Advisor
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Randal Abler
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Jim Hamblen
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. John Copeland
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Jon Giffin
College of Computing
Georgia Institute of Technology

Date Approved: June 30, 2008

For my father

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Henry Owen. His support and advice guided me through my years of Ph.D. study and enabled me to focus on my goal.

I would also like to thank my committee members, Dr. John Copeland, Dr. Randal Abler, Dr. Jim Hamblen, and Dr. Jon Giffin. Thank you for your support and availability on a short schedule.

I am also grateful for those in the Network Security and Architecture Lab, Kevin Fairbanks and Michael Nowatowski for their assistance in my research. Also, much appreciation is extended to those in the Communications Systems Center, Chris Lee, Yusun Chang, Bongkyoung Kwon, and Selcuk Uluagac for their valuable time and friendship.

Lastly, I would like to thank my parents for their understanding, especially my father, who suffered from bad health yet still encouraged me onward. And special acknowledgement to my fiancée Diana, whose selfless support during my bad days made all of this possible.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
SUMMARY	x
<u>CHAPTER</u>	
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Dissertation Outline	2
2 BACKGROUND	3
2.1 Terms	5
2.2 Operating System Model	6
2.3 Related Work	7
2.4 System Calls	14
3 SURVEY OF MONITORING AND COUNTERMEASURE TECHNIQUES	18
3.1 Program Monitoring	18
3.2 Anti Monitoring Techniques	23
3.3 Examples of Encrypted Programs	30
4 ON DEMAND SYSTEM CALL MONITORING METHODOLOGY	34
4.1 Architecture and Methodology Overview	34
4.2 Trust Development	38
4.3 Intrusion Detection	40
4.4 System Operation	40

5	EVALUATION OF PROGRAM MONITORING	44
5.1	Program Monitoring	44
5.2	Results	45
6	PROGRAM BEHAVIOR PROFILING	50
6.1	Architecture	50
6.2	Database Generation	52
6.3	Results	56
7	VISUALIZATION	62
7.1	Tools	64
7.2	Approach	64
7.3	Results	68
8	CONCLUSIONS AND RECOMMENDATIONS	72
8.1	Future Work	73
	REFERENCES	75
	PUBLICATIONS	78
	APPENDIX A: LINUX SYSTEM CALLS	80
	VITA	88

LIST OF TABLES

	Page
Table 1: Example policies	39
Table 2: Sample /bin/cat file list	57
Table 3: Vi and vim accuracy comparison	60

LIST OF FIGURES

	Page
Figure 1: Operating system model	6
Figure 2: Breakdown of sequences of system calls	8
Figure 3: Invoking System Calls	14
Figure 4: Simple code injection example	20
Figure 5: Stack smashing	21
Figure 6: The decompilation of code	23
Figure 7: Multiple layers of encryption	25
Figure 8: Hidden mov instruction	26
Figure 9: Breakpoint detection	27
Figure 10: Opaque predicate example	29
Figure 11: Modified system call layer	35
Figure 12: Three possible locations for system call monitoring	36
Figure 13: Raw sys_open data	38
Figure 14: Modified system call layer	42
Figure 15: Sys_open calls of skype	46
Figure 16: Monitoring an exploit	47
Figure 17: Linux RST.B-1 modifying bin processes	48
Figure 18: The executions of Emacs	49
Figure 19: Forensics Process	51
Figure 20: Raw journal data	53
Figure 21: Data analysis procedure	55
Figure 22: [Click here and type figure title.]	58

Figure 23: Distance measure analysis	58
Figure 24: Dominant feature extraction accuracy	59
Figure 25: Read-write relationship accuracy	61
Figure 26: Visualization concept	66
Figure 27: Propagation algorithm	67
Figure 28: Process tree	69
Figure 29: File package highlighting	70
Figure 30: Taint propagation	71

SUMMARY

We propose a methodology to develop trust in encrypted programs. The goal of this research is to provide system administrators, users, and security personnel with the information necessary in order to safely execute encrypted programs on their systems without compromising their sensitive data. Traditional monitoring techniques aim to observe program behavior and characteristics in order to detect potential security weaknesses. However, there are many program encryption techniques designed to defeat many of the current monitoring approaches. Our goal is not to defeat the encryption of a program, but to inform the user if the encrypted program is behaving in a dangerous way. There is no such methodology available at present to perform this function.

In our work, we present the results of implementing on demand system call monitoring, which uses a policy based and behavior based intrusion detection system to ensure that a program is not compromised or is accessing data in an unsafe manner. We believe that implementing this layer minimizes the changes to the operating system thus lowers the probability of incompatibility with executing encrypted software. Further, we use the data gathered from this monitoring to provide several types of program analysis and detection.

CHAPTER 1

INTRODUCTION

Encryption is increasingly being used as deterrence for software piracy and vulnerability exploitation. Unencrypted or insecure programs can be the subject of intensive scrutiny by attackers in an attempt to disable protective features or to find buffer overflows as an avenue of attack of other systems. The application of encrypted programs, however, leads to other security concerns as users are no longer able to distinguish between malicious and benign behavior due to the secretive nature of encryption. Furthermore, should an attacker gain access to the software update process then malicious updates or modifications can be made to the system without the knowledge of the users. Therefore, system administrators running encrypted software now have a need for techniques that would allow such encrypted software to execute properly while minimizing the possibility of the system being compromised. The goal of this research is to develop a methodology that can enable users to trust encrypted software to allow their execution.

1.1 Motivation

Although executable encryption techniques have been around since the 1980s [10], the generated encrypted executables were either too easy to decrypt or not mainstream applications. This situation was changed in 2003 when Niklas Zennström and Janus Friis developed Skype, a peer-to-peer Voice-over-IP (VoIP) software [2]. This software employed many new and innovative encryption methods and it has the ability to bypass most firewalls by using an overlay network to transmit voice data. Currently with over 150 million downloads and 50 million registered users, Skype is perhaps the most widely distributed encrypted software in the world. This rapid growth, however, raise concerns from system administrators who fear that the software may be used as a

backdoor by attackers to gain access into the computers using covert channels embedded into the VoIP traffic. Today, many techniques and methods have been developed for malicious software detection; however, they are not as effective against the polymorphic nature of encrypted programs such as Skype. This presents a dilemma for system administrators as they must make the tradeoff between restricting certain software and potentially compromising system security.

Currently, we are not aware of any program monitoring system available for observing and establishing trust in encrypted software. The purpose of this research is to develop a methodology that would enable users to minimize their risk potential. By doing so, we aim not to reverse-engineer encrypted software but to allow users to safely execute these programs while remaining secure with the knowledge that their sensitive information and data are being protected. To do so, we must examine the program monitoring and encryption techniques as well as methods to safeguard systems from unknown software attacks.

1.2 Dissertation Outline

Chapter 2 provides the background information on prior program monitoring approaches and definitions of some important concepts and terms. Additionally, an overview of key related research is discussed. Chapter 3 provides a survey of current program monitoring and encryption techniques. Chapter 4 presents our methodology for monitoring encrypted programs and establishing trust. Evaluations of our approach by testing a prototype against several types of programs and intrusions are presented in Chapter 5 through 7. Finally, chapter 8 presents our conclusions and avenues of future work.

CHAPTER 2

BACKGROUND

The continued battle between software vendors and piracy has introduced new technology that reveals shortcomings of current security mechanisms and monitoring systems. For instance, Apple has devised several measures such as encrypting some key applications to make it “non-trivial” to pirate Mac OS X [1]. Other notable software that employs extensive encryption techniques includes Skype [2] and program packers such as Burneye [3], EXECryptor [4], UPX [5], and Shiva [6]. The main motivations for program encryption are for the following reasons:

- **Illegal Distributing** – One of the biggest concerns for the software industry is the loss of revenue due to the illegal distribution of proprietary software. Companies can employ program encryption and only allow users with legal software keys.
- **Tampering** – Illegal software vendors often modify the binary of the program to disable any checks for legality of the software. By encrypting the program, the software companies can hinder such tampering.
- **Reverse-Engineering** – Companies may not wish to show their proprietary code to rival companies, thus program encryption can be used to hinder reverse engineering attempts.
- **Security through Obscurity** – Although controversial, it can be argued that program obfuscation makes it more difficult for attackers to detect exploitable program flaws such as buffer overflows.

As a result, new program monitoring techniques must be explored to improve system security and to establish user trust for encrypted programs. These encrypted programs may be the favored target for attackers as they find these targets interesting. Moreover, if they do manage to find vulnerabilities, it is possible for them to hide their activities due to the encryption. Some possible motives for such attacks include:

- Information Harvesting – An attacker may wish to obtain personal information about the user for identity theft. This may include banking information, social security numbers, classified documents, and so on. If the program is encrypted, the attacker's actions may go unnoticed.
- Resource Gathering – An attacker may want to use the resources of the system for illegal activity such as in denial of service attacks or for transmitting spam. Once the encrypted software is compromised, the attacker can regain access to the system by installing an undetectable backdoor into the program.
- Program Distribution – The attacker can use the encrypted program to infect other software on the system or other computers within the network. If an encrypted software is brought into a company by an unwitting user, it can now access computers within the network and bypass company security measures such as firewalls.

Due to the difficulty of monitoring encrypted programs, we believe that new detecting methods must be investigated. Such methods must obey several important properties to be successful:

- Program Invariability – Programs cannot be modified in any way or form as encrypted programs often have ways of detecting and rejecting inserted code. Also, it is uncertain how encrypted program will change its behavior or nature when modified.
- Faultless Execution – The program must execute properly while being monitored. Program execution under monitoring must behave exactly the same if the program is not being monitored.
- System Integrity – The integrity of the program monitoring system must remain intact and cannot be affected by the program being monitored. Often, encrypted programs may contain code to defeat certain monitoring techniques. Such

attempts must be detectable in order to know if the system integrity has been compromised.

In this work, we focus on malicious behavior detection. In particular, we explore the file system integrity and execution properties through the use of system call monitoring. The section below is a list of terminology used in this research followed by a short system architecture overview and the role of system calls. Finally, we conclude this chapter with a discussion of related work.

2.1 Terms

Black-Box Program – Any program that is obfuscated and can only be viewed in terms of its input and output characteristics can be categorized as a black-box program. Black-box programs may also have many countermeasures to defeat any attempts to obtain program instructions or program behavior information.

Trust – Trust, as defined by Banerjee in [9] says that a software system is trustworthy if it encompasses security, reliability, availability, fault-tolerance, and survivability. In [8], Trust is defined as the level at which a user believes a computer system executes as specified and does nothing else. For this research, we modify these two terms and define trust as the level at which a user believes a computer *program* executes as specified and does nothing else while providing convincing security and reliability.figure

Compromised Program – When an attacker has made some form of unauthorized access to a program, the program is said to be compromised. If this compromise allows the attacker to implement a means into the system, then the attacker can gain root access to the entire system. Other compromises may deal with information security; the attacker may simply use the compromised program to transmit sensitive information to an outside source.

Instrumentation – Instrumentation is when direct modifications are performed to the system to achieve a specific purpose. In this case, we instrument the kernel to enable system call monitoring. Although there are many methods of providing the necessary system call monitoring information as discussed in section 2.3, we argue that instrumentation is necessary to defeat the anti reverse-engineering techniques discussed in section 4.

On Demand System Call Monitoring – On demand system call monitoring is a term for the ability to enable and disable system call monitoring as required. The reason why it is undesirable to enable system call monitoring at all times is due to its overhead when dealing with particular system calls, such as those for reading from a file. With on demand system call monitoring, the user can enable this feature when executing a black-box program.

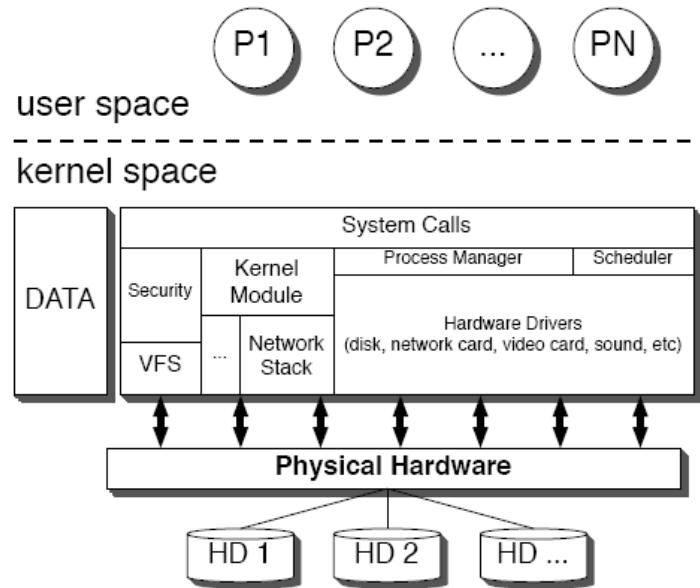


Figure 1. Computer system model {Source: [46]}

2.2 Operating System Model

Our model of the operating system is shown in Figure 1. All processes must remain in user space and are isolated from each other and the kernel level resources,

while any object inside the kernel space has direct access to all other resources as well as all user level processes. As shown in this figure, in order to interact with the hardware and access kernel level resources, all processes must submit their request through the system call layer. Due to this characteristic, it is possible for a monitoring program to examine the input and outputs as seen at the system call level in order to detect all changes made on the system by a process. This, however, does not guarantee that once system wide access has been granted to a particular process, additional modifications can be detected.

2.3 Related Work

Program-monitoring and intrusion detection techniques have been studied for many years. There are many methods and architectures for monitors and IDS systems. This section presents a survey of the current state of program monitoring, in particular, system call monitoring. Monitoring system calls as opposed to monitoring machine instructions have several benefits, including less data to analyze and less program execution performance penalty as compared to instruction level monitoring since system calls occur less frequently. Finally, it is possible to filter on function call names or arguments.

Forrest, Hofmeyr, Somayaji, and Longstaff in [15] created an “artificial immune system” for the UNIX operating system that may lead to automatic intrusion detection. The assumption made in this paper is that short sequences of system calls can be used to build a profile for the operating system to distinguish between normal and abnormal behavior. The method developed breaks down sequences of system calls into a database that can be easily accessed in linear time for comparisons. For example, the system call sequence of `execve`, `access`, `open`, `open`, `access`, `mmap`, `read`, `close` is stored in a table format shown in Figure 2. When comparing this table with the sequence of `execve`, `open`, `open`, `access`, `open`, `open`, `access`, `close`, the following mismatch errors are generated:

- Execve should not be followed by open at position 1
- Execve should not be followed by access at position 3
- Open should not be followed by open at position 3
- Open should not be followed by open at position 2
- Open should not be followed by open at position 3
- Open should not be followed by close at position 3.
- Open should not be followed by close at position 2.
- Access should not be followed by close at position 1.

call	position 1	position 2	position 3
execve	access	open	open
access	open	open	access
	mmap	read	close
open	open	access	mmap
	access	mmap	read
mmap	read	close	
read	close		
close			

Figure 2. Breakdown of sequences of system calls

Forrest et al show that after approximately 3000 system calls, virtually no new patterns are encountered under normal conditions for their example program (sendmail). Their research also demonstrated that short sequences of system calls can be used to define a stable signature for the detection of some common anomalous behaviors. Since this architecture has relatively modest computation and storage requirements, it can be implemented on an online system in which the kernel checks each system call made by root processes.

Hofmeyr, Forrest, and Somayaji continued their previous work by introducing a method of intrusion detection based on patterns in sequences of system calls via anomaly detection [13]. The idea behind their research is to gather data on a running program to build a behavior profile for the executable. The data stored in the profile is simply a

database containing all unique sequences of system calls for a given length k . Each program under observation is associated with a unique database. Once a stable database is constructed, it can be used to monitor the ongoing behavior of the process. The intrusion detection system reports the number of mismatches, percentage of mismatches, and normalized anomaly signal generated by program execution. With this method of detection, Hofmeyr et al successfully detected several classes of abnormal behavior including: intrusions in the UNIX programs sendmail, lpr, and ftpd, failed intrusion attempts on sendmail, and error conditions in sendmail.

Christina Warrender, Stephanie Forrest, and Barak Pearlmutter in [16] further examine Hofmeyr's work in the previous work through additional performance analysis using a much larger and complete dataset. The architecture described here is tested with three attacks: A denial-of-service attack for tying up network connection resources that contains a slightly modified daemon process but same startup and child processes, a second denial-of-service attack designed to tie up all system memory during intrusion monitoring, and a final data set with normal data designed for false positives analysis. Warrender et al used four different methods for modeling the normal behavior of the data sets: sequence time-delay embedding (STIDE), STIDE with frequency threshold (T-STIDE), repeated incremental pruning to produce error (RIPPER), and Hidden Markov Models (HMMs). This study shows that the amount of false negatives decreases as the window threshold increases, but the amount of false positives also increases.

Jae-Kook Lee and Hyong-Shik Kim proposed a Log-Based Intrusion Recovery Module (LBIRM) that utilizes system calls dealing with the file system to recover damaged or lost files due to intrusions [17]. The system proposed is a method of keeping the contents of designated files as a chain of logs. Whenever a designated file is modified, a log will be automatically created to keep track of the changes. The log file contains modifications from several prior changes in time thus giving the file retriever the ability to rollback to specific instances of the designated files at different periods in the

past. Due to the increasing size of the log file, a purge cycle is performed on the log after a certain amount of time has passed to keep the log file size manageable. This system is achieved through the replacement of several system calls dealing with the file system: `sys_open`, `sys_write`, and `sys_close`. At the time `sys_open` is called, a new entry in the log is created and old logs lasting beyond a specific time are purged. The `sys_write` function keeps a copy of the modifications performed to the file. The `sys_close` function closes the file descriptor and the log descriptor. This paper also performs some rudimentary overhead analysis showing the extra costs of the new system calls typically result in an overhead of 53-65% as compared to the regular system calls.

Sebek [18] is a data-capturing tool designed to record malicious activities on Honeynets [19], and it allows researchers to bypass the need to break session encryption during an attack by circumventing it altogether. Sebek resides entirely in kernel space and records data accessed by users through monitoring system calls. Some of the information recorded includes keystrokes, files transferred via SCP, passwords used to log into the remote system, and all `sys_read` data. All `sys_read` data is captured by replacing the stock `sys_read` function with a modified version. The new `sys_read` simply calls the original function after transmitting a copy of the contents to the logging server. Due to the nature of the Honeynet, it is safe to assume that all traffic to the honeypot computers are considered suspect and therefore should be logged by Sebek. The Sebek architecture has two components, clients and servers. Clients are installed on honeypot machines and transmit all activities to the Sebek server for collection via hidden packets. This separation of clients and servers is done to mask the logging activities from attackers while increasing the difficulty for attackers to falsify the logging data. The goal of the captured data is to allow researchers to reconstruct the events on the honeypot during the intrusion after it has occurred. With this information, it is possible to determine when the attack occurred, how the attackers broke into the system, and what modifications the attackers performed on the system.

Giffin et al in [22] introduced the concept of environment sensitive intrusion detection. In this work, event sequences that cannot be correctly generated in the current environment are restricted. This is achieved by comparing system-call arguments with shared objects of the target executable. Giffin et al demonstrated that their technique performed significantly better than prior approaches against a variety of attacks.

A collection of common system call monitoring utilities is provided by most modern operating systems. These utilities include trace, strace, truss, ptrace, and others. This set of utilities, when given a specific program, will capture the trace of the system calls the target program performs, the signals it receives, and the machine faults that occur.

SystemTap [24], an open source project that is released under the GPL, provides an infrastructure to assist in the gathering of information about the running Linux system. SystemTap allows users to develop tapset script libraries for a variety of uses, such as placing a tap on a specified system call. This is achieved through adding kernel modules to provide debug information in the kernel. The advantage of SystemTap is that users no longer have to recompile and reinstall kernel files in order to monitor a particular system call. Developers can use this utility to debug their work more easily.

NSA originally developed Security-Enhanced Linux (SELinux) [25] [26] [28] [29] as a research prototype of the Linux kernel with enhanced security features enforcing mandatory access control. In SELinux, the concept of root access is ignored as all programs are given the minimum privileges required for execution. This will deter many forms of attack as well as rendering some common Linux exploits ineffective.

While SELinux supports a variety of access control policy models, its main focus has been an extended type enforcement (TE) model. Traditional TE models have subject types, object types, and access control, which are represented by the permissions of the subject types to the object types. In SELinux, the distinction between subject and object types has been removed, so the only types are object types that may also act as subject

types [25]. The “allow” statement grants permission for a certain type to perform operations on another type. Since any element of the permission relationship can be expressed with this statement, it is possible for the expression to contain the lowest number of privileged rights.

One major drawback with SELinux is the difficulty in configuring the system securely for individual administrators. Although, SELinux comes with a default example policy, it is still very difficult to configure properly. The default policy does not define a secure system, but is intended as input for the development of a custom policy that suits the goals of individual users. Moreover, since the application policies are specialized to the environment in which they were developed, interactions between policies of multiple applications may lead to vulnerabilities. Thus, combining policies that have been proven secure may introduce security loopholes. The task of customization is further complicated by the sheer size of the example policy (over 50,000 statements in the example policy for Linux 2.4.19), and this represents over 700 subject types and 100,000 permission assignments [25]. This size and complexity make it difficult for typical administrators to customize to insure protection of their trusted computing bases and to satisfy their security needs.

Trent Jaeger, Reiner Sailer, and Xiaolan Zhang presented an approach for analyzing the integrity protection in the SELinux example policy [25]. Using their analysis tool, Gokyo, Jaeger et al examined the SELinux policies with their defined integrity goals to identify conflicts and to estimate the resolutions. Their ultimate goal was to define a minimal trusted computing base (TCB) for SELinux that includes 30 “subject” types. In their research, Jaeger et al used the Clark-Wilson integrity model and attempted to capture the notion that vulnerability exists when a higher integrity process cannot handle input from lower integrity processes. In the Clark-Wilson model, constrained data items (CDIs) were defined as high-integrity data that was processed only by certified transformation procedures (TPs), which can also process unconstrained data

items (UDIs). Integrity verification procedures (IVPs) can be used to verify the integrity of CDIs at particular times. The actual model can be represented by the following set of rules:

- Each TP operates on a particular list of CDIs and CDIs are only manipulated by a TP.
- The system must contain a list of subjects, TPs, and the CDIs those TPs may reference, and only those references are permitted.
- The system must authenticate the identity of each user who attempts to execute a TP.
- The list of TPs and IVPs can only be changed by a subject permitted to certify those TPs and IVPs.

In their results, Jaeger et al noticed several conflicts in the `dpkg_t`, `initrc_t`, `kernel_t`, `local_login_t`, `mount_t`, `sysadm_t`, and `sshd_t` processes. Most of these conflicts stem from differences between trusted subject types and the pseudo-terminals that they share with user processes, and the permission assignment differences. However, the minimal TCB containing only 30 types that meet the Clark-Wilson integrity requirements have been found and this may lead to more customizable and user friendly SELinux policies.

In [26], Chad Hanson attempted to improve the Multi-Level Security (MLS) model within SELinux. The existing MLS policy maps permissions of a security class to a set of MLS base permissions. In the proposed system, a flexible mechanism that allows the ability to grant policy overrides on a granular level will be implemented.

2.4 System Calls

Currently, there are 325 system calls in the Linux 2.6.23 kernel `syscall_table.S` file (see appendix A). Additionally, there are also other system calls not listed in this file such as those for network access which are declared in `/net/socket.c` that are accessed through a single gateway system call. Due to these large numbers of system calls, user mode processes must pass a parameter known as the system call number to identify the required system call. Figure 3 shows the typical process when invoking a system call. Many of these system calls are either redundant or are a part of legacy code. For example, `sys_creat` is simply a wrapper function for `sys_open` with the `O_CREAT` flag enabled.

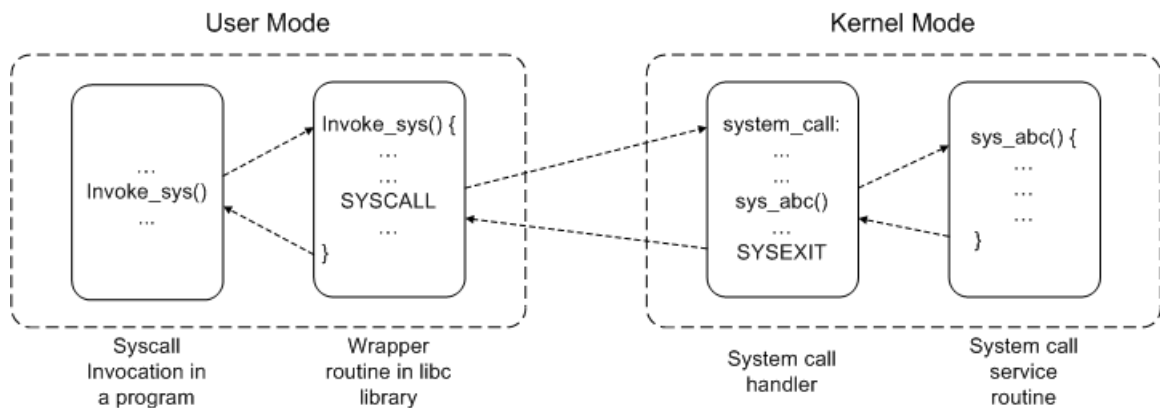


Figure 3. Invoking System Calls

The parameters passed by the system calls are different than that of normal C functions. For normal functions, the parameters are usually passed by writing their values to the active program stack, be it the User Mode stack or the Kernel Mode stack. For system calls, the parameters are written to the registers before issuing the system call, and then the kernel copies the parameters from the registers to the Kernel Mode stack before invoking the system call routine.

There are many system calls pertaining with the same type of functionality, and thus can be grouped together under one general category. Process system calls handle the behavior and invocation of processes. File-system system calls such as `open` handles the

requests for reading and writing to files by all processes. Network system calls such as `accept` performs the function of connecting a socket to a client.

2.4.1 Process System Calls

In Linux, processes are created using one or more of the following system calls.

- `sys_execve` – Reads the filename of the new program from the `ebx` register and begins execution of the process.
- `sys_clone` – Lightweight processes are created with this system call. The child process can share a number of resources with the parent process, such as memory descriptors or page tables.
- `sys_fork` – A process created with `sys_fork` does not share any resources with the parent process.
- `sys_vfork` – A process created with `sys_vfork` creates a child process that shares the memory address space with the parent process. However, the parent's execution will be blocked until the child exits or executes a new program to avoid the parent from overwriting data required by the child.
- `do_fork` – The `do_fork` function is not a system call, but it handles the `clone`, `fork`, and `vfork` system calls. This is the function that allocates and returns the assigned PID of the child process.

When a user begins the execution of a program, such as typing in `/bin/ls` at the command shell, the command shell will fork a new process, which in turn invokes `sys_execve` with the parameter of “`bin/ls`.” The `sys_execve` function then proceeds to check the corresponding file, the executable format, and modifies the execution context of the current process to begin execution of the new code as soon as `sys_execve` terminates.

2.4.2 File-system System Calls

Files can only be accessed by processes when they are “opened”. To open a file, the process invokes the `sys_open` system call, which returns a file handler. In Linux, all files are handled with an identifier called a file descriptor. The file descriptor represents an interaction between a process and an opened file. The same file object may be identified by several file descriptors in the same process.

To access open files, the program must use a combination of the `sys_read`, `sys_write`, and `sys_lseek` functions. When a file is opened, the kernel sets the file pointer to the first byte in the file. A process must use the `lseek` system call to move the file pointer to the specified location. Once there, the process can begin either reading or writing to the file with the `read` and `write` system calls. Finally, when a process is done accessing a file, it must invoke `sys_close` to release the open file object. When a process terminates, all remaining opened files are automatically released.

A list of file-system system calls:

<code>sys_access</code>	<code>sys_chdir</code>	<code>sys_chmod</code>	<code>sys_chown</code>	<code>sys_chroot</code>
<code>sys_close</code>	<code>sys_creat</code>	<code>sys_dup</code>	<code>sys_dup2</code>	<code>sys_dupfd</code>
<code>sys_fchdir</code>	<code>sys_fchmod</code>	<code>sys_fchown</code>	<code>sys_fdatasync</code>	<code>sys_flock</code>
<code>sys_fstat</code>	<code>sys_fsync</code>	<code>sys_ftruncate</code>	<code>sys_lchown</code>	<code>sys_link</code>
<code>sys_llseek</code>	<code>sys_lseek</code>	<code>sys_lstat</code>	<code>sys_mkdir</code>	<code>sys_mknod</code>
<code>sys_mount</code>	<code>sys_msync</code>	<code>sys_open</code>	<code>sys_pread</code>	<code>sys_pwrite</code>
<code>sys_read</code>	<code>sys_readlink</code>	<code>sys_rename</code>	<code>sys_rmdir</code>	<code>sys_stat</code>
<code>sys_symlink</code>	<code>sys_truncate</code>	<code>sys_umount</code>	<code>sys_unlink</code>	<code>sys_ustat</code>
<code>sys_write</code>				

2.4.3 Network System Calls

The system calls pertaining to the network are defined in `/net/socket.c`. The actual entries of these networking system calls in the system call table, however, are nonexistent. In Linux, all networking related system calls are accessed through a gateway system call named `sys_socketcall`. `sys_socketcall` takes in a parameter for the type of networking call required and calls the appropriate sub-function. Appendix A lists the networking system calls under the gateway system call `sys_socketcall` (#104).

2.4.4 Other System Calls

Due to the large number of system calls existing in Linux, we choose to target the system calls for process control, file access, and networking to demonstrate a proof of concept for our monitoring and trust building methodology. The remaining system calls listed in appendix A that were not discussed can also be tapped for additional sources of information in black-box programs and will be left for future work.

CHAPTER 3

SURVEY OF MONITORING AND COUNTERMEASURE TECHNIQUES

3.1 Program Monitoring

Research into program analysis methods has been started since the 1960s [10]. The two major methods of analysis are static and dynamic. Static analysis considers program activities based on the program instruction code without consideration of its execution. On the other hand, dynamic analysis is based on explicit observations of execution histories and behavior. Early on program monitoring techniques focused on primarily execution monitoring. This was achieved through user-controlled break points, tracing, observing and setting values of variables, and local modification of the program (patching). Most of the early execution monitors were developed for assembly languages, but were soon applied to higher level programming languages.

3.1.1 Debugging

Most modern microprocessors now contain features in their CPU design to assist in debugging such as having hardware support that allows users to step through the program one line at a time with the trap flag [47]. Debugging programs is a widely used approach, providing many insights to aid in error detection and program optimization. A common debugger allows programmers to obtain information such as register values and memory contents. Debuggers also help users in cracking software by allowing them to execute or skip over certain subroutines such as those dealing with protecting the software. Some of the earliest debugging system techniques involved using a preprocessor to convert debugging statements into source statements. These source statements were then compiled together with the program to be monitored.

The evolutions of debuggers have migrated away from being language specific as were the debuggers of the sixties. Today, one of the most widely used debugging utility is GDB: The GNU Project Debugger [30] (created in 1986). GDB allows users to debug programs written in many languages such as Ada, C, C++, Pascal, etc. Also, GDB can even work remotely and debug programs executing on other machines.

Debuggers perform their task using the information generated by the compiler to associate the source code with machine addresses where the program is executing. This allows the debugger to see the corresponding source code. This is achieved by inserting breakpoint instructions into special positions of the code or by using the compiler to generate stop instructions that will make programs wait for instructions from a supervisor program. The prior solution is successful when the debugger has access to the code stream and can modify it. The second solution is used when the code is in read only memory and modifications are not possible.

3.1.2 Injection

Code injection is a technique to insert code into a program by taking advantage of unenforced and unchecked assumptions for program inputs [31]. This is often done through the use of a buffer overflow on the unchecked input. Malicious users often apply this method of attack to gain unauthorized root access or to insert malware into the compromised system (such as buffer overflow). Code injection has other uses as well. For example a user may wish to modify the behavior of programs or simply to insert additional software breakpoints for debugging purposes. There are several reasons why this approach is used, such as when modifying the software is either impossible or prohibitively costly.

There are several types of code injection techniques. The first of which is categorized as script injection to take advantage of unchecked entries in web applications.

In this type of injection, additional code is added into input buffers and then is executed

by other users as part of the background code. Figure 4, we see that a web server has a standard guest book script that accepts small messages from users. If this input is unchecked, then it is possible for someone to embed some code into this guest book with a message shaped like standard scripting language to hijack the website. Once this is done, any subsequent visitors to this page can be targeted by the injected code.

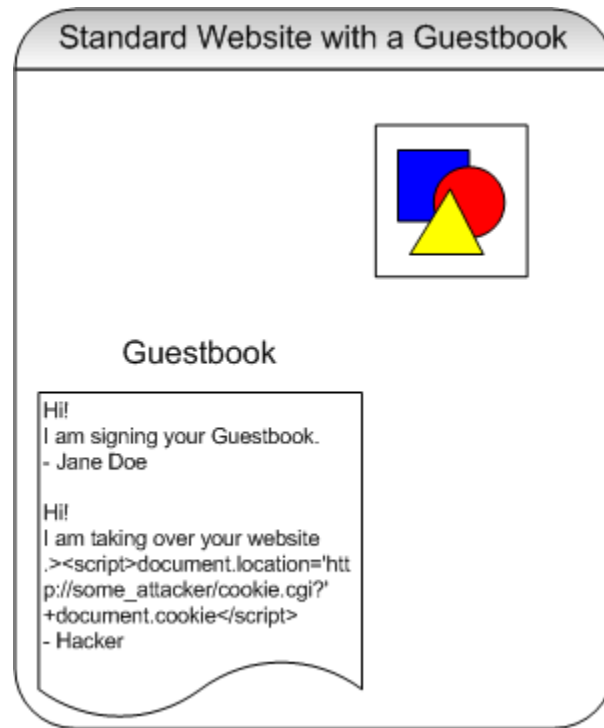


Figure 4. Simple code injection example

The second type of code injection is from stack smashing. Figure 5a shows the normal operation of a function call and the stack. The first method of stack smashing is the standard stack based buffer overflow as shown in Figure 5b. In this scenario, the buffer is filled with data until the return address is replaced to point to the injected code segment. In Figure 5c, only the saved frame pointer is overwritten, and thus when the current calling function is popped off the stack, it will follow the injected return address to the code segment. If the attacker cannot overwrite the return address or frame pointer due to some countermeasures, they can use a method called indirect pointer overwriting

as shown in 5d. In this case, the overflow is used to overwrite the local variable that is pointing to value1 and redirect the pointer to the return address instead. Next, the pointer is then de-referenced and the value it points to is changed at some point in function 1 to an attacker-specified value. This indirect pointer overwriting method allows the attacker to overwrite any memory location, and any pointer to code that will later be executed could be used for an attacker to overwrite.

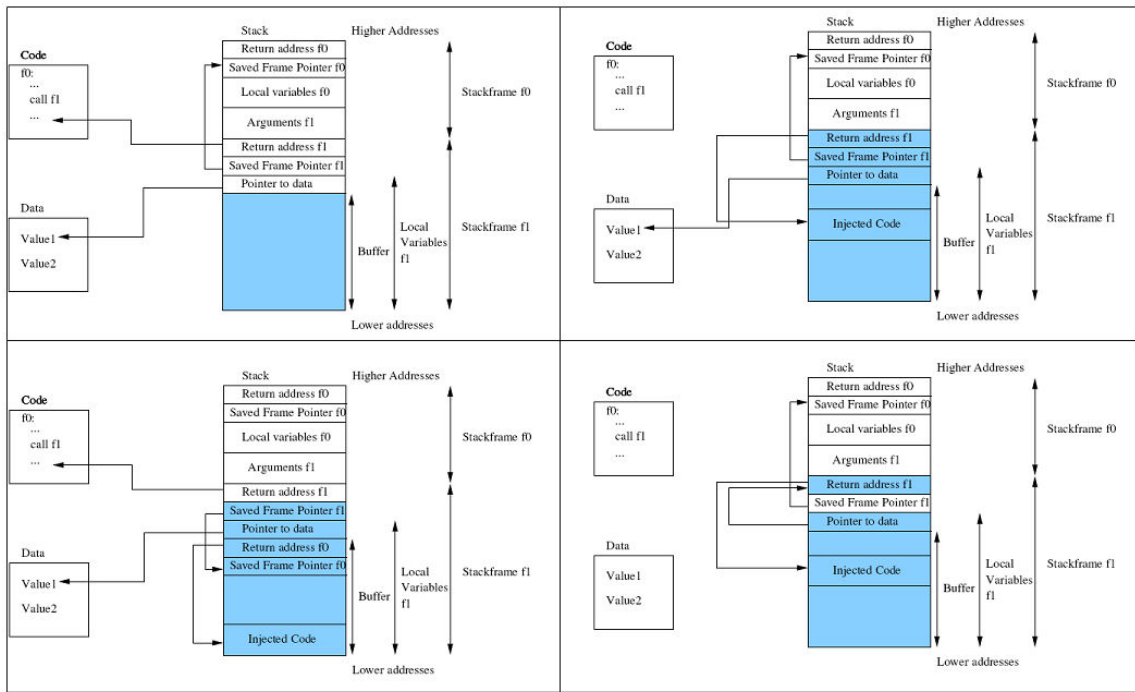


Figure 5. Stack smashing. a(top left) Standard stack execution, b(top right) Basic stack smashing, c(bottom left) Frame pointer overwrite, d(bottom right) Indirect pointer overwriting. {Source: [31]}

3.1.3 Memory Scanning

A memory scan is performed by recording the state of the working memory of a computer program at specified time intervals. In effect, this is taking a snapshot of the program during execution and dumping it to a disk. The contents of a memory dump contain the information from the dumped regions of the address space of the process. Users can then examine these contents offline to minimize the interference with the program execution. Finally, there are many downloadable tools such as objdump that can

be used to analyze memory dumps. Moreover, since all instructions must be executed as clear text, it is possible to obtain the current executing code that is residing in memory.

3.1.4 Tracing

Program tracing can be performed with several methods, which include system call tracing, library call tracing, and execution path tracing. System call tracing, with tools such as `strace` in Linux, is a technique where all system calls of a process, return values of those system calls, as well as the parameters passed to those system calls are intercepted and recorded. System call tracing is a valuable tool for users to diagnose and solve problems where the source code is not readily available. A great deal of information about the program can be obtained by examining its interactions with the kernel. This information can be used to perform sanity checks and to detect race conditions. Execution path tracing takes place on several levels such as conditional branch tracing and instruction flow tracing. These tracing methods allow users to construct a standard behavior pattern for the executable containing information such as the frequency of the branches taken or the number of instructions executed in a single execution. Finally, library call tracing, such as `ltrace`, gives details about the library files in use by the target program. All of these tracing techniques, however, are single process oriented and are invoked statically [48] [49].

3.1.5 Decompiling

The translation of an executable into assembly or higher-level languages is called decompiling. The success of this process depends on the amount of information present in the original executable as well as the cleverness of the analysis performed on the program. The decompilation of a program is often broken up into several stages: loader, disassembly, program and type analysis, structuring, and finally code generation [50]. In the loader phase, the basic architecture of the program and main function are examined.

The objective of this phase is to locate the program entry point and discover the architecture the program was compiled on. The disassembly process translates the machine code into assembly language. During program and type analysis, the expressions and the type of value in the program are reconstructed. The structuring process rebuilds the loops along with the if/then/else commands of the program. Finally, the code generation phase attempts to recreate the high-level code. Figure 6 shows a sample piece of code during the decompiling process.

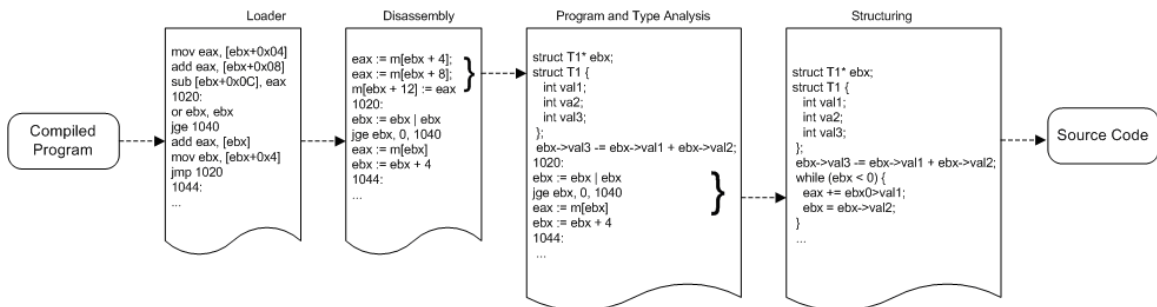


Figure 6. The decompilation of code

3.2 Anti Monitoring Techniques

Encrypted executables employ a wide variety of techniques that deter code reverse engineering efforts. In particular, good program encryptors must overcome the methods described in the previous section. The following two dilemmas are faced by all program encryptor creators:

- To be able to execute, the code of the program must eventually be decrypted at some point. Thus, the encrypted executable must have the ability to access the encryption keys for the images. Because of this, encryption is simply a way to make it extremely difficult for attackers to retrieve those keys.
- The attackers have complete control of the operating environments. The goal of writing a program is to be able to sell and distribute the software to users. However, this also allows attackers to obtain copies of the software and run them on customized reverse engineering environments such as virtual machines and

modified kernels. There is no way program encryption can completely block reverse engineering attempts. However, good program encryption can deter such attacks for a long time.

This section will provide the reader with a survey of the common methods employed in encrypting executables.

3.2.1 Encryption

The first and most obvious method is the idea of encrypting the binary to defeat static disassembly. There are many executable encryptors available for this task, such as Shiva and UPX. To increase the security of the program, it is also possible to employ multiple layers of encryption. Thus, in order to decrypt key portions of the program, one must peel back the layers of encryption like peeling the layers of an onion. A program for encrypting ELF-binaries is actually a combination of two programs: the encryptor, which performs the execution process and wraps up the target executable, and the decryptor, which is a statically-linked executable that performs the decryption and handles runtime processing [33]. The decryptor is often embedded into the encrypted executable and is completely self-contained, and it does not rely on linking to any library files.

Figure 7 demonstrates how multiple layers of encryption can be employed by a program encryptor. The highest layer is most often the obfuscation layer. This layer is intended to be easy to implement, can avoid simple static analysis. See section 3.2.5 for obfuscation techniques.

Underneath the obfuscation layer frequently comes the password layer, which employs strong encryption techniques such as 256-bit AES and wraps the executable using this method. The key used for this encryption will be the SHA1 hash of the password used to encrypt the program itself, and thus the encryption is only as strong as the password. The crypt block layer is the third and innermost layer, where the binary is

broken up into many code segments called blocks. The encryption used on the block contents is typically very strong and unpublicized to further increase the difficulty of analysis. The code to generate the keys for this layer is pseudo-random and is never stored in plain text inside the program at any given point. Typically the decryptor performs this by interacting with the dynamic linker resident on the system. This is done by mapping the dynamic linker and then having the decryptor regain control after the linker is done. Dynamically linking permits the program to load and unload routines at runtime, which in turn allows those routines to be encrypted and only decrypted on demand.

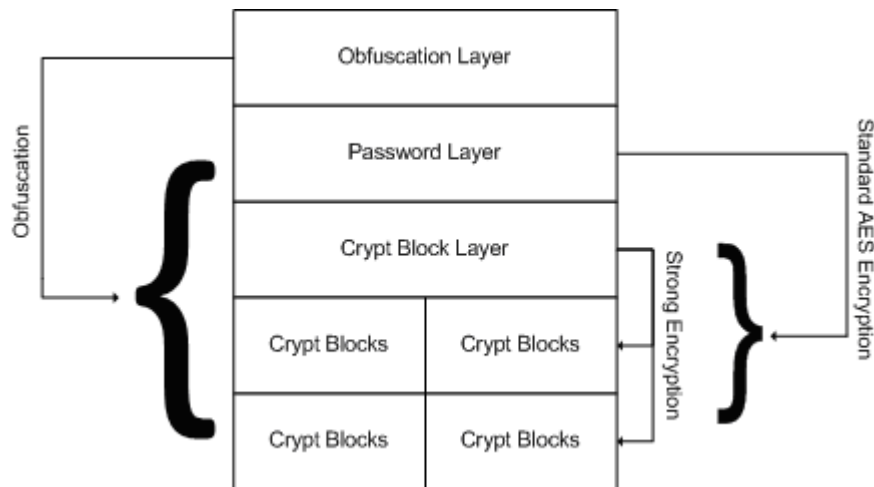


Figure 7. Multiple layers of encryption

3.2.2 Integrity Checks

Code integrity checkers can be employed to prevent attacks from modifying instructions such as code injection. Although the use of checksums will slow down program execution, they provide the benefit of making it difficult for users to insert code into the executable without first disabling the checksums. There are also several methods that will improve the difficulty of disabling the checksums. To increase the difficulty of detection, it is possible to randomize the checksum operators and to make them polymorphic. To enhance the difficulty of removing the checksums, the values computed

by the checksums can be used in a non-trivial task, such as using the result to calculate the address of the next code segment.

3.2.3 Anti Debugging

Anti-debugging and emulator detection techniques are employed to prevent the examination of the source code through debugging and emulation tools. There are debugging detection methods such as using tick timers to detect abnormally long periods of execution time and exploiting certain properties of the operating system to prevent their function. For example, the Shiva encryption program takes advantage of the Linux property that no program can be traced by multiple sources at the same time through two forked processes tracing each other. This method can also be used to detect if certain debugging programs, such as gdm, is currently using ptrace on the executable. If a program tries to invoke ptrace on itself and fails, then it knows that it is currently under analysis by a debugger using ptrace.

```
start:
    jmp    label + 1
label:  DB    0xe9
        mov    eax, 0xf001
-----
Disassembly of code:
# objdump -M intel -d anti02
Anti02:      file format elf32-i386
08048080 <start>:
8048080:    e9 01 00 00 00      jmp     8048086 <label+0x1>

08048085 <label>:
8048085:    e9 b8 01 f0 00      jmp     0f48242 <__bss_start+0xeff1b6>
```

Figure 8. Hidden mov instruction

A second method to defeat debugging is by jumping into the middle of an instruction to defeat linear sweep. Figure 8 gives a short sample of code that hides an instruction from objdump. In this example, we show an instruction which assembles into more than 1 byte where objdump will not follow the jump command and will just disassemble the program linearly from start to end. The result is that objdump ignored the jump destination and disassembled the instruction directly following the first jmp. As shown in this figure, the 0xe9 byte was also displayed as a jmp instruction and thus the move operation became hidden.

Finally, it is also possible to detect breakpoints generated by common debuggers such as gdb. Gdb sets breakpoints by replacing the byte at the address to break with an int 3 Opcode, which is 0xcc [34]. Thus, it is easy for a program to check addresses for 0xcc as shown in figure 9.

```
// -- antibreakpoint.c --
void foo()
{
    printf("Hello\n");
}

int main()
{
    if ((*volatile unsigned *)((unsigned)foo) & 0xff) == 0xcc {
        printf("BREAKPOINT\n");
        exit(1);
    }
    foo();
}
// -- EOF --

# gdb ./x
GNU gdb 6.0-2
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-linux-gnu"...Using host libthread_db library "/
lib/tls/libthread_db.so.1".

gdb> bp foo
Breakpoint 1 at 0x804838c
gdb> run
BREAKPOINT

Program exited with code 01.
```

Figure 9. Breakpoint detection {Source: [34]}

3.2.4 Virtual Machine Detection

Virtual machine emulators such as VMware can be detected by examining the hardware that it is supposed to emulate, such as having VMware Inc [VMware SVGA II] PCI Display Adapter for the video card [20]. VMware also has an interface used to configure VMware during runtime and can be accessed with several special assembly commands. Unfortunately, encryption programs can easily add in code to use these commands to determine whether if the machine they are attacking is running VMware as well.

3.2.5 Obfuscation

The goal of code obfuscation is two fold, to protect the code from being reverse engineered, and to make it more difficult for someone to create a generic unwrapper for this software. This is achieved by rendering the code unreadable to users and to defeat common disassembly techniques [51]. Although code obfuscation can slow down code study and avoid direct code stealing, it also slows down the application and will increase the software size.

There are several common techniques used for code obfuscation:

- Dynamically computed calls – The jmp and call values are calculated at run time to make it difficult to follow the code statically. Functions are not invoked directly but with address pointers that are calculated using other means, such as subtracting a constant from the checksum of the previous code block.
- Opaque predicates – Although the result of such an operation is known ahead of time logically, branch predictors will begin code execution of the sub branches while the computation of the statement is being processed. If a computationally intensive statement is used, then it further increases the amount of dummy code that requires analysis. Here in the following example (figure 10), the cosine of

any value cannot be greater than 1 and thus the code inside the conditional statement should never affect the program execution.

- Execution flow disruption – False exceptions can be raised by the program and shaped in such a way that the error handler can detect the characteristic. If it is a fake error, the error handler will perform decoy tasks such as tweaking memory addresses and registers before returning to the calling code.
- Junk insertion – Junk bytes can be inserted into code to cause disassembly errors. These junk bytes must satisfy two qualities: they must be partial instructions, and must be inserted in such a way that they are unreachable at runtime [51]. To achieve this goal, code is broken up into block segments that end with unconditional control transfers, and junk bytes are inserted in between these blocks.

```
if ( cos(sin(x)) > 1 ) {  
  // perform dummy operations }  
  // Continue program execution
```

Figure 10. Opaque predicate example

3.2.6 Kernel Module Detection

Often, encrypted executables employ various tasks to detect the existence of recording tools and replaced system calls as others attempt to reverse engineer the program. In [20] part 2, Thorsten Holz and Frederic Raynal discussed methods attackers could use to detect Sebek on honeypot machines. The Linux version of Sebek is a loadable kernel module that has similar properties as a rootkit. This is achieved through the unlinking of the last module in the module list. This unlinking allows Sebek to remain hidden from programs such as lsmod, but Sebek still has an easily detectible module header structure resident in memory. In [35], Dornseif et al mentions that since Sebek provides its own sys_read function, it has to modify the system call table to update

the pointer from the original `sys_read` function to the modified version. This will cause the addresses of the `sys_read` and `sys_write` functions to be very different, which allow hackers to detect that one of these functions may have been modified at some point. Unmodified read and writes tend to be near each other in the memory address range. Dornseif et al also found that it is possible to avoid logging commands into Sebek by using the `mmap` system call. Rather than using `sys_read` to open files, a hacker can employ `mmap` to map files directly into memory and bypass the logging capabilities of Sebek completely. Finally, it is possible for attackers to detect the traffic Sebek generates by performing an operation that makes heavy usage of `sys_read`, such as doing a `dd` of `/dev/zero` and writing it to `/dev/null`. The amount of traffic generated by Sebek introduces a noticeable delay of over 4800 milliseconds in packet round trip time. The informed developer for the encrypted software will take advantage of this knowledge and build in detections for such events and even attempt to shut down such monitoring methods.

3.3 Examples of Encrypted Programs

Skype is an overlay peer-to-peer (P2P) voice over IP (VoIP) program that has come under scrutiny from many researchers [12] [14] due to the intricate security and encryption techniques employed in the executable file. There are two types of nodes in the Skype network, basic nodes and super nodes. Any node having a public IP address and meeting sufficient system requirements has the potential to become a super node. Baset and Schulzrinne performed a study in 2004 [12] to examine Skype in greater detail. Their goal was to determine key Skype functions such as login, NAT and firewall traversal, call establishment, media transfer, codecs, and conferencing under different network setups. In their study, Baset and Schulzrinne determined that Skype was able to pierce through firewalls by establishing TCP links with super nodes on the `http` and `https` ports.

At the Blackhat Europe conference in 2006, Philippe Biondi and Fabrice Desclaux presented the results of their attempts at penetrating the encryption methods employed in Skype [14]. The usage of Skype is a concern for many network security administrators at major corporations because of its secretive nature. The very first technique Skype employs is code encryption. Rather than having the entire Skype program decrypted in memory at any given time during execution, Skype applies techniques to hide portions of its code until the instructions need to be executed. Moreover, during the decryption process, Skype erases previously decrypted code along with overwriting portions of its import table. Thus, at any instance of execution, only a small portion of the total code is unencrypted and it is also unclear which portion of the program this section of code came from. This anti-dumping technique defeats the possibility of observing instructions during the runtime of Skype.

There are also intensive code integrity checks in Skype. From [14], we see that overall, there were nearly 300 checksums embedded in the source code. These checksums were encrypted and the checksum operators were randomized. Finally, the values computed by the checksums are used by Skype to perform a non-trivial task, such as determining the pointer for the next code segment. These checksums make it impossible to insert software breakpoints into the program, which is one method for program monitoring. Moreover, Skype employs several tick timers that attempt to detect abnormally long execution periods, once the tick timers reach a certain count, the program will randomize the registers and jump to a randomized page.

Building a program profile for Skype is also a difficult task due to the lack of behavior patterns. In fact, the developers of Skype intentionally obfuscated the code by inserting randomized behavior. For instance, there are many dynamically calculated opaque predicates that make Skype difficult to follow statically. Other code randomizations can occur where Skype performs a series of calculations based on a randomly calculated condition and proceeds to disregard the result. Finally, Skype can

generate fake errors and raise fake exceptions. If a fake error is generated, the handler simply tweaks some memory addresses and registers and then returns to the calling code. Skype encrypts all digital content it transmits to and receives from the network. This makes it difficult to determine whether if the activity Skype performs is malicious or benign. Moreover, Skype often sends and receives information to portions of the network even when the user is not actively using the program to make a VoIP call. Finally it is impossible to determine if the latest update to Skype has installed a backdoor to the program due to its encrypted nature.

Shiva [6] [32] [33] is a free binary encryptor developed to encrypt ELF executables under Linux. Executables encrypted by Shiva continue to run as normal but are more difficult to reverse engineer. In their presentation [33], Clowes and Mehta presented the idea behind the creation of Shiva: to offer a method to prevent trivial reverse engineering of algorithms, to protect setuid programs with passwords, and to hide sensitive data and code segments in the programs. Their first goal is to render the `ptrace()` API ineffective and thus defending against common Linux tools that rely on that API such as `ltrace`, `strace`, `fenris`, and `gdb`. Next, Shiva attempts to embed many active detections of monitoring attempts into the target program. Third, Clowes and Mehta noticed that static analysis becomes much more difficult if the executable is encrypted on more than one level. This led to the development of layers similar to that of an onion where attackers must strip each layer of the encryption before attacking the next layer. Finally, adding in unpredictable behavior that differs from one executable to the other makes the encryption method less standard and thus makes it harder to create a generic unwrapper. Shiva contains two separate components, an encryption process that performs the encryption and wrapping of the executable, and a decryption process that performs decryption and handles runtime processing. The decryption process is embedded within the encryption portion of the program and is completely self-contained. Finally, the main executable thread will create a controller process (a clone) so that both

threads can trace each other, thus effectively locking other programs from performing tracing on the program due to the Linux property that no thread can simultaneously be traced by more than one other thread at any given time. The authors also mentioned future work on Shiva which includes dividing programs into blocks at compile-time so only one block at a time will be decrypted during execution.

In [1], Amit Singh brought attention to the use of encrypted binaries in the new operating system from Apple computers. In his article, Singh stated that the use of encrypted programs is one solution Apple came up with to make the process of running Mac OS X on none Apple hardware “non-trivial.” The Apple binaries are Mach-O files containing one or more AES-encrypted segments. A simple search that looks for Mach-O files with encrypted properties turns up the following list of programs: dock, finder, loginwindow, systemuiserver, mds, atsserver, and translate.

In summary, the field for program encryption is very active as innovative methods of encryption and anti-reverse-engineering methods are being explored daily. As the number of commonplace encrypted program grows, so will the necessity of monitoring those encrypted programs.

CHAPTER 4

ON DEMAND SYSTEM CALL MONITORING METHODOLOGY

There is currently no methodology to establish user trust in encrypted programs with program monitoring. Since statistical analysis does not work due to program encryption does not work, runtime monitoring methods must be examined. We believe that such a monitoring and trust developing methodology will be needed in the future as more and more executables become encrypted due to increasing security and anti-piracy concerns. Even today, encrypted executables such as Skype present a great security threat to companies [12]. Our initial interest in this research area came about as a result of several publications about the potential risk of Skype and the lack of a good solution for administrators to use in insuring the safety of their system. We believe that implementing on demand system calls for the execution of encrypted programs is the best type of mechanism to incorporate into the kernel, as all requests to access the underlying resources must go through this gateway. Below is a discussion of key concepts for system call monitoring, trust development, and detection methods using the gathered data to obtain a trust development architecture.

4.1 Architecture and Methodology Overview

As previously described, system call monitoring is not a new concept. However, the thrust of this approach is to provide a system wide safety net to catch potential misbehaving actions of encrypted software, unlike previous system call monitoring techniques and utilities that focus on system call sequences or on individual programs. We argue that this is necessary due to the intrinsic obscurity of these black-box programs since it is unknown what parts of the system are being accessed and what processes the black box program may spawn off or modify. Our approach of allowing users to execute encrypted software on their system can be broken down into several stages:

- Data Acquisition – We must modify the kernel to insert additional code at the system calls and to provide support to extract information out of the kernel space into user space for proper storage and handling.
- Monitoring and Analysis – During this phase, the gathered data is analyzed to be presented in an effective manner to the user. Also, potential risks and warnings about any software currently in execution are logged.
- Display – The results from the analysis must be presented to the user in an effective and intuitive manner.

By modifying the system calls themselves and thus adding in an additional monitoring layer to the actual system call layer as shown in figure 11, we minimize our impact on program execution and lower our chances of detection or modification by encrypted software. Our goal is not to be able to reverse-engineer programs but to ensure the user that their sensitive system information and processes are not disrupted by the execution of such software.

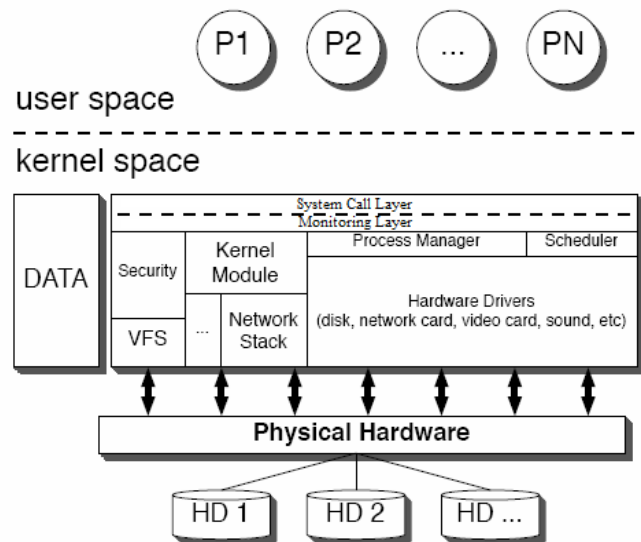


Figure 11. Modified system call layer

The critical component of the trust development architecture is the system call data gathering agent that is residing in the kernel. There are several locations in the kernel

where this data acquisition can take place, the syscall handler, at program startup by invoking strace on all new processes, and at the actual system call functions (see figure 12). There are tradeoffs with each of these approaches as we have to consider overhead, ease of modification, security, and customizability.

The invocation of strace upon every process startup is a simple means of obtaining the necessary system call information. The main drawback of this approach was discussed before. Any processes that are already being traced cannot be traced by another process. Thus, invoking strace or ptrace on all system processes will disable many utilities and features such as gdb. The other two locations in the linux kernel where data acquisition can take place can be considered in further detail. Both tapping the syscall handler and the functions themselves provide different security and monitoring benefits. By tapping just the system call handler, we are still able to acquire data and we can ignore potential changes at other system calls, such as sys_read being replaced and are still able to acquire data. In using this data acquisition location, the syscall handler becomes the single point of failure and thus should it be replaced by an attacker, then all system call information will be lost.

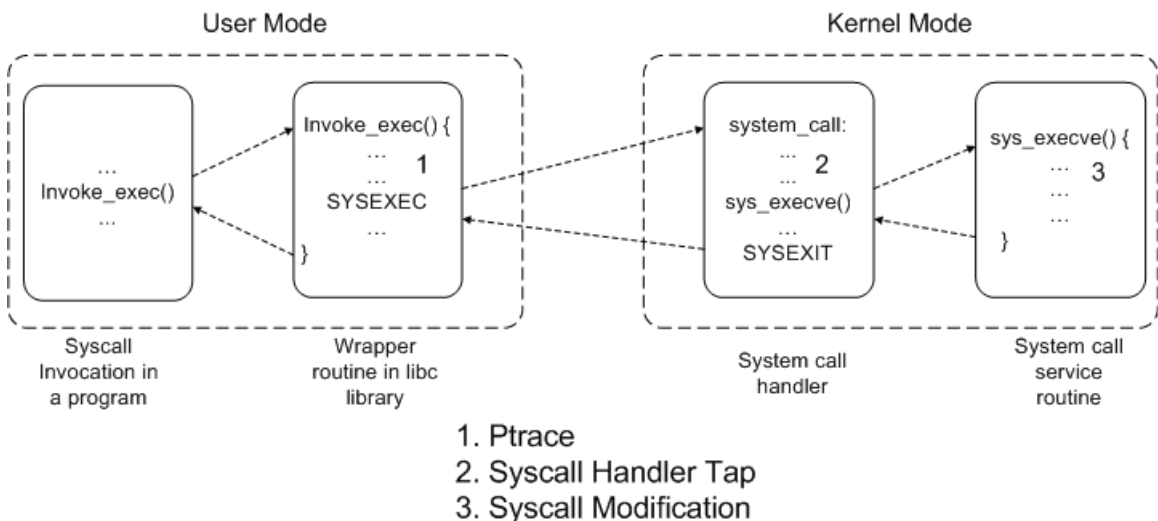


Figure 12. Three possible locations for system call monitoring.

The approach we choose to use was to modify individual system calls so as to monitor various aspects of the operating system. One of the primary concerns in the

modification of the kernel is the possible introduction of too much excessive code and the possible generation of too much data. If we tapped the system call handler, additional code must be introduced to ignore every unwanted system calls. On the other hand, it is possible to simply record all system call activity, but this approach can result in an overwhelming amount of data. Thus, we choose to modify target specific system calls to evaluate our methodology.

To be able to successfully monitor a process, a basic minimum set of system calls must be instrumented to obtain information. These are: `sys_execve`, `sys_fork`, and `sys_open`. These three system calls provides critical information about the program being executed, such as its binary filename, all the child processes, and the files accessed by that program. The logging mechanism will always be active due to the relatively infrequent hits to them when compared to other system calls.

The data gathered at the system call level must be exported to user space for storage. The process used to perform this phase takes advantage of the kernel message buffer. First, the size of the kernel message buffer is increased to lower the possibility of the buffer being overflowed due to excessive volume of data. Second, using this buffer allowed us to take advantage of the inherent kernel message-prioritizing property of the kernel message daemon. Figure 13 is a sample of raw `sys_open` data. The first column denotes the system call, followed by the pid, timestamp in seconds and micro seconds, the target file, the file descriptor number, the inode of the file, and a flag for successful or unsuccessful open. The data gathered from these modified system calls are then stored in a SQL database for easy access and searching by the analysis phase.

```
sys_open|7010|1192730750|892841|/etc/ld.so.cache|4|1374711|v1|
sys_open|7010|1192730750|892880|lib/tls/i686/cmov/libreadline.so|-2|v2|
sys_open|7010|1192730750|892970|/etc/ld.so.cache|4|1374711|v1|
sys_open|7010|1192730750|893025|usr/lib/libedit.so.2|4|834021||v1|
```

Figure 13. Raw `sys_open` data.

4.2 Trust Development

To develop trust in a black-box program, the program system call behavior is compared against a list of policies of the operating system. We begin by subdividing our system calls into several categories.

σ represents a system call

T represents the operating system

$T(\sigma)$ represents applying the system call to the operating system.

μ represents memory

Δ represents a change, i.e. $\Delta\mu$ represents a change in memory

$\Sigma = \{ \Omega, \Phi, \Lambda, M, Z, P \}$ - all system calls

$\Omega = \{ \text{execve, fork, open} \}$ - baseline calls

$\Phi = \{ \text{read, write, lseek, ...} \}$ - file system calls

$\Lambda = \{ \text{socket, connect, listen ...} \}$ - network calls

$M = \{ \text{mmap, munmap, mlock ...} \}$ - memory calls

$H = \{ \text{getpriority, setpriority, ...} \}$ - system operation calls

$\Theta = \{ \text{all other system calls} \}$ - all other calls

A second method of categorizing system calls is based on the type of modification is performed by the system call on the operating system:

$A \in \Sigma : \forall \sigma \in \Sigma \text{ where } T(\sigma) = T$ - subset of calls that only reads data

$B \in \Sigma : \forall \sigma \in \Sigma \text{ where } T(\sigma) = T + \Delta\mu$ - subset of calls that modifies memory

$\Gamma \in \Sigma : \forall \sigma \in \Sigma \text{ where } T(\sigma) \neq T$ - subset of calls that writes to I/O

$\Psi \in \Sigma : \sigma \notin \{ A \cup B \cup \Gamma \}$ - system calls that cannot be categorized

We further break down the subcategory of system calls that creates output into three groups:

$N \in \Gamma$ - subset of calls that writes data to user interfaces, such as the monitor

$Z \in \Gamma$ - subset of calls that writes data to the file system

$K \in \Gamma$ - subset of calls that writes data to output devices, such as network card

It is possible, that given the previous definitions, some system calls can exist in multiple categories. For instance, programs often use `sys_write` not only to write to files, but also to the monitor. Finally, one last subdivision breaks down the calls reading or writing to the file system based on the degree of sensitivity of the files.

A_p, Z_p - system calls that reads and writes files belonging to the program

A_l, Z_l - system calls that reads and writes to library files

A_u, Z_u - system calls that reads and writes to user files

A_s, Z_s - system calls that reads and writes to none library system files

A_x, Z_x - system calls that reads and writes to user defined sensitive files

Based on these definitions, it is possible to come up with rules for the operating system to estimate the degree of trust allowed for any particular program. For example, table 1 contains some sample policies for low, medium, and high risk programs.

Table 1. Example policies

Risk Factor	Policy	Explanation
Low	$\forall \sigma \in \text{Program where}$ $\sigma \in \{ \Omega \cup (\Phi \cap A_{pl}) \cup (M \cap B) \cup N \}$	Any program that read only from program and library files, performed memory operations, and output data to user devices such as the screen or the sound device is low risk.
Medium	$\forall \sigma \in \text{Program where}$ $\sigma \in \{ \Omega \cup (\Phi \cap A_{pl}) \cup (M \cap B) \cup N$ $\cup Z_p \cup (\Lambda \cap A) \}$	Any program that has the properties as the previous low risk program but additionally modified its own program files as well as read data from the network is medium risk.
High	$\forall \sigma \in \text{Program where}$ $\sigma \in \{ \Omega \cup A_x \cup Z_l \}$	Any program that read user defined sensitive files (such as a file containing their banking information) or modifies system library files is high risk.

4.3 Intrusion Detection

Providing intrusion detection for encrypted software is a necessary and integral part of the development of user trust. Due to its obfuscated nature, it is difficult for users to detect intrusions on obfuscated programs. It is the job of the intrusion detection system for scanning the processes on the system for potential compromises. If a compromise is detected, the IDS will build a report and transmit it to the user for verification. In this section, we discuss two approaches for accomplishing successful intrusion detection.

Through repeated executions of programs, it is possible to build up a profile based on the policies for the program. For example, a program with a low risk factor policy given in table 1 that begins to issue system calls belonging to the $\square\square$ subset is highly suspicious.

The second approach to intrusion detection is based on program behavior profiling. This is a combination of system call sequence monitoring and system call data monitoring. For system call sequence monitoring, the actions of the program are examined to detect anomalous behavior that should not typically exist. For example, any process that invokes `sys_execve` twice warrants further attention since this is a characteristic of a buffer overflow followed by some injected code designed to execute another program. For data monitoring, we examine the typical libraries used by the program and characteristic program behavior, such as the order in which library files are read from, and how the program creates and writes to temporary files.

4.4 System Operation

The procedure to accomplish our trust development and intrusion detection system is shown below in figure 14. This process begins when a system call request is made. Once the request is filtered down to the individual system calls, they are passed

through the monitoring layer, which duplicates the request and transmits it to the behavior intrusion detection system. The duplicate request is checked for behavior characteristics and is compared to the policy profile of the process. If an intrusion is detected by the behavior IDS, then a reject signal is sent to kill the system call request. If the program system call request does not match the policy of the program, then user input is requested for verification such as when a program is requesting write access to /etc/passwd. If the new policy is confirmed by the user, a “pass” signal is transmitted and the system call request is transmitted back to the process. If the program request matches the policy profile, then a “pass” signal is transmitted as well. In parallel with this operation, the monitoring layer passes the system call request through to be processed by the system. Once this request has been processed by the operating system, the resulting reply will be held by the monitoring layer until it is determined that it is okay for the process to receive the resulting data.

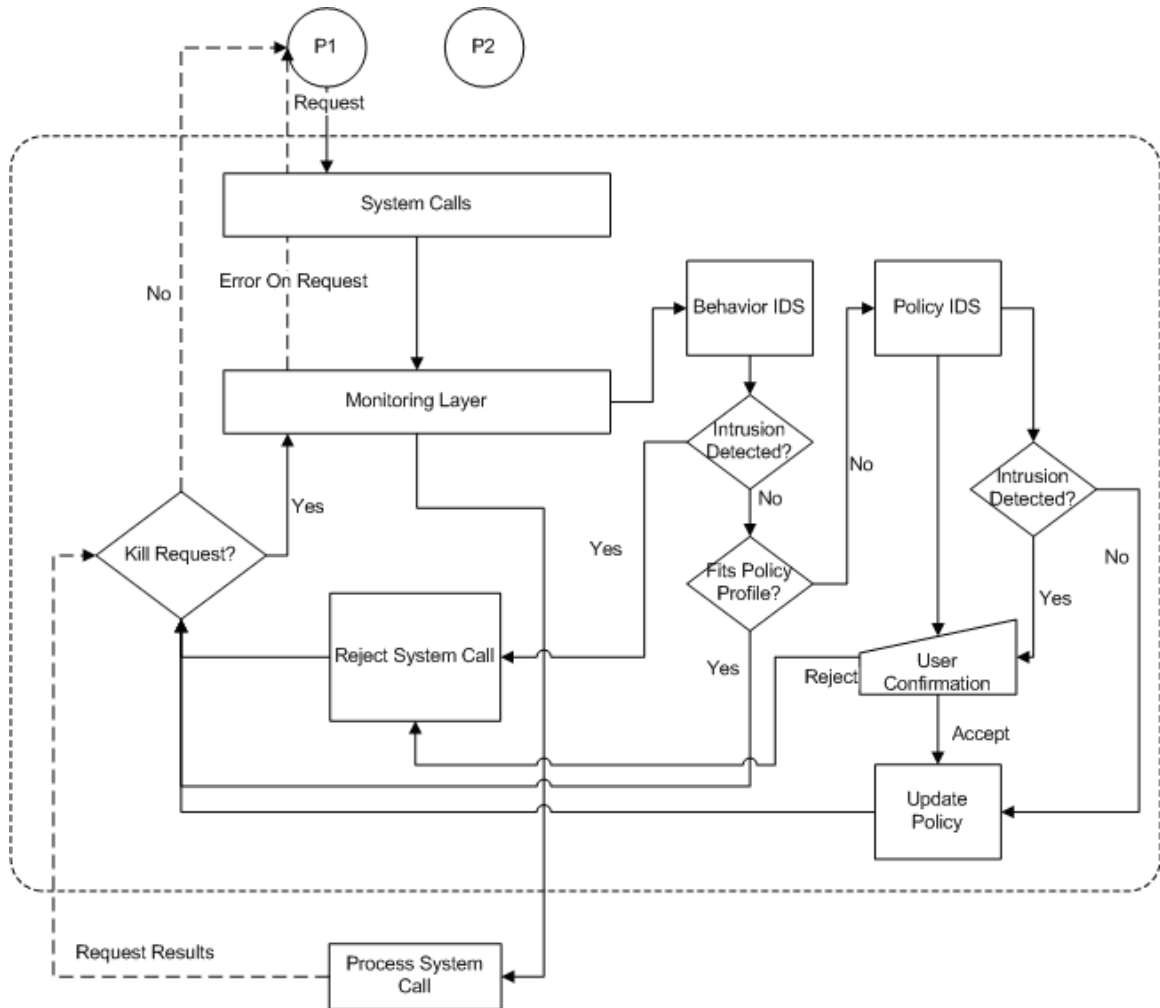


Figure 14. Modified system call layer

4.4.1 Behavior IDS

The behavior IDS is responsible for detecting changes in the program behavior, locating deviations with timing analysis, and identifying illegal operations. This phase consists of several databases generated from historical system call data and process information. This set of databases is then used as a baseline for new process behavior for anomaly detection.

The program behavior databases are used for standard error checking, such as accessing typical library files and writing to files. Each program has an associated file list consisting of common and uncommonly used system library files. During every

execution, it is expected that these files are opened for use. If a program begins to access other files during every execution without any recent modifications to the system or the program, then this bears further investigation. File access behavior is similarly examined. Should a program begin to write to its library files, for example, then a warning is generated and is brought to the attention of the user.

Timing analysis is very system and user specific. Our approach is to analyze the difference in time of easily identifiable program signatures. We believe that it is possible to obtain some characteristic information based on the amount of time measured by the system (in microseconds) from standard program access and behavior. If a process that is typically resolved in X amount of time requires $X+n$ amount of time where $n > \text{security threshold}$, this signifies a deviation from standard behavior.

Illegal operation identification is performed by checking for illogical system call behavior. This is different from the policy enforcement in that it is possible this illogical behavior uses only system calls allowed by the policy for the program.

4.4.2 Policy IDS

The policy intrusion detection system is responsible for keep track of system policies for all processes. When a system call is received by the policy IDS, it is tested against the policies of the program to determine if the system call belongs in any subset of the policies. If the check returns true, then the IDS will pass the system call. If the check returns false, a warning is generated and sent to a logging file for the user to peruse at a later time. If the system call belongs the categories Z_1, A_x, Z_x , then the system call will be suspended until the user accepts or rejects the action.

CHAPTER 5

EVALUATION OF PROGRAM MONITORING

In this section, we present the results of our research and demonstrate some applications of the data gathered from system call monitoring. First, we present our results from applying this method to encrypted software such as Skype and evaluate its success at detecting several types of viruses and exploits. Second, we introduce a possible application of this research with computer forensics. Third, we present our approach of the visualization our data for a more intuitive representation for users.

5.1 Program Monitoring

In order to evaluate our methodology to build trust in encrypted programs and the success of detecting intrusions, we have gathered a suite of programs, program encryptors and experiments that include legitimate programs and malicious code and exploits.

5.1.1 Tools

Below is a summary of the tools and scripts used in this experiment.

- **Sqlite3** - Sqlite3 is an open source serverless embedded SQL database. It uses a single file on the host to construct all of its database structures. As such, Sqlite3 is an excellent alternative for data storage in large applications as opposed to the creation of various customized file formats. The main goal behind the design of Sqlite3 is to make it easy to administer, operate, embed into programs, maintain, and customize when compared to large enterprise database solutions such as PostgreSQL or Oracle while maintaining simplicity [43].
- **Xnee Suite** - Cnee and Gnee are a part of the GNU Xnee suite of tools designed for recording and replaying user actions. For the recording of playing back user actions for our experiment, we used Gnee configured with

the following even list: KeyPress, KeyRelease, ButtonPress, ButtonRelease, MotionNotify, EnterNotify, LeaveNotify, FocusIn, and FocusOut.

- Tinyweb - Tinyweb is an extremely small and simple daemon for tcp/http web-servers. Tinyweb requires no configuration other than through the command line and consumes very little system resources. In our experiment, we intentionally left a buffer overflow exploit at the receiving buffer for incoming traffic and disabled the stack smashing protection option with gcc.
- Ubuntu with 2.6.20 kernel - The Linux system for the experiments consists of a stable Ubuntu[44] hardy heron distribution running a customized 2.6.20.16 custom kernel.
- Emacs - A recent Emacs 22.1 design flaw published by Security Focus allowed the injection of code into the user-init file if a specific sequence of local variables were embedded a file. It is possible for attackers to execute programs and other code through this Emacs exploit.
- Linux Elf_Nel.A (Caline) - The Caline virus [38] is a simple program written by researchers to infect Linux executables. When executed and given a parameter, the virus will attempt to attach itself to the binary.
- UPX - The Ultimate Packer for eXecutables software is a free and portable executable packer that offers very high compression ratios and fast decompression. UPX supports a variety of operating system environments, including Linux.

5.2 Results

In one experiment we demonstrated that our architecture can successfully monitor an encrypted program by the successful monitoring of Skype (see figure 15). This figure denotes the files opened by Skype during startup. While not interesting in itself, this

demonstrates that our monitoring method has no issues with the anti reverse-engineering mechanisms of Skype.

```
└─usr/bin/skype Assigned PID 9789
PID: 9789 Opening /etc/ld.so.preload
PID: 9789 Opening /etc/ld.so.cache
PID: 9789 Opening /usr/lib/libqt-mt.so.3
PID: 9789 Opening /usr/X11R6/lib/libXext.so.6
PID: 9789 Opening /usr/X11R6/lib/libX11.so.6
PID: 9789 Opening /lib/tls/libpthread.so.0
PID: 9789 Opening /usr/lib/libstdc++.so.5
PID: 9789 Opening /lib/tls/libm.so.6
PID: 9789 Opening /lib/libgcc_s.so.1
PID: 9789 Opening /lib/tls/libc.so.6
PID: 9789 Opening /usr/lib/libfontconfig.so.1
```

Figure 15. `sys_open` calls of Skype.

Over a period of 4 seconds, Skype made 16 reads from `/dev/urandom`, attempted to open 394 files (238 of which were successful), and 217 writes.

In another experiment, we simulated an attacker running a buffer overflow exploit on Tinyweb (Figure 16). For the first few sequences, we see the normal operation of tinyweb as it accepts web requests and transmits the response by accessing the `index.html` file. Once the exploit request is transmitted, however, we detected that the PID of tinyweb made a second `sys_execve` call, this time as `/usr/bin/sh`. This second call to `sys_execve` is highly suspect as PIDs are assigned per process and is not reused. Thus, this call would immediately be marked for inspection by the program behavior database. After this call to `/usr//sh`, our attacker connects via the bound port and begins executing commands as root. Each of these executions was captured by the system call monitoring as a new forked process by the PID.

```

sqlite> select * from syscall_list where pid=7043 order by sec, usec;
sys_close|7043|1209238764|608269|N/A|4|N/A|N/A|N/A|N/A
sys_read|7043|1209238764|608707|N/A|3|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|608723|N/A|3|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|609152|N/A|255|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|609296|N/A|3|N/A|N/A|N/A|N/A
sys_execve|7043|1209238764|609310|./tinyweb|N/A|N/A|N/A|N/A|N/A
sys_open|7043|1209238764|609399|/etc/ld.so.cache|3|1375382|N/A|v1|N/A
sys_close|7043|1209238764|609418|N/A|3|N/A|N/A|N/A|N/A
sys_open|7043|1209238764|609455|/lib/tls/i686/cmov/libc.so.6|3|2075940|N/A|v1|N/A
sys_read|7043|1209238764|609465|N/A|3|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|609649|N/A|3|N/A|N/A|N/A|N/A
sys_write|7043|1209238764|609997|N/A|1|N/A|N/A|N/A|N/A
sys_write|7043|1209238764|88963|N/A|1|N/A|N/A|N/A|N/A
sys_write|7043|1209238764|89234|N/A|1|N/A|N/A|N/A|N/A
sys_open|7043|1209238764|89263|./webroot/index.html|5|2382732|N/A|v1|N/A
sys_write|7043|1209238764|89277|N/A|1|N/A|N/A|N/A|N/A
sys_read|7043|1209238764|89515|N/A|5|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|89546|N/A|5|N/A|N/A|N/A|N/A
sys_write|7043|1209238764|115707|N/A|1|N/A|N/A|N/A|N/A
sys_write|7043|1209238764|117438|N/A|1|N/A|N/A|N/A|N/A
sys_open|7043|1209238764|117464|./webroot/image.jpg|6|2382731|N/A|v1|N/A
sys_write|7043|1209238764|117474|N/A|1|N/A|N/A|N/A|N/A
sys_read|7043|1209238764|117598|N/A|6|N/A|N/A|N/A|N/A
sys_close|7043|1209238764|121182|N/A|6|N/A|N/A|N/A|N/A
sys_write|7043|1209238781|631145|N/A|1|N/A|N/A|N/A|N/A
sys_write|7043|1209238781|634394|N/A|1|N/A|N/A|N/A|N/A
sys_write|7043|1209238781|634404|N/A|1|N/A|N/A|N/A|N/A
sys_close|7043|1209238792|10613|N/A|9|N/A|N/A|N/A|N/A
sys_execve|7043|1209238792|10632|/bin//sh|N/A|N/A|N/A|N/A|N/A
sys_open|7043|1209238792|10718|/etc/ld.so.cache|9|1375382|N/A|v1|N/A
sys_close|7043|1209238792|10733|N/A|9|N/A|N/A|N/A|N/A
sys_open|7043|1209238792|10767|/lib/tls/i686/cmov/libc.so.6|9|2075940|N/A|v1|N/A
sys_read|7043|1209238792|10777|N/A|9|N/A|N/A|N/A|N/A
sys_close|7043|1209238792|10823|N/A|9|N/A|N/A|N/A|N/A
sys_read|7043|1209238796|190471|N/A|0|N/A|N/A|N/A|N/A
sys_fork|7043|1209238796|190622|N/A|N/A|N/A|7062|N/A|N/A
sys_read|7043|1209238797|143590|N/A|0|N/A|N/A|N/A|N/A
sys_fork|7043|1209238797|143871|N/A|N/A|N/A|7063|N/A|N/A
sys_read|7043|1209238828|179423|N/A|0|N/A|N/A|N/A|N/A

```

Tinyweb Execution

Responding to standard inquiry with html and jpg files.

shell exploit!

Forks of processes by attacker using the new shell

Figure 16. Monitoring an exploit.

In the following experiment, we traced two Linux viruses, Caline and Linux.RST.B-1. We first performed a scan on the two viruses and determined that McAfee antivirus detected them. Next, we used the UPX program packer to pack the two viruses and repeated the scan. This time, the antivirus scanner did not detect the virus signatures anymore. Finally, we executed these viruses on our system and recorded the results. During this execution we observed the behavior of Caline. It is a simple virus designed to infect a target executable when the infected program is executed. In the process, our monitoring system detected the creation of a temporary file used by Caline during the infection process. For Linux.RST.B-1 we determined that the virus, when executed, attempts to infect as many common Linux programs in /usr/bin as possible. The list of detected files is presented in Figure 17. We can also obtain an estimate of the age of the virus and the Linux distro it is mainly targeting based on the names of the

/usr/bin files Linux.RST.B-1 attempts to open. This is due to the fact that the contents of /usr/bin have changed over the years and for different distributions.

```
sys_open|7134|1197312984|187557|mount|5|1452545|N/A|v1|N/A
sys_read|7134|1197312984|187796|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|187923|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|187935|N/A|5|N/A|N/A|N/A|N/A
sys_close|7134|1197312984|187941|N/A|5|N/A|N/A|N/A|N/A
sys_open|7134|1197312984|187966|ps|5|1452562|N/A|v1|N/A
sys_read|7134|1197312984|188162|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|188173|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|188185|N/A|5|N/A|N/A|N/A|N/A
sys_close|7134|1197312984|188191|N/A|5|N/A|N/A|N/A|N/A
sys_open|7134|1197312984|188210|stty|5|1452581|N/A|v1|N/A
sys_read|7134|1197312984|188262|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|188271|N/A|5|N/A|N/A|N/A|N/A
sys_write|7134|1197312984|188281|N/A|5|N/A|N/A|N/A|N/A
sys_close|7134|1197312984|188285|N/A|5|N/A|N/A|N/A|N/A
```

Figure 17. Linux.RST.B-1 modifying /bin processes

In another experiment, we tested an infection of the Emacs init file through an exploit of Emacs 22.1. In this exploit, a file containing a variable list exploit allows the attacker to compromise the Emacs init file resulting in the addition of the payload to the file for execution. We performed several analysis types with this exploit. First, we captured the standard Emacs behavior during startup, and compared it to the behavior of Emacs if this exploit was embedded in the document being opened. Figure 18 shows the results for inode usage and timing based on the two executions. On the left, we see the standard emacs execution behavior. On the right, we see the exploited emacs execution behavior. There is a noticeable difference between the timing delays of the two types of executions based on inode usage.

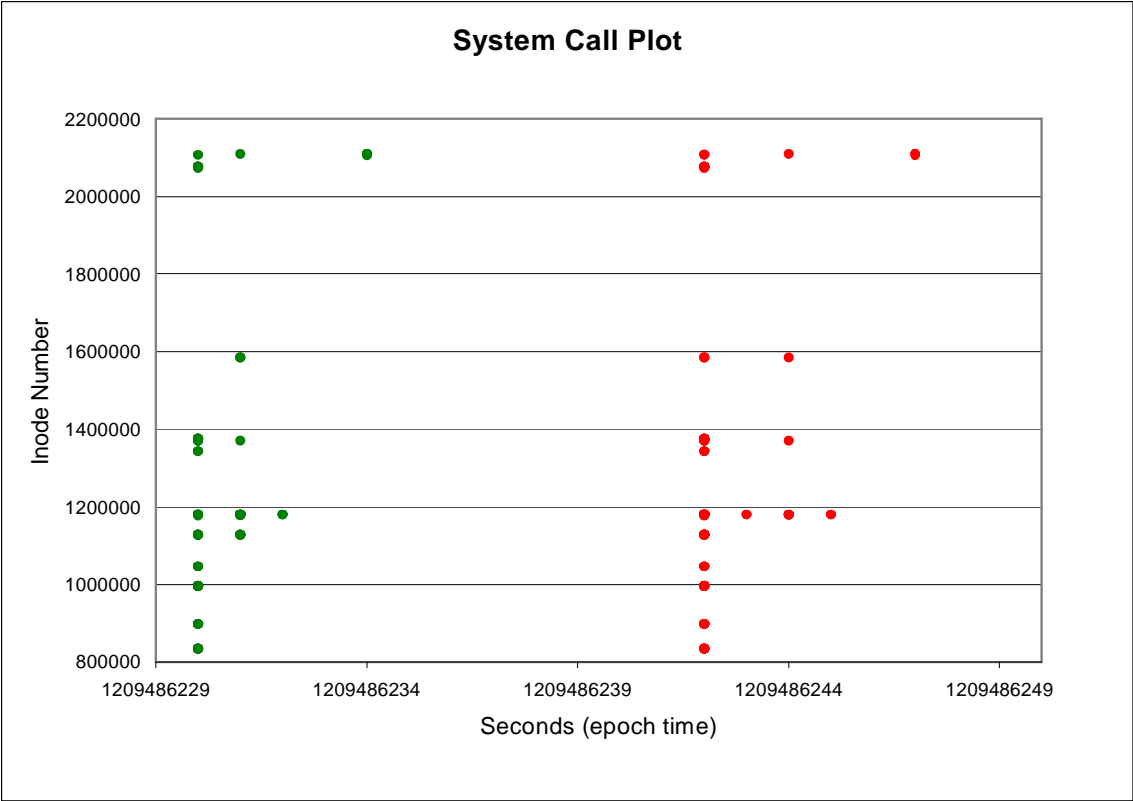


Figure 18. The execution on the left is standard Emacs behavior when opening a file. The execution on the right is the behavior of Emacs when opening an infected file.

CHAPTER 6

PROGRAM BEHAVIOR PROFILING

In this chapter, we apply our program monitoring data to assist in the reconstruction of past historical events captured in the file system journal. Although there can be a degree of error in this process, we show that by using the captured system call data as a training set to create a program behavior profile of programs as seen from the journal, we lower the chances of misidentifying the program. Moreover, using this approach gives forensic experts a new source of information that they can use to reconstruct past events.

6.1 Architecture

The generation of our program behavior database requires two components: a system call database and a journal entry database. These must be created concurrently while performing standard program and code execution on the target computer. For post intrusion analysis, the forensic investigator will have to execute all possible permutations of programs on the target machine to generate this behavioral database. It should be noted that the use of advanced applications requires more resources which increases the complexity of this analysis, thereby increasing the time required for the forensic investigation. However, this system is designed to aid in offline forensic investigations and thus the retrieval of more extensive evidence offsets the heavier computational requirements. The database will be used to derive events from the original journal log.

A program behavior database is created from a combination of system call and journal harvested metadata databases. The system call database contains information as seen from the operating system such as file opens, reads, writes, program executions, and terminations. The journaling database contains inode metadata inserted into the file system. By correlating the syscall data with the information that has been written to the

file system journal, we are able to detect a range of identifying information from the programs being executed on the system. This gathered evidence allows us to identify key processes and events by examining the journal log in detail. Figure 19 presents our forensic process.

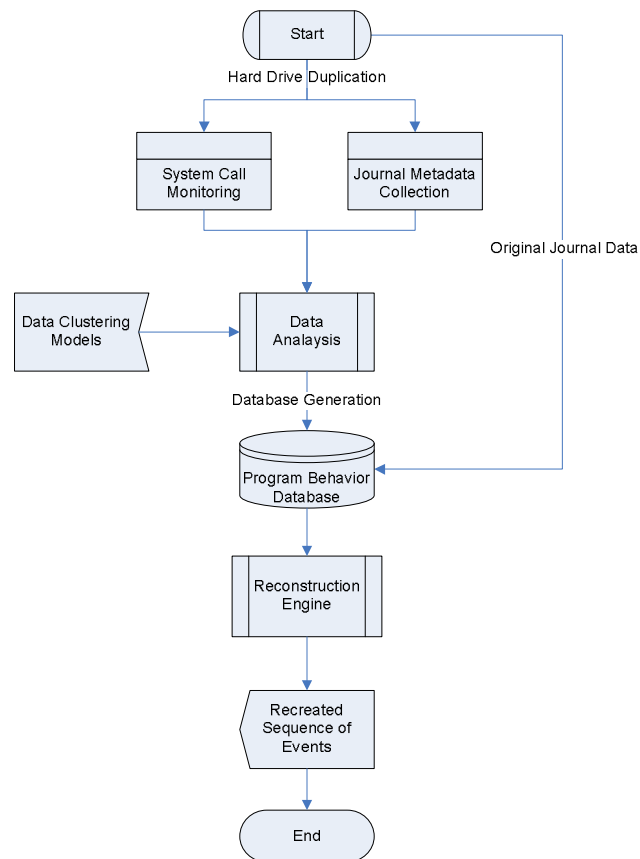


Figure 19. Forensics Process

As discussed in section 2, system call monitoring is not a new concept. However, in our architecture, we apply system call monitoring in a new application by using it to aid in file system forensics rather than intrusion detection. Our primary motivation for this approach is to retrieve information about the file system, such as file names of particular inodes and the sequences of syscalls performed to identify and catalog relationships.

6.2 Database Generation

We must first generate the program behavior database to empower the forensic investigators with the ability to reconstruct events based on the journal metadata. At this point in our system, we have two separate master databases containing the behavior information of various processes as they execute on the system in question. The next step is to apply our data extraction models to obtain the various program signatures for forensic reconstruction of events documented in the journal log. This is perhaps the most time consuming phase of the forensics investigation as there is often a vast quantity of data to dissect. Moreover, there are additional data modeling techniques that may be applied to these logs to create different program signatures. To demonstrate our architecture, however, we chose the methods described below.

The first database generated contains program-file relationships (PFR). In this database, we keep track of every file required for the execution of any particular process. A counter is used for every instance of the program to keep track of the number of executions. The inode number and filename of every file touched by that program is also entered into the database along with a counter showing how many times that process has accessed that particular file. This information allows us to quickly identify the library and data files required by locating files that are read during every execution of the process. Lsof produces a similar data set without the benefit of knowing the order in which the files were accessed.

The second database, using only the system call information, is a time ordered behavior database (TOB). Given that most processes follow an ordered sequence of algorithms to perform certain tasks, we can develop an order of events for any particular program in terms of file accesses. For example, some programs may choose to load all the libraries in one particular directory first while others will load files as needed in no particular order. To generate this database, different instances of program execution are

compared with each other to identify the similarities and differences under varying program behavior. File access orders are compared as well as the times between each file access.

Similarly, several databases can be created with just the raw journal data (see Figure 20). Along system call monitoring, these databases are used to generate the program behavior database as well as to verify or reject detected patterns. The first column of the data is the inode number, followed by mtime, atime, ctime, and dtime.

```
898630|1176638725|1192740160|118581580
6|0
898631|1176638618|1192740160|118581580
6|0
2107354|1192740160|1192740160|11927401
60|1192740160
2105282|1192740160|1192740088|11927401
60|0
833391|1185815764|1192740160|118581576
4|0
```

Figure 20. Raw journal data.

The first journal-based database is obtained by applying a data clustering approach with a distance measure calculated by using the modification, access, creation and deletion time values for the inodes being considered. With this approach, files that are touched in a relatively short time period from each other and files with similar inode numbers will be added into the database as being “related.” As this database grows, more and more relations between files will be inserted, and files that become strongly related due to a large amount of near concurrent accesses in the system will stand out more than others. This database is called the Time Relationship (TR) Database. Several further refinements can be applied to the TR Database to generate additional relationships by examining the specific MAC times that created the link between the two inodes. For example, if we repeatedly see an atime in the first file matching the mtime in a second file, we can establish a flow relationship read-write between those particular inodes.

Some other relationships include read-delete, read-create, read-read, write-write, and so on. Figure 21 gives an overview of our data analysis process.

The list of files and behaviors used to uniquely identify a process is called a “dominant feature”. This information is extracted from the TRD and PFR databases. There are times where multiple programs may share some or all library files resulting in the dominant features of one program masking that of the other, we call this phenomena data collusion. The extraction of the library files to identify dominant features is performed through analyzing the PFR database. Combining the information about the program library files and the TOB generates a time ordered dominate feature list. This step and other dominant feature extraction methods are performed in the program behavior engine.

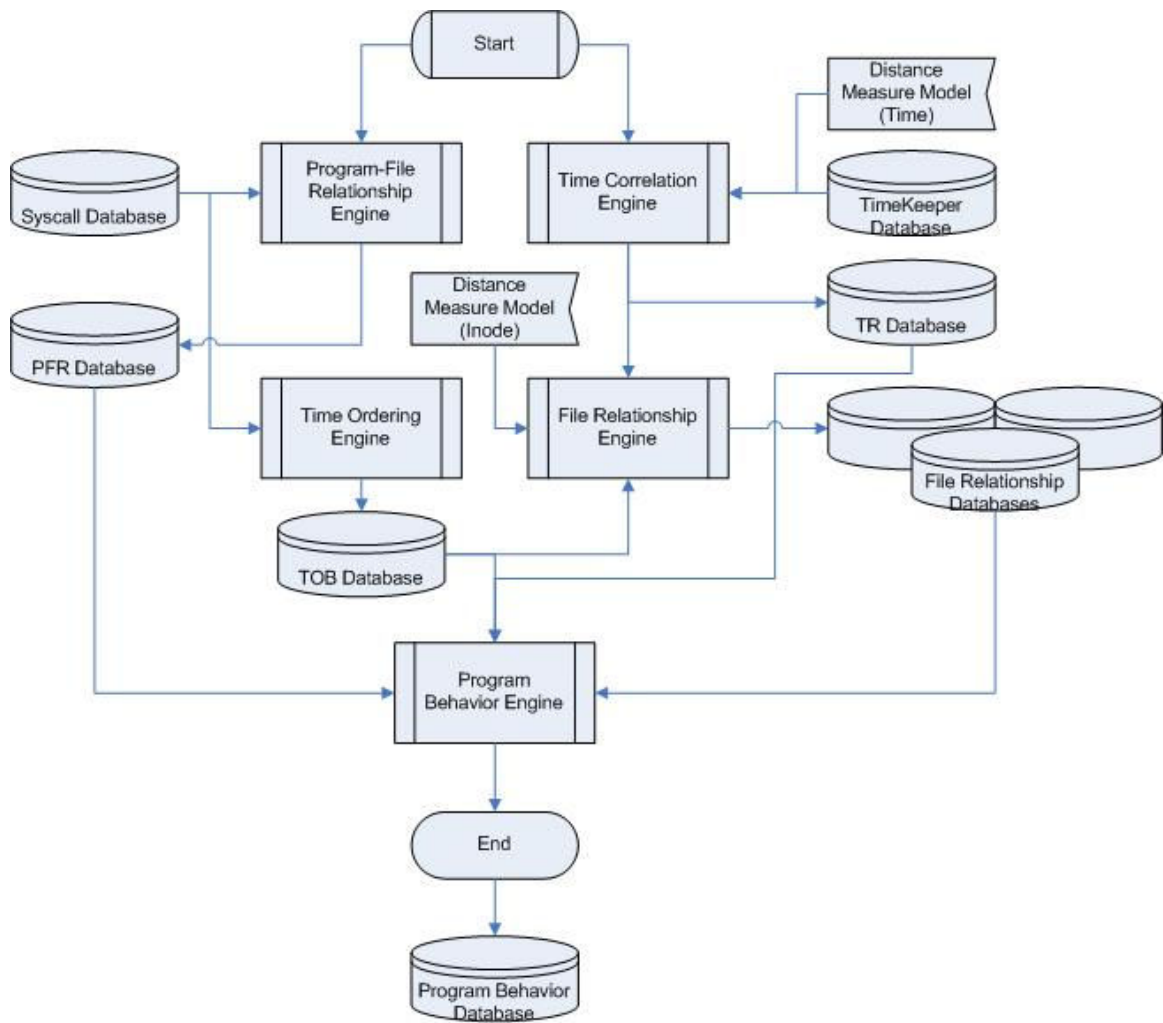


Figure 21. Data analysis procedure

The final objective of the program behavior engine is the creation of various program signatures from the dominant feature extraction routines. These signatures can fall under several categories such as program library files, program file access behavior, and time based behavior. Program library file signatures identify programs based on the library files accessed during runtime. Program file access behavior signatures identify specific methods and ordering a program uses to access files. Program time based behavior signatures identify the amount of time programs may take to perform common actions. The creation of these signatures makes it possible to uniquely distinguish program executions from the file system journal.

6.3 Results

In this section, we will present some of the more interesting results obtained from our experimentation. Thus far, we have successfully detected the execution of programs through the correlation of the journal data with our database via dominant feature extraction as well as program behavior signature identification. In the process, we have learned in great detail the operations of these programs and the background events that are typically obscured from the user.

Applying dominant feature extraction and program behavior models to the data generated in the experiment identified several unique program signatures. Several smaller programs were less readily identifiable due to data collusion to be discussed later. The dominant feature targeted in this demonstration was the set of library and support files for each program. Using our PFR database, we compared the number of times a program used a file during execution to the number of times a program was executed. Files were sorted based on priority depending on the correlation between the file usage and program execution. Files accessed upon every program execution were labeled as a library file. Files accessed above a certain threshold (in this case 80%) were labeled as potential library files. Files with a correlation above 10% and below 80% were considered behavior files that relied on program usage. All other files were considered temporary files that were touched in only a few scenarios. Table 2 shows a subset of a file list for one program and the file categories.

In order to distinguish a program in the event of data collusion, additional files have to be added to the dominant feature list. Although some programs may camouflage others, the lack of a library or data file may be telltale distinction as well. In our example, although vim uses every library file cat uses, it also uses several additional library files that cat does not use. Thus, notable absent files are also appended to the set of library files for programs known to be masked by others.

Table 2. Sample /bin/cat file list

Program - /bin/cat	usage
424463 - /tmp/v750125/0	1 (temp)
898622 - /usr/lib/locale/en_US.utf8/LC_CTYP E	98 (lib)
836025 - /usr/lib/gconv/gconv- modules.cache	98 (lib)
1371027 - /usr/share/locale/locale.alias	98 (lib)
1371051 - /etc/papersize	49 (behavior)
898620 - /usr/lib/locale/en_US.utf8/LC_ADD RESS	98 (lib)

The sequence of actions for each program was examined to detect uncommon flows of events. In Figure 22, we show the behavior of vi and vim when editing a new file that was passed to the program at the prompt. This sequence of the creation and deletion of the backup files .swp and .swpx upon program execution is a unique behavior only exhibited in vi and vim. Using this pattern, it is then possible to detect the creation and editing of new files if done through these two editors.

```

Syscalls  PID  Time sec  usec  filename or FD
sys_execve|8307|1192732696|118015|/usr/bin/vi      vi execution
sys_open|8307|1192732696|123568|/etc/passwd|3|1372878|v1
sys_close|8307|1192732696|123621|3              Permission checks
sys_open|8307|1192732696|123684|temp|-2|v2
sys_open|8307|1192732696|123707|.temp.swp|-2|v2
sys_open|8307|1192732696|123824|.temp.swp|3|2107221|v1
sys_open|8307|1192732696|123891|.temp.swpx|-2|v2
sys_open|8307|1192732696|123912|.temp.swpx|4|2107354|v1
sys_close|8307|1192732696|123924|4
sys_unlink|8307|1192732696|123941|2107354
sys_close|8307|1192732696|123954|3
sys_unlink|8307|1192732696|123968|2107221
sys_open|8307|1192732696|124001|.temp.swp|3|2107221|v1
sys_write|8307|1192732696|124049|3
sys_write|8307|1192732697|616468|3
sys_open|8307|1192732699|360917|temp|4|2107354|v1
sys_write|8307|1192732699|360965|4
sys_close|8307|1192732699|374346|4
sys_write|8307|1192732699|374579|3
sys_write|8307|1192732699|374688|1
sys_write|8307|1192732699|374734|1
sys_close|8307|1192732699|374773|3
sys_unlink|8307|1192732699|374801|2107221

```

Unique vi behavior, after a failed file attempt, it will attempt to open, create, and then delete 2 temporary files labeled .swp and .swpx

A recreation of the .swp file for temporary storage

The saving of data to the official file name of "temp", an update to the directory entry (fd = 1), and the deletion of the .swp file

Figure 22. Unique vim behavior

Using the TR database, we detect relationships between files that are not in the library list of the programs. In our example (see Figure 23), we first identify several key library file inodes used by each program and then examine the database to detect additional files that may have a correlation to those library files. In this example, we know that inodes 1374711 and 2075940 are a pair of files that are opened concurrently only by cat and vi. With this knowledge, we examine some of the other files that have been given a relationship to 1374711 with roughly the same number of established relationships, and we see that inodes 2075946, 836025, and 898620 have this pattern and thus are potentially related to the execution of cat and vi/vim. Examination of the PFR database confirms those files as being location files used by those processes.

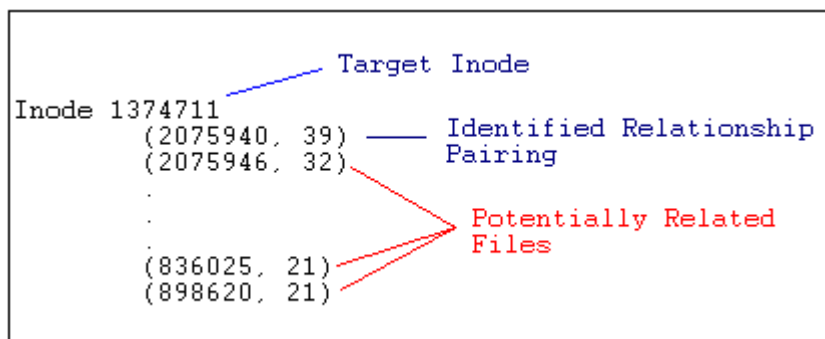


Figure 23. Distance measure analysis.

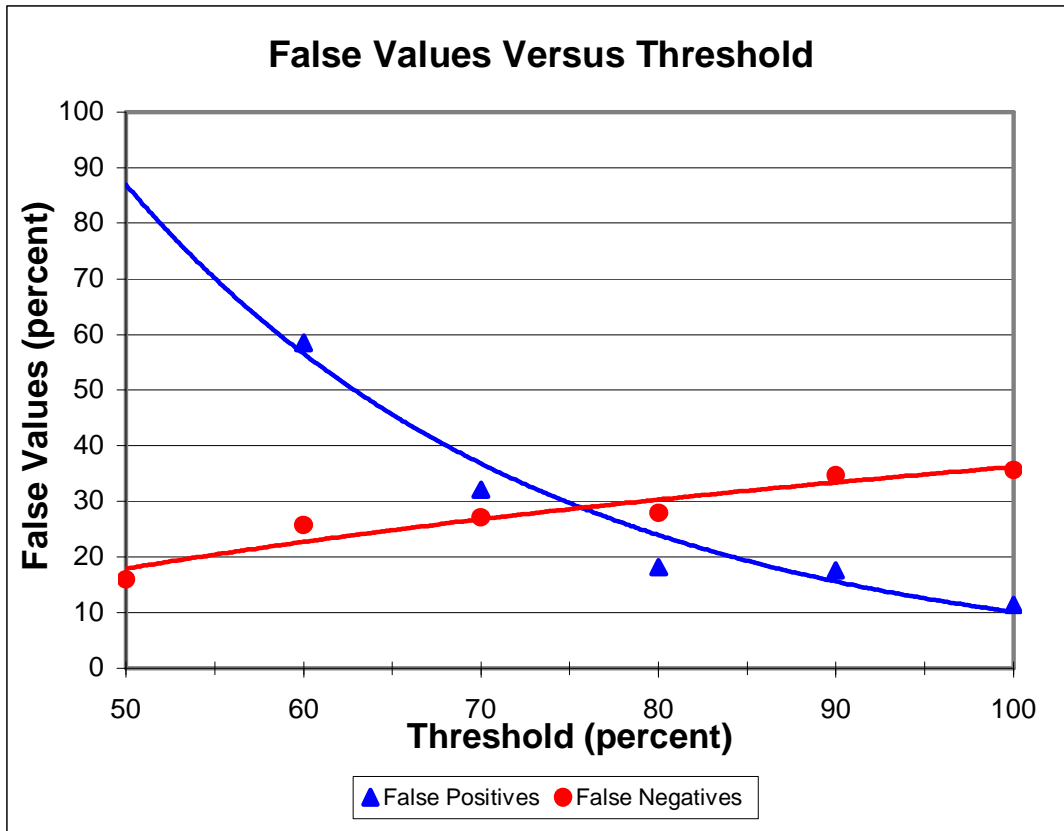


Figure 24. Dominant feature extraction accuracy.

We created a database of dominant features for the sample programs we chose. With this database, we performed dominant feature extraction on a set of journal data generated from the execution of these programs. Figure 24 shows the accuracy of our method when given a varying set of thresholds. In our experiment, we define a false positive as a time segment incorrectly identified as a target program execution. A false negative is defined as a target program execution that was not identified based on the journal data. The threshold value used here is the percentage of files from the dominant feature list that must be matched for positive program identification. From this graph, we can see that as the threshold percentage is increased to 100%, the amount of false positives drop to 11% while the amount of false negatives increase from 16% to 35%. We also can conclude that for the programs and tools used in this experiment, the optimal threshold value is approximately 76%.

We also noticed that programs with bigger dominant feature sets generated less false positives but more false negatives than programs with smaller sets. There were also more false negatives than false positives for programs with a small dominant feature set. For programs that have may involve partial data collusion where they share a majority of libraries and files, such as vi and vim, the false positive value drops off sharply after the threshold is raised to be above the percentage of files that each program shares with the other. Table 3 displays the false positive and negative values for one trial of 50 executions of vi and vim. From this table, we can conclude that between 70% to 80% of the files in the dominant feature set of vim is also a subset of vi while almost 100% of the dominant feature set of vi is a subset of vim.

Table 3. Vi and vim accuracy comparison

Threshold - vim	False Positive #	False Negative #
50	127	0
60	89	0
70	46	1
80	0	4
90	0	49
100	0	49
Threshold – vi		
50	149	0
60	149	0
70	93	0
80	49	0
90	46	1
100	0	1

The first type of program behavior matching we performed was the identification of read-write relationships. During a trial execution, we created, edited, copied, and deleted various test files using our chosen programs. Using the data collected with metadata archiving [4], we built our TR database and extracted file pairings that exhibited this read-write relationship. Using this relationship, we attempted to pick out instances where a write to a file occurred and matched the program execution at that interval with

the file. From Figure 25, we can see that the degree of accuracy at which our program behavior model identified read-write flow relationships between files is independent of the number of relationship links. This result is rather surprising as we expected to see a growing degree of accuracy as more relationship link pairs are identified. We hypothesize that this may be due to the large number of temporary files that certain editors such as vi create and delete as part of normal operation. This behavior may have skewed the accuracy of files pairs with a low number of relationship links.

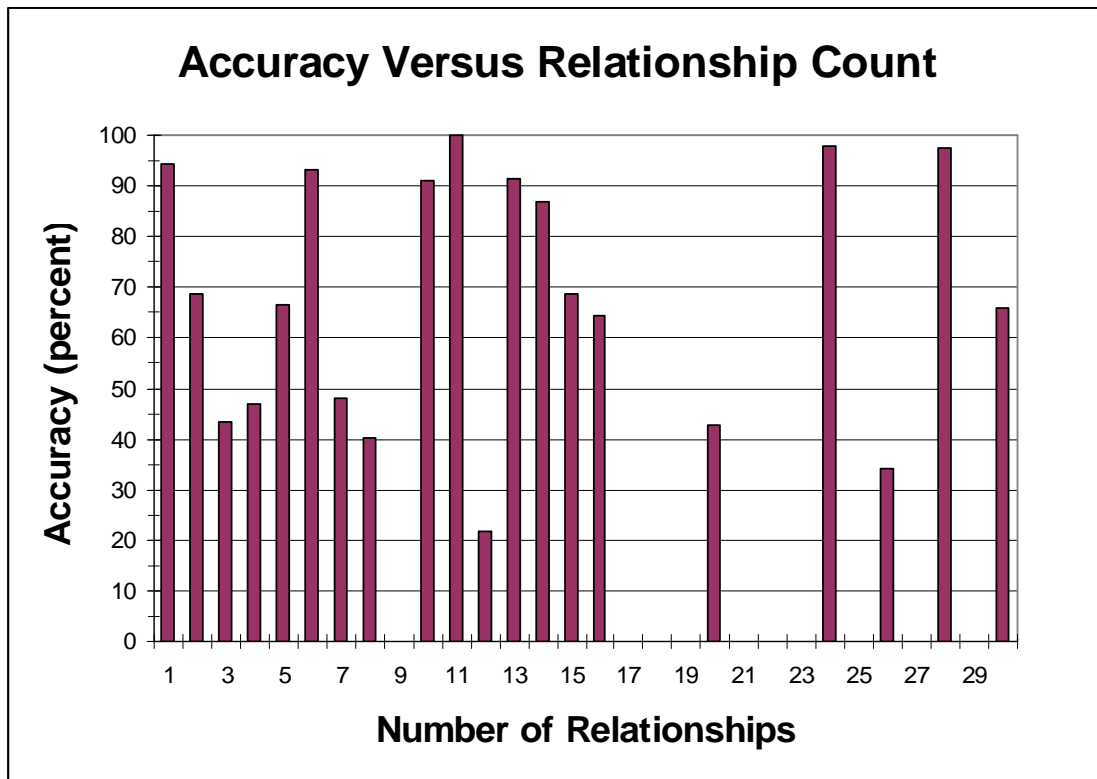


Figure 25. Read-write relationship accuracy

The previously discussed dominant features that are manifested in the journal due to the execution of a program are but one view of the information. There also exist other files that are regularly accessed. These files may not be as prominent as the dominant features, but are still telltale signs that a particular program may have been executed. If files are accessed during the same period of time, there may be a relationship. In addition, files which have close inode numbers may be related as they may be in the same

directory structure. Here, it is difficult to derive a standard parameter to determine the closeness as every file system should be viewed as unique, but as the distance increases the probability of the files being related should decrease. Therefore, a threshold is set that attempts to use the distance between inodes and the time values to establish a relationship.

CHAPTER 7

VISUALIZATION

Host based program monitoring tools are an essential part of maintaining proper system integrity due to growing malicious network activity. As systems become more complicated, the quantity of data collected by these tools often grows beyond the ability of analysts to easily comprehend in a short amount of time. In this section, we present a method for visual exploration of a system program flow over time to aid in the detection and identification of significant events. This allows automatic accentuation of programs with irregular file access and child process propagation, which results in more efficient forensic analysis and system recovery times.

The motivation of our research is to address the following points.

- Visualization of program flow data gives the user a better overall view of system behavior.
- Irregular events are more intuitively identifiable when presented visually.
- Automatic accentuation of events lowers data analysis time and draws attention to trouble spots.
- Taint propagation of process flows allows for rapid estimation of damage suffered following an intrusion.

To achieve these goals, we first present our system call data as a flow diagram based on time. As additional system processes spawn and files are accessed, each event is displayed at the time interval in which it occurred. Over time, as repeated program executions and file accesses occur, a standard pattern for proper behavior emerges for the system. During periods of irregular activity, new flow patterns may emerge and are brought to the attention of the user. Finally, should any process or file become tainted,

we then propagate that taint through the flow diagram based on program and file access to show the extent of potential damage occurred on the system.

7.1 Tools

Prefuse - Prefuse [45] is a software framework written in Java to aid developers in the creation of information visualization applications. The goal of Prefuse is to greatly simplify the process of data handling and representation. The reason we chose Prefuse over other visualization toolkits such as Piccolo, Dot, and the Visualization Toolkit is its flexibility, the ability to render large amounts of data in an efficient manner, and excellent documentation.

7.2 Approach

Our approach to program flow visualization can be broken down into three stages: data acquisition and storage, analysis, and visualization. It is not necessary that these operations be performed in the order listed. In fact, further data analysis may be performed based on user input from examining the resulting visualization. Finally, only the data acquisition and storage aspects must be performed in real time on the target system to be monitored. The analysis and visualization aspects can be performed elsewhere. The data used in this section was collected using our method of modifying and gathering specific system call data targeting that of process execution and file usage. In order to present our data visually, we must first process the raw data obtained from system call monitoring. This is achieved through the creation of several databases containing subsets of the logged information. The first such database contains the program-file relationships. This database contains the file usage of programs during execution, as seen by `sys_open`. Our goal for creating this database is to identify the key library files used by the program for execution. The second database we create using the system call data is a time ordered database which shows the order in which a program

accesses its library files. Generally, each program accesses files in a predefined manner at predictable intervals between each access. Using the program-file relationship database and the time ordered database, we are thus able to obtain a general idea of proper program execution behavior. A third database, containing the list of sensitive system files (such as `/etc/passwd`), is also generated for modification detection and accentuation in the visualization phase.

As previously discussed, one of the goals of visually representing the program flow is to provide a better overall view of the entire system. To achieve this goal, we chose to visually represent our data in the following way:

Each PID is a node in the flow diagram; child PIDs and accessed files are children of that node.

The flow diagram is time ordered using the available time stamp of each logged operation. Due to the fine-grained time unit used by Linux (1-10 seconds per step), we chose to categorize every event that occurs in a certain time period as concurrent. The way this is chosen is user defined. For example, the user may chose that all events that occurred within the same second be treated as concurrent, or perhaps all events that occurred within the same millisecond. The length and scope of the tree will be determined by this user input.

The diagram is collapsible at each node and the tree is in collapsed mode by default. As the user clicks on each node, it renders itself into a sub-tree consisting of child processes and accessed files. The files identified as library files under the program-file relationship database are displayed as a single collapsed cluster. A search option is also provided for the user to quickly locate a particular PID or file within the tree. Figure 26 shows a generalized view of our visualization.

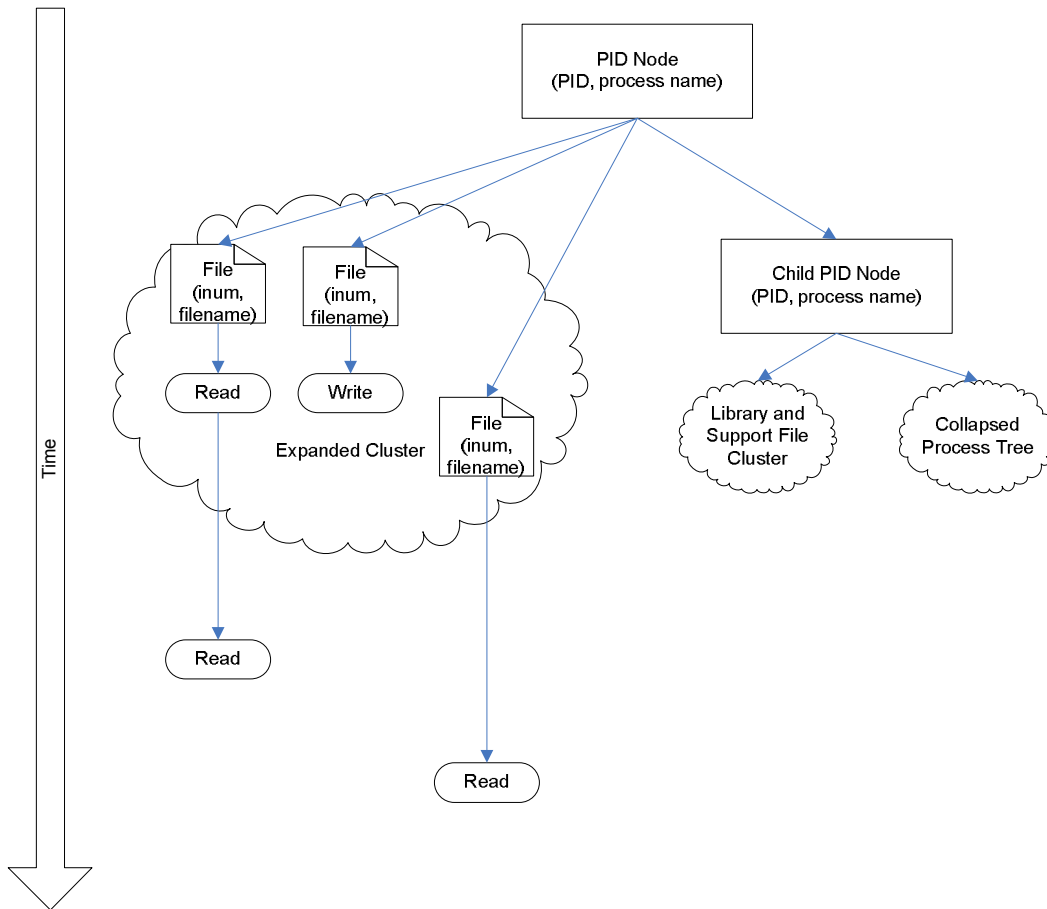


Figure 26. Visualization concept

For our research, we chose to target program-file access behavior for automatic accentuation. Using the program-file relationship database, we can identify the necessary library and support files required by each program for execution. These files are then displayed as a cluster on our overall visualization. Should any particular program access a different library file or one that is not in the order specified as by the time ordered database, then this program will be expanded and brought to the attention of the user. Also, any program that modifies a declared sensitive file will also be noted for further investigation.

Our goal for propagation is to judge the extent of influence that any particular file or process has on the entire system. This is also a way for the user to quickly estimate the extent of damage caused by an intrusion should a file or program be later determined as

malicious. Once a specific PID or file has been marked on the tree as being tainted, propagation takes places using the algorithm in figure 27.

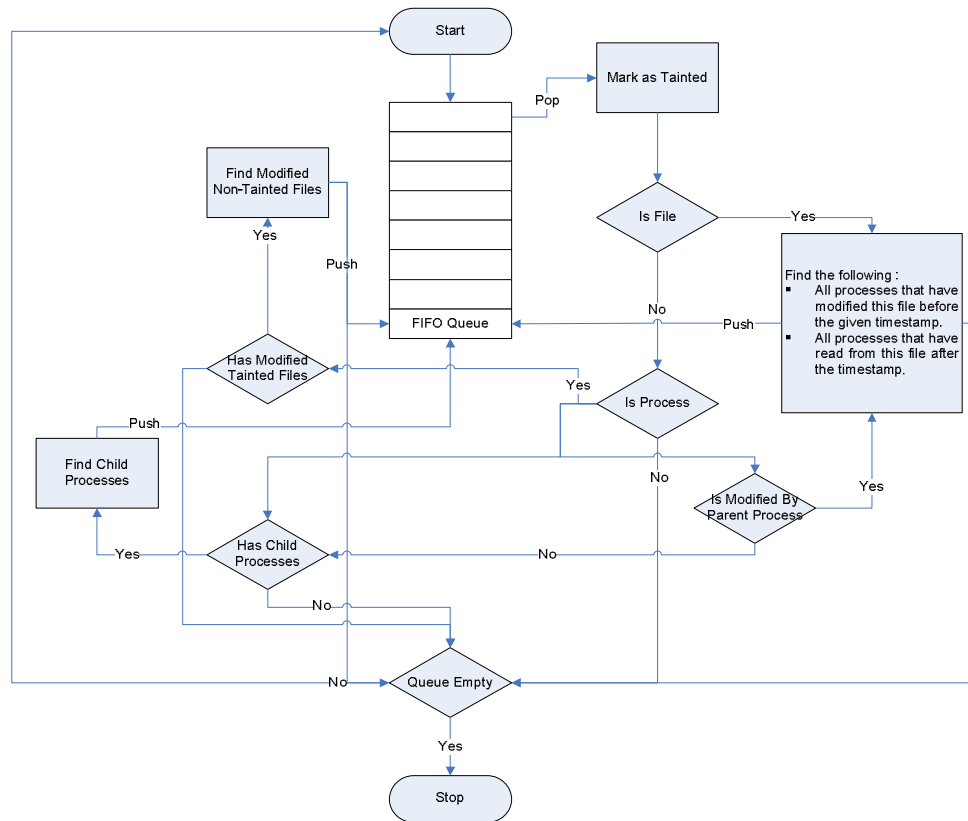


Figure 27. Propagation Algorithm

This algorithm begins by popping the top item for processing. If the object is a file, then any process that has modified the file as well as any process that has read from the file will be pushed onto the queue. If the object is a process, three checks are made. First, if the process has modified any tainted files, then any file modified by the program that is not marked as tainted is added to the queue. Second, any parent process that has modified the current process will be added to the queue. Third, any child processes

created by this process is added to the queue. This algorithm terminates under two conditions, if the queue is empty, and if the distance to origin value has been exceeded.

To terminate this propagation, we use a value called distance to origin. This value is specified by the user and is defined as the number of links between the current object to the original marked source. For example, the process that directly modified the marked tainted file is considered to have a distance of 1. The parent process or the files modified by that marked process have a distance value of 2. Once this distance value grows beyond a limit, then the current examined item is dropped from the queue.

7.3 Results

For our experiment, we obtained the data from a modified 2.6.20 kernel running Ubuntu Linux. During the monitoring process, we executed several common programs, such as Firefox, gedit, cp, and cat, and several Linux viruses.

Figure 28 shows our process tree on the left and a view of the same tree zoomed in on a particular file on the right. When an object is selected by clicking on an object name, it will be highlighted as light green. All parent nodes within the tree are displayed as dark green.

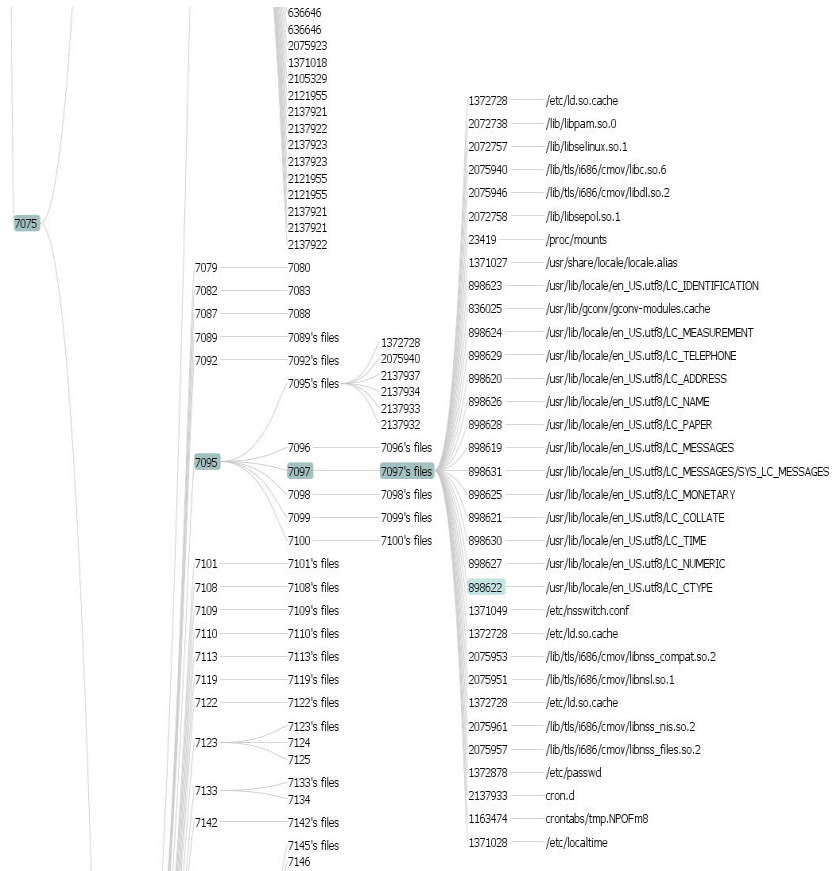


Figure 28. Process Tree

Figure 29 shows our selection of the “/usr/lib/locale/en_US.utf8/LC” file package (a commonly used set of files) and the subsequent accentuation of all such packages on the process tree. Using file package selection quickly allows users to identify when and where programs perform actions on those files, such as reading or modifying. Highlighting file packages also allows users to identify programs that may not have been executed through normal channels or was renamed in order to disguise the program.

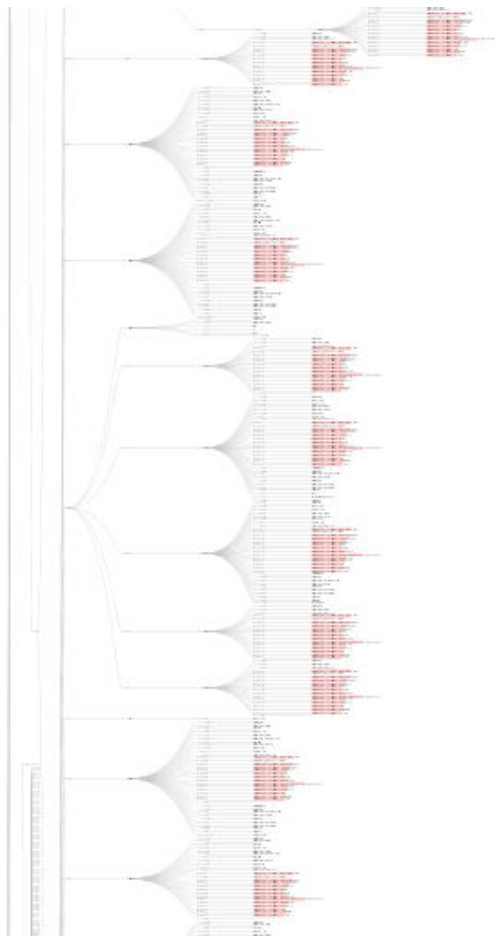


Figure 29. File package highlighting

Figure 30 shows the results of marking a file as tainted and propagating the results through the process tree. Everything marked as red is at risk for potential infection, and a system restore must insure that every file marked thusly is examined. Further improvements to the taint propagation algorithm can trim down the number of files to be examined by the taint propagation.

CHAPTER 8

CONCLUSIONS AND RECOMMENDATIONS

We have presented methods to monitor encrypted programs to allow their execution yet preserving the security of the system. Specifically, we have focused on implementing on-demand system call monitoring for developing user trust in black-box programs. A black-box program inherently does not start with a high level trust due to its multi-layer encryption along with unpredictable behavior. By implementing system call monitoring and a behavior and policy based intrusion detection system, it is possible for users to execute such software while safe with the knowledge that their sensitive information remains secure.

We believe that integrating the monitoring aspect directly into the system calls makes it significantly more difficult for a program to disable security or compromise a system without being detected. The trust building system with program policies is flexible enough to allow program execution yet can proactively prevent sensitive information from being accessed without user permission. Using this method, we have successfully monitored an encrypted program, detected when a program was exploited, and tracked the activities of several viruses on the system.

The data obtained through program monitoring contains a wealth of additional information that was not supplied by previous methods of system call tracing. Using the data gathered, we have demonstrated the ability to forensically reconstruct prior system behavior based on the file system journal in case the system call monitoring becomes inactive or is disabled. We have also shown the ability to obtain more detailed program signatures based on file access and time based dependencies and collected several new behavior signatures that can be used to uniquely identify the program.

Finally, as a part of our data presentation, we have introduced a method to visually represent the data gathered with system call monitoring in an intuitive manner.

This visual representation allows the user to quickly identify executions of programs on the process tree, isolate programs that access certain files, and to propagate the effects of a tainted program or file for an initial estimate of the extent of damage on the system.

Currently, there are not many encryption tools and programs that operate on the Linux operating system due to lack of demand. However, we feel that the general principle and technique behind our approach remains the same with or without encryption in place. Our reasoning behind this argument is that no matter what type of program is being executed, its interactions with the system must pass through the system calls and thus will be under the scrutiny of our monitoring system. Since trust development and intrusion detection is designed to be successful no matter what type of program is being executed on the system, we feel that detection of anomalous behavior of programs by our system is possible regardless of the encryption strength of these programs.

8.1 Future Work

Based on our work in building a system for monitoring encrypted programs, we have identified several areas of future research.

8.1.1 System call security

In our current work, we have not examined the countermeasures available to defeat replacement of system calls or system call tables by an attacker. Although it is possible for our monitoring system to detect this action taking place, once the system calls are compromised, all further data collection will stop.

8.1.2 Automatic Policy Generation

We have not investigated any method of automatically generating a policy that provides the lowest number of system calls required for successful program execution. Currently, the system will simply log the addition of the policy to the program policy list

unless the policy to be included requires user permission. A method for the automatic population of the policy list before program execution can be examined.

8.1.3 Machine Learning for Program Behavior Database

In our work presented in our publication [2], we discussed the possibility of using machine learning to assist in the creating and identification of program behaviors based on gathered system call data. Further research should be conducted on machine learning.

8.1.4 Performance Optimizations

In addition to system integration, there are many ways to optimize our work. This work was deliberately proof of concept only, thus a study of optimization issues would be beneficial.

8.1.5 Better Visual Presentation

Currently, our visualization representation is not an optimal approach as the large amounts of data make it difficult for users to examine. Better visual representations or interfacing methods should be examined to assist the users in the analysis of the process data.

REFERENCES

- [1] A. Singh, "Understanding Apple's Binary Protection in Mac OS X", October 2006. Available: <http://www.osxbook.com/book/bonus/chapter7/binaryprotection>
- [2] Skype, <http://www.skype.com/>
- [3] Burneye, <http://www.xsec.org/index.php?module=tools&act=view&type=8&id=14>
- [4] EXECryptor, <http://www.strongbit.com/execryptor.asp>
- [5] UPX, <http://upx.sourceforge.net>
- [6] Shiva, <http://www.securereality.com.au>
- [7] Pfleeger, C. P. and Pfleeger, S. L., Security in Computing. Upper Saddle River, NJ: Prentice Hall, 2003.
- [8] Grizzard J., Levine J., and Owen H., "Re-establishing trust in compromised systems: recovering from rootkits that Trojan the system call table", In Proceedings of 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 2004, pp. 369-384.
- [9] Banerjee S., Mattmann C., Medvidovic N., Golubchik L., "Leveraging architectural models to inject trust into software systems", in ACM Software Engineering for Secure Systems, July 2005.
- [10] Plattner B., Nievergelt J., "Special Feature: Monitoring Program Execution: A Survey", In Volume 14, Issue 11, IEEE Computer Society press, November 1981
- [11] Younan Y., Joosen W., Piessens F., "Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures", in Technical Report CW385 by Department Computerwetenschappen, Katholieke Universiteit Leuven
- [12] Baset S., Schulzrinne H., "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol", In Columbia Technical Report CUCS-039-04, September 2004.
- [13] Hofmeyr S., Forrest S., Somayaji A., "Intrusion detection using sequences of system calls", In Journal of Computer Security, August 1998, pp. 151-180.
- [14] Bondi P., Desclaux F., "Silver Needle in the Skype", BlackHat Europe Conference, March 2006. Available: <http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf>

- [15] Forrest S., Hofmeyr S., Somayaji A., Longstaff T., "A sense of self for UNIX processes", in IEEE Symposium on Security and Privacy, Oakland CA, May 1996, pp. 120-128
- [16] Warrender C., Forrest S., Pearlmuter B., "Detecting Intrusions using System Calls: Alternative Data Models", In Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium, 1999, pp.133-145
- [17] Lee J., Kim H., "Implementation and Overhead Analysis of a Log-Based Intrusion Recovery Module on Linux File System", In 2004 Joint Conference of the 10th Asia-Pacific Conference, September 2004, pp. 548 – 552.
- [18] The HoneyNet Project, "Know Your Enemy: Sebek", Available: <http://HoneyNet.org/papers/sebek.pdf>
- [19] HoneyNet Project. "Know Your Enemy: HoneyNets", Available: <http://www.HoneyNet.org/papers/HoneyNet/>
- [20] Holz T., Rayna F., "Defeating honeypots: System Issues", available: <http://www.securityfocus.com/infocus/1826>, <http://www.securityfocus.com/infocus/1828>
- [21] Oudot L., Holz T., "Defeating honeypots: Network Issues", available: <http://www.securityfocus.com/infocus/1803>, <http://www.securityfocus.com/infocus/1805>
- [22] Giffin J., Dagon D., Jha S., Lee W., Miller B., "Environment-sensitive intrusion detection.", in 8th International Symposium on Recent Advances in Intrusion Detection, Seattle, Washington, September 2005.
- [23] Warrender C., Forrest S., Pearlmuter B., "Detecting intrusions using system calls: Alternative data models." In IEEE Symposium on Security and Privacy, Oakland, California, May 1999.
- [24] SystemTap: Available: <http://sourceware.org/systemtap>
- [25] Jaeger T., Sailer R., Zhang X., "Analyzign Integrity Protection in the SELinux Example Policy", in Proceedings of the 12th USENIX Security Symposium,
- [26] Hanson C., "SELinux and MLS: Putting the Pieces Together"
- [27] Wright C., Cowan C., Smalley S., "Linux Security Modules: General Security Support for the Linux Kernel", in Proceedings of the Eleventh USENIX Security Symposium, August 2002.
- [28] Smalley S., Fraser T., "A Security Policy Configuration for the Security-Enhanced Linux"

- [29] Loscocco P., Smalley S., “Integrating Flexible Support for Security Policies into the Linux Operating System”
- [30] GDB: The GNU Project Debugger, Available: <http://sourceware.org/gdb>
- [31] Younan Y., Joosen W., Piessens F., “Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures”, Report CW 386, July 2004
- [32] Clowes S., “A Security Microcosm Attacking and Defending Shiva”, Blackhat Asia Conference, December 2003. Available: <http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-clowes.pdf>
- [33] Mehta N., Clowes S., “Shiva: Advances in ELF Binary Protection”, in CanSecWest, 2003
- [34] Schallner M., “Beginners Guide to Basic Linux Anti Anti Debugging Techniques”, CodeBreakers magazine, Security & Anti-Security – Attack & Defense, vol 1, iss 2, 2006
- [35] Dornseif M., Holz T., Klein C., “NoSEBrEaK – Attacking Honeynets”, in Proceedings of the 2004 IEEE Workshop on Information Assurance and Security, Westpoint NY, June 2004
- [36] Linux System Calls for HLA Programmers
- [37] Strunk J., Goodson G., Pennington A., Soules C., Ganger G.. “Intrusion Detection, Diagnosis, and Recovery with Self-Securing Storage”, in CMU SCS Technical Report CMU-CS-02-140, May 2002.
- [38] ELF_NEL.A Description and Solution. Available: http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=ELF_NEL.A
- [39] Ko C., Ruschitzka M., Levitt K., “Execution monitoring of security-critical programs in distributed systems: a specification-based approach”, in IEEE Symposium on Security and Privacy, Oakland CA, May 1997, pp. 175-187
- [40] Rutkowski J., “Advanced Windows 2000 Rootkit Detection (Execution Path Analysis)”, in BlackhatUSA – Las Vegas 2003, July 2003
- [41] Garfinkel S., “Anti-Forensics: Techniques, Detection and Countermeasures”, in the 2nd International Conference on i-Warfare and Security (ICIW), Monterey, CA, March 2007
- [42] Wei X., Ju J., “SCR algorithm: saving/restoring states of file systems”, in Operating Systems/Review, January 1999, pp. 33

- [43] “Appropriate Uses for Sqlite”, Sqlite DOI: <http://www.sqlite.org/docs.html> Accessed January 22, 2008
- [44] Ubuntu <http://www.ubuntu.com/products/whatisubuntu>
- [45] Prefuse: Information Visualization Toolkit, <http://prefuse.org/doc/faq>
- [46] Grizzard, J., “Towards Self Healing Systems: Re-establishing Trust in Compromised Systems”
- [47] Rosenberg, J., “How Debuggers Work: Algorithms, Data Structures, and Architecture”, Wiley, 1996
- [48] Larus, J., “Efficient program tracing”, in Computer, May 1993
- [49] ptrace <http://www.linuxmanpages.com/man2/ptrace.2.php>
- [50] Schwarz, B., Debray, S., Andrews, G., “Disassembly of executable code revisited”, in Proceedings of the Ninth Working Conference on Reverse Engineering, 2002
- [51] Linn, C., Debray., S, “Obfuscation of executable code to improve resistance to static disassembly”, in Proceedings of the 10th ACM conference on Computer and communications security, 2003

PUBLICATIONS

[1] Xia Y., Fairbanks K., Owen H., “Establishing Trust in Black-box Programs”, in Proceedings of the IEEE 2007 Southeast Conference, March 2007

[2] Xia Y., Fairbanks K., Owen H., “A Program Behavior matching Architecture for Probabilistic File System Forensics.,” ACM SIGOPS Operating Systems Review special issue on Computer Forensics, April 2008.

[3] Xia Y., Fairbanks K., Owen H., “Visual Analysis of Program Flows with Propagation.,” Vizsec 2008

[4] Fairbanks K., Lee C., Xia Y., Owen H., “TimeKeeper: A Metadata Archiving Method for Honeypot Forensics.,” 8th Annual IEEE SMC Information Assurance Workshop. West Point, NY. June 2007.

APPENDIX A – LINUX SYSTEM CALLS

```
1 ENTRY(sys_call_table)
2 .long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting */
3 .long sys_exit
4 .long sys_fork
5 .long sys_read
6 .long sys_write
7 .long sys_open /* 5 */
8 .long sys_close
9 .long sys_waitpid
10 .long sys_creat
11 .long sys_link
12 .long sys_unlink /* 10 */
13 .long sys_execve
14 .long sys_chdir
15 .long sys_time
16 .long sys_mknod
17 .long sys_chmod /* 15 */
18 .long sys_lchown16
19 .long sys_ni_syscall /* old break syscall holder */
20 .long sys_stat
21 .long sys_lseek
22 .long sys_getpid /* 20 */
23 .long sys_mount
24 .long sys_oldumount
25 .long sys_setuid16
26 .long sys_getuid16
27 .long sys_stime /* 25 */
28 .long sys_ptrace
29 .long sys_alarm
30 .long sys_fstat
31 .long sys_pause
32 .long sys_utime /* 30 */
33 .long sys_ni_syscall /* old stty syscall holder */
34 .long sys_ni_syscall /* old gtty syscall holder */
35 .long sys_access
36 .long sys_nice
37 .long sys_ni_syscall /* 35 - old ftime syscall holder */
38 .long sys_sync
39 .long sys_kill
40 .long sys_rename
41 .long sys_mkdir
42 .long sys_rmdir /* 40 */
43 .long sys_dup
44 .long sys_pipe
```

```

45 .long sys_times
46 .long sys_ni_syscall /* old prof syscall holder */
47 .long sys_brk /* 45 */
48 .long sys_setgid16
49 .long sys_getgid16
50 .long sys_signal
51 .long sys_geteuid16
52 .long sys_getegid16 /* 50 */
53 .long sys_acct
54 .long sys_umount /* recycled never used phys() */
55 .long sys_ni_syscall /* old lock syscall holder */
56 .long sys_ioctl
57 .long sys_fcntl /* 55 */
58 .long sys_ni_syscall /* old mpx syscall holder */
59 .long sys_setpgid
60 .long sys_ni_syscall /* old ulimit syscall holder */
61 .long sys_olduname
62 .long sys_umask /* 60 */
63 .long sys_chroot
64 .long sys_ustat
65 .long sys_dup2
66 .long sys_getppid
67 .long sys_getpgrp /* 65 */
68 .long sys_setsid
69 .long sys_sigaction
70 .long sys_sgetmask
71 .long sys_ssetmask
72 .long sys_setreuid16 /* 70 */
73 .long sys_setregid16
74 .long sys_sigsuspend
75 .long sys_sigpending
76 .long sys_sethostname
77 .long sys_setrlimit /* 75 */
78 .long sys_old_getrlimit
79 .long sys_getrusage
80 .long sys_gettimeofday
81 .long sys_settimeofday
82 .long sys_getgroups16 /* 80 */
83 .long sys_setgroups16
84 .long old_select
85 .long sys_symlink
86 .long sys_lstat
87 .long sys_readlink /* 85 */
88 .long sys_uselib
89 .long sys_swapon
90 .long sys_reboot

```

```

91  .long old_readdir
92  .long old_mmap      /* 90 */
93  .long sys_munmap
94  .long sys_truncate
95  .long sys_ftruncate
96  .long sys_fchmod
97  .long sys_fchown16 /* 95 */
98  .long sys_getpriority
99  .long sys_setpriority
100 .long sys_ni_syscall /* old profil syscall holder */
101 .long sys_statfs
102 .long sys_fstatfs  /* 100 */
103 .long sys_ioperm
104 .long sys_socketcall
Sub-function system calls
#define SYS_SOCKET      1      /* sys_socket(2)      */
#define SYS_BIND        2      /* sys_bind(2)        */
#define SYS_CONNECT     3      /* sys_connect(2)     */
#define SYS_LISTEN      4      /* sys_listen(2)      */
#define SYS_ACCEPT      5      /* sys_accept(2)      */
#define SYS_GETSOCKNAME 6      /* sys_getsockname(2) */
#define SYS_GETPEERNAME 7      /* sys_getpeername(2) */
#define SYS_SOCKETPAIR  8      /* sys_socketpair(2)  */
#define SYS_SEND        9      /* sys_send(2)        */
#define SYS_RECV        10     /* sys_recv(2)        */
#define SYS_SENDFROM    11     /* sys_sendto(2)     */
#define SYS_RECVFROM    12     /* sys_recvfrom(2)   */
#define SYS_SHUTDOWN    13     /* sys_shutdown(2)   */
#define SYS_SETSOCKOPT  14     /* sys_setsockopt(2) */
#define SYS_GETSOCKOPT  15     /* sys_getsockopt(2) */
#define SYS_SENDMSG     16     /* sys_sendmsg(2)    */
#define SYS_RECVMSG     17     /* sys_recvmsg(2)    */
105 .long sys_syslog
106 .long sys_setitimer
107 .long sys_getitimer /* 105 */
108 .long sys_newstat
109 .long sys_newlstat
110 .long sys_newfstat
111 .long sys_uname
112 .long sys_iopl      /* 110 */
113 .long sys_vhangup
114 .long sys_ni_syscall /* old "idle" system call */
115 .long sys_vm86old
116 .long sys_wait4
117 .long sys_swapoff  /* 115 */
118 .long sys_sysinfo

```

```

119 .long sys_ipc
120 .long sys_fsync
121 .long sys_sigreturn
122 .long sys_clone /* 120 */
123 .long sys_setdomainname
124 .long sys_newuname
125 .long sys_modify_ldt
126 .long sys_adjtimex
127 .long sys_mprotect /* 125 */
128 .long sys_sigprocmask
129 .long sys_ni_syscall /* old "create_module" */
130 .long sys_init_module
131 .long sys_delete_module
132 .long sys_ni_syscall /* 130: old "get_kernel_syms" */
133 .long sys_quotactl
134 .long sys_getpgid
135 .long sys_fchdir
136 .long sys_bdflush
137 .long sys_sysfs /* 135 */
138 .long sys_personality
139 .long sys_ni_syscall /* reserved for afs_syscall */
140 .long sys_setfsuid16
141 .long sys_setfsgid16
142 .long sys_llseek /* 140 */
143 .long sys_getdents
144 .long sys_select
145 .long sys_flock
146 .long sys_msync
147 .long sys_readv /* 145 */
148 .long sys_writev
149 .long sys_getsid
150 .long sys_fdatasync
151 .long sys_sysctl
152 .long sys_mlock /* 150 */
153 .long sys_munlock
154 .long sys_mlockall
155 .long sys_munlockall
156 .long sys_sched_setparam
157 .long sys_sched_getparam /* 155 */
158 .long sys_sched_setscheduler
159 .long sys_sched_getscheduler
160 .long sys_sched_yield
161 .long sys_sched_get_priority_max
162 .long sys_sched_get_priority_min /* 160 */
163 .long sys_sched_rr_get_interval
164 .long sys_nanosleep

```

```

165 .long sys_mremap
166 .long sys_setresuid16
167 .long sys_getresuid16 /* 165 */
168 .long sys_vm86
169 .long sys_ni_syscall /* Old sys_query_module */
170 .long sys_poll
171 .long sys_nfsservctl
172 .long sys_setresgid16 /* 170 */
173 .long sys_getresgid16
174 .long sys_prctl
175 .long sys_rt_sigreturn
176 .long sys_rt_sigaction
177 .long sys_rt_sigprocmask /* 175 */
178 .long sys_rt_sigpending
179 .long sys_rt_sigtimedwait
180 .long sys_rt_sigqueueinfo
181 .long sys_rt_sigsuspend
182 .long sys_pread64 /* 180 */
183 .long sys_pwrite64
184 .long sys_chown16
185 .long sys_getcwd
186 .long sys_capget
187 .long sys_capset /* 185 */
188 .long sys_sigaltstack
189 .long sys_sendfile
190 .long sys_ni_syscall /* reserved for streams1 */
191 .long sys_ni_syscall /* reserved for streams2 */
192 .long sys_vfork /* 190 */
193 .long sys_getrlimit
194 .long sys_mmap2
195 .long sys_truncate64
196 .long sys_ftruncate64
197 .long sys_stat64 /* 195 */
198 .long sys_lstat64
199 .long sys_fstat64
200 .long sys_lchown
201 .long sys_getuid
202 .long sys_getgid /* 200 */
203 .long sys_geteuid
204 .long sys_getegid
205 .long sys_setreuid
206 .long sys_setregid
207 .long sys_getgroups /* 205 */
208 .long sys_setgroups
209 .long sys_fchown
210 .long sys_setresuid

```

```

211 .long sys_getresuid
212 .long sys_setresgid /* 210 */
213 .long sys_getresgid
214 .long sys_chown
215 .long sys_setuid
216 .long sys_setgid
217 .long sys_setfsuid /* 215 */
218 .long sys_setfsuid
219 .long sys_pivot_root
220 .long sys_mincore
221 .long sys_madvise
222 .long sys_getdents64 /* 220 */
223 .long sys_fcntl64
224 .long sys_ni_syscall /* reserved for TUX */
225 .long sys_ni_syscall
226 .long sys_gettid
227 .long sys_readahead /* 225 */
228 .long sys_setxattr
229 .long sys_lsetxattr
230 .long sys_fsetxattr
231 .long sys_getxattr
232 .long sys_lgetxattr /* 230 */
233 .long sys_fgetxattr
234 .long sys_listxattr
235 .long sys_llistxattr
236 .long sys_flistxattr
237 .long sys_removexattr /* 235 */
238 .long sys_lremovexattr
239 .long sys_fremovexattr
240 .long sys_tkill
241 .long sys_sendfile64
242 .long sys_futex /* 240 */
243 .long sys_sched_setaffinity
244 .long sys_sched_getaffinity
245 .long sys_set_thread_area
246 .long sys_get_thread_area
247 .long sys_io_setup /* 245 */
248 .long sys_io_destroy
249 .long sys_io_getevents
250 .long sys_io_submit
251 .long sys_io_cancel
252 .long sys_fadvise64 /* 250 */
253 .long sys_ni_syscall
254 .long sys_exit_group
255 .long sys_lookup_dcookie
256 .long sys_epoll_create

```

```

257 .long sys_epoll_ctl /* 255 */
258 .long sys_epoll_wait
259 .long sys_remap_file_pages
260 .long sys_set_tid_address
261 .long sys_timer_create
262 .long sys_timer_settime /* 260 */
263 .long sys_timer_gettime
264 .long sys_timer_getoverrun
265 .long sys_timer_delete
266 .long sys_clock_settime
267 .long sys_clock_gettime /* 265 */
268 .long sys_clock_getres
269 .long sys_clock_nanosleep
270 .long sys_statfs64
271 .long sys_fstatfs64
272 .long sys_tgkill /* 270 */
273 .long sys_utimes
274 .long sys_fadvise64_64
275 .long sys_ni_syscall /* sys_vserver */
276 .long sys_mbind
277 .long sys_get_mempolicy
278 .long sys_set_mempolicy
279 .long sys_mq_open
280 .long sys_mq_unlink
281 .long sys_mq_timedsend
282 .long sys_mq_timedreceive /* 280 */
283 .long sys_mq_notify
284 .long sys_mq_getsetattr
285 .long sys_kexec_load
286 .long sys_waitid
287 .long sys_ni_syscall /* 285 */ /* available */
288 .long sys_add_key
289 .long sys_request_key
290 .long sys_keyctl
291 .long sys_ioprio_set
292 .long sys_ioprio_get /* 290 */
293 .long sys_inotify_init
294 .long sys_inotify_add_watch
295 .long sys_inotify_rm_watch
296 .long sys_migrate_pages
297 .long sys_openat /* 295 */
298 .long sys_mkdirat
299 .long sys_mknodat
300 .long sys_fchownat
301 .long sys_futimesat
302 .long sys_fstatat64 /* 300 */

```

303 .long sys_unlinkat
304 .long sys_renameat
305 .long sys_linkat
306 .long sys_symlinkat
307 .long sys_readlinkat /* 305 */
308 .long sys_fchmodat
309 .long sys_faccessat
310 .long sys_pselect6
311 .long sys_ppoll
312 .long sys_unshare /* 310 */
313 .long sys_set_robust_list
314 .long sys_get_robust_list
315 .long sys_splice
316 .long sys_sync_file_range
317 .long sys_tee /* 315 */
318 .long sys_vmsplice
319 .long sys_move_pages
320 .long sys_getcpu
321 .long sys_epoll_pwait
322 .long sys_utimensat /* 320 */
323 .long sys_signalfd
324 .long sys_timerfd
325 .long sys_eventfd
326 .long sys_fallocate

VITA

YING H. XIA

Ying H. Xia was born in Nanjing, China. He came to the United States of America in 1989 with his parents. Ying H. Xia attended public schools in Indiana, Hawaii, Ohio, and Maryland, received a B.S. in Computer Engineering from the University of Maryland at College Park, College Park, Maryland in 2004 before coming to Georgia Tech to pursue a masters and doctorate in Electrical and Computer Engineering. In August of 2008, he completed his doctoral studies. His research interests include computer security, encryption, and network security.