10-27-2008

Studying a Virtual Testbed for Unverified Data

William Carl Bridges

Studying a Virtual Testbed for Unverified Data

By

William Carl Bridges

Thesis Committee:
Peter Lutz (Chair)
Hans-Peter Bischof
Sidney Marshall

Thesis submitted in partial fulfillment of the requirements for the
degree of Master of Science in
Computer Security and Information Assurance

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

October 27, 2008

Table of Contents

Abstract

It is difficult to fully know the effects a piece of software will have on your computer, particularly when the software is distributed by an unknown source. The research in this paper focuses on malware detection, virtualization, and sandbox/honeypot techniques with the goal of improving the security of installing useful, but unverifiable, software. With a combination of these techniques, it should be possible to install software in an environment where it cannot harm a machine, but can be tested to determine its safety. Testing for malware, performance, network connectivity, memory usage, and interoperability can be accomplished without allowing the program to access the base operating system of a machine. After the full effects of the software are understood and it is determined to be safe, it could then be run from, and given access to, the base operating system. This thesis investigates the feasibility of creating a system to verify the security of unknown software while ensuring it will have no negative impact on the host machine.

Introduction

It can be difficult to determine exactly how software will impact your system prior to installation. Many popular and useful applications are freely available from open source communities or free downloads, but they do not come with any guarantee that they will not harm your system. It is also possible that they can contain malware, whether it is there intentionally or not. This paradigm is not only applicable to software, but also patches, add-ons, and other third party "enhancement" applications. Many websites have different kinds of embedded media that require various players; it is difficult to determine if such a player may be malicious before installing it and executing the file.

To gain confidence when installing unknown software, the areas of virtualization, malware detection, and sandbox testing were thoroughly investigated. These areas offered techniques to evaluate unknown and unverified software before installing it. Projects such as SpyProxy[6] help to prevent malicious web sites from installing malware on a user's machine by tunneling the website through a virtual machine then studying the effects it has on the virtual machine before allowing the host system to access the website. While SpyProxy uses a remote machine running a virtual machine similar to your host system to diagnose potentially malicious web sites, projects such as Manitou[1] and Solitude[5] harness local virtualization techniques to help isolate and diagnose potentially malicious software. These projects show how virtualization can be used so that software can be installed in a secure environment, where it cannot harm the host system, so that it can be fully evaluated to determine its true intent before installing it on the host system. Virtualization has progressed so that it can be harnessed by home users in an efficient way to provide a useful testbed.

While virtualization has been used extensively to test unknown software, the malware detection techniques have evolved from initial signature-based techniques. Heuristic techniques are becoming more feasible at both a host and a network level. Many innovative techniques[2][10][12][15] have been created to monitor or determine the execution sequence of code to determine if it is malware. These techniques have proven to have a low false positive rate while maintaining a high degree of detection accuracy without sacrificing efficiency. To support these new techniques, signature-based scanning techniques have also improved[9] to detect viruses faster and more effectively than before.

Honeypots have been used to combine virtualization and malware detection to gather useful statistical information regarding current malware on the local network and capture malicious binaries.[23] A honeypot is created to look like a tempting target for malware. No legitimate access should be made to the honeypot, so any access to the honeypot can be defined as malicious. This distinction makes it very easy to locate malware – anything interacting with the system can be considered malicious.[24] The intentional vulnerabilities created in a honeypot help to determine what kinds of malware are circulating through a network. In addition to determining what is attacking a network, honeypots can collect malware binaries for further analysis.[45] This can be used to determine if the anti-virus(AV) software used is adequate for the network by scanning the binaries to see if the AV engine detects them as malware.[46] They can also be analyzed further to see their impact on the system and their potential payloads. Any unidentified binaries can be sent to AV vendors to ensure that signatures are quickly created. The honeypot phenomenon has also spurred the creation of novel malware detection approaches. Combining virtualization for quickly deployable systems allows for the Virtual Machine(VM) to be monitored directly through the Virtual Machine Monitor(VMM).[52] This allows for kernel level trapping and event handling to yield fine-grain controls over malware.

Combining new malware detection strategies, virtualization techniques, and honeypot and sandbox technologies can lead to novel security designs based on the interface being secured. Effective

security strategies can be developed based on the specific interfaces they are securing because the attack surface is well known. Instead of trying to make a general security system that covers every attack vector of a computer, a modular security system with differing design requirements based on the interface could achieve successful computer security.

Review of the Literature

The classical technique of malware detection is to use signature-based algorithms. These attempt to identify a unique character sequence in malware, then scan the computer for that unique sequence. If it is found, it is determined to be malware. Smobile Systems[9] worked on a method to make signature-based scanning fast and accurate enough to work on mobile devices. A signature-cut method was used to find the smallest substring of a signature to use as a filter-hash. This filter-hash was more effective than scanning for the entire signature and eliminates legitimate files from further scrutiny. If the filter-hash finds a match, the full signature-based hash process is used to determine if it is malware or not. By using a filter-hash, they eliminated the overhead of checking each signature against every potential piece of malware. Heuristic detection is another malware detection technique that has received much research. A heuristic technique monitors the variables of a computer system long enough that it can draw a statistical baseline for any given time and day. This information is used to create a model of what normal working conditions are. If the current conditions stray far enough from this model, then the system needs to be checked for compromise. Heuristic techniques take advantage of probabilistic models.[11] The threshold of malware detection can be adjusted by the administrator to be more or less sensitive. The further a monitored system deviates from the statistical average, the more probable it is that malware is present on the system. Due to heuristic systems being probability-based, their rate of false positives are higher than other malware detection techniques. A false positive occurs when the system detects an abnormality as malware, but in actuality that abnormality is caused by a legitimate source.

Instead of scanning or monitoring for problems, the Intelligent Malware Detection System[10] (IMDS) parses binaries to determine if their execution would be malicious. An accurate assumption was made that the executable properties of malware and legitimate software are measurably different. In order for malware to compromise a system, a vulnerability must be exploited, usually involving spawning new processes and hooking in to other modules. Depending on the point of execution, elevation of privileges can also be detected. The IMDS looks at Windows PE executables – they are formatted to be run on any Windows platform and are an attractive format for malware creators due to their interoperability. The IMDS parses the binary and generates a database of the execution sequence. This database is then data mined and compared to legitimate software to determine if the execution sequence is legitimate or not. The API calls of malicious and legitimate software differed enough that IMDS was able to accurately detect 92% of unknown malware and all known malware. The next closest malware detection solutions tested were McAfee and Kaspersky, which only detected 75.3% and 78% of malware respectively.

Virtualization has created a new platform that can be accessed for malware detection. A virtual machine (VM) offers many advantages to malware detection. Running a VM on a host allows malware detection software to have a kernel-like view of the host operating system. This allows for semantic view reconstruction[2] that avoids distortion from rootkits. In order for the VM to work properly, all contents need to be stored in some manner on the host machine. These contents are then reconstructed to create a virtual environment. A malware detection system on the host machine could reconstruct the data the same way the virtual machine would to have unfiltered access to any of the machines data. This has an advantage over looking at sensitive areas from within the VM because nothing within the VM can alter the semantic view. In addition to accurate semantic view reconstruction, virtualization offers an environment where software can be tested without harming the host machine. If a VM is compromised, it can be taken down and set back to its original state. The SpyProxy[6] project takes advantage of this capability by tasking a VM to examine the flow of data between a user's web browser

and the web server. When a web page is accessed, the cache is checked to see if the page has already been accessed. If it has not been loaded, and contains more than standard HTML code, the web page is loaded into a VM. The VM is then monitored for any events that would indicate malicious behavior from the web server. Any request to download additional software is granted so that it can be tested as well. The web pages are examined from their root directory down to ensure safety. If no alerts are triggered, the web page is then sent to the user; if anything malicious occurs, access to the page is denied.

The use of virtualization for malware detection allows for semantic view reconstruction and interesting testbed experiments, but also creates a platform for controlled program execution. Some virtualization techniques accomplish virtualization by implementing a layer between the virtualized system and the host machine. This layer intercepts system calls between the virtual and host layers; they are interpreted going both ways so that either layer can understand the other. This additional layer can be modified to examine system calls before passing them on to the host machine to be implemented. The Manitou[1] project controls program execution in a virtualization environment by controlling the paged memory and per-page executable and writable bits. Before an instruction is executed, it must pass through Manitou. If Manitou does not want the instruction to be executed, it disables the execution bit of the request. This causes a page fault to occur that is returned to, and dealt with, in the virtualization layer. This generally causes the program to crash or be aborted. This process allows Manitou to monitor different processes and control what is permitted to execute. Even if malware was to compromise a machine, Manitou could be used to disable execution privileges so that it could not do any harm.

Instead of taking advantage of virtualization by implementing execution controls in a layer below the virtualized environment, Solitude[5] uses an isolation file system (IFS). The IFS can be generated on demand based on the needs of the user. When an application is committed to an IFS, everything is done within that virtualized file system. The application cannot interact with things outside of its own IFS. More than one application can be joined in a single IFS if they need to interact with each other. There are also write methods created so that data from the IFS can be written directly to the host's hard drive. These methods allow for information to pass through the IFS in a user-controlled manner in the event that it needs to be shared or has been verified in some manner. If something within an IFS is considered problematic, the IFS can be removed and everything within that IFS will be removed as if it had never existed on the computer. If malware were to spread from an application in the IFS all of the writes the malware executed would be contained within the IFS, so when the IFS was removed, all traces of the malware would also be removed.

A honeypot can combine virtualization and malware detection to allow malware to attack a vulnerable emulated environment. The malware detection system can detect the attack and collect data to further benefit the malware detection system or the community the honeypot was deployed in. Honeypots allow for the collection of real-time statistical information regarding the network - where attacks are coming from, what services are being targeted, and what types of payloads are being delivered. For purposes of software verification, honeypots offer valuable insight because they allow an attack to occur, but closely monitor and contain the attack. Sensors[52] can be placed in the Virtual Machine Monitor[52] (VMM) that monitor the events of the virtual environment for malicious behavior. The VMM is a layer between the virtualized environment and the host operating system. This gives it insight into the virtualization environment. Much malware has similar attacks, such as forking a new process and loading a malicious service. Traps can be placed at these critical areas to trigger an event handler. For example, a trap could be placed in the exec() process that triggers an event handler to look at the process being creating before allowing it to be spawned. The trap and associated event handler is a sensor. More specific malware can also be detected by placing sensors in

rarely accessed areas that the malware depend on. This does not increase computational overhead significantly because the traps are not accessed very often. However, it is very useful for detecting malware because the semantic view of the virtual environment can be examined before any further execution is permitted. The event handlers can be used to trigger warnings, stop execution, or add dynamic traps in other places for further analysis.

The Nepenthes[45] honeypot is a hybrid approach to honeypot technology. Instead of using a purely high or low interaction design, it uses a customizable modular design. This allows for vulnerable services to be emulated in a controlled manner. A service could be emulated in a very vague way until something tries to communicate with it. It can then modify itself based on the needs of the network communication. For example, a general FTP service could be emulated, then based on the network interaction, it could respond as a Win32 protocol or a Unix based protocol. The modular design allows for the creation of different emulation environments. Various services can be emulated based on the needs of the user. This can be applied to software verification by emulating the type of environment an application is trying to access. The vulnerabilities can be included in the emulated environment in a way that triggers alarms if they are taken advantage of. If the software tries to manipulate the environment in a malicious way, it can be classified as malware without giving it access to the host operating system, and a low amount of resource usage in the emulation environment is maintained. The ability to create customized modular emulation environments to mimic a desired vulnerability allows for the Nepenthes honeypot to remain efficient, useful, and scalable.

Malware

Malware Defined

Malware is any piece of software that affects your computer in a non-consensual way. This can range from viruses that negatively impact your computer to adware that impacts your computer only with minimal unwanted resource usage. The intent of the software is also taken into consideration when determining if it is malware. For instance, consensual software with the intent of doing one thing that behaves irregularly and accidentally does damage to the computer is not considered malware. A piece of defective software that contains harmful bugs is not considered malware if the software's usage is for legitimate, agreed upon reasons. [13]

Types of Malware

Malware can be classified in many different ways, including, but not limited to, propagation techniques, payloads, in-memory locations, or infection strategies. It is difficult to create well-defined categories for malware because the programs often overlap, and some pieces of malware can fall into more than one category, or define a new subcategory. Regardless of the problems, it is still useful to create a nomenclature for malware so that professionals can communicate about and discuss pieces without having to describe every characteristic of every piece of malware. In Peter Szor's book "The Art of Computer Virus Research and Defense"[16] the following well-known and accepted malware categories are discussed.

Viruses

A virus is the most commonly misused malware category. Many people refer to any piece of malware as a virus. However, a virus is specifically any code that replicates itself. A virus does not execute itself until the program that it has infected is executed.[27] This leads to viruses targeting executable files. Depending on the virus, it can evolve when replicating to create new generations of itself. It can infect host files or other areas of the system. It is also possible for a virus to modify existing references to objects in order to take control of them, and then multiply again. The goal of a virus is to infect a system, which can have destructive or irreparable effects on the system.

An early virus known as the Jerusalem virus (Jerusalem.B) was detected in 1991.[27] It is a good example of the destructive nature of a virus. It infected any file with an exe, sys, and com extension. As the files were infected and ran, they increased in size as a result of the virus. When the system clock turned to Friday 13th, the virus deleted all files on the computer. A virus can harm a computer but it does not have any financial benefit to the creator. As shown in Figure 1, viruses have been declining as more lucrative types of malware are focused on. Currently, Panda Security[28] finds that viruses only make up 11.37% of malware traffic.

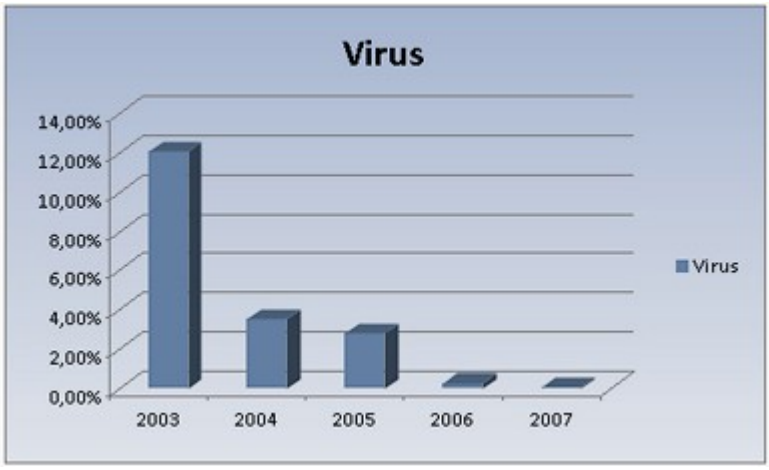


Figure 1: The number of viruses seen each year. [27]

Worms

A worm is a virus on the network level. Instead of replicating itself on an isolated machine, it replicates itself over a network. A worm can automatically execute itself in a manner to move from system to system, or wait for a user-initiated method of replication. An example of a worm is the *mass-mailer* type. It propagates through a user's email, sending itself to anyone accessible via the email system. The W32/MSNBot.D.worm worm was detected on September 9, 2008.[30] It spreads to users via e-mail through a file with a butterfly icon. Once a system is infected, it gathers personal information about the user such as user names and passwords. Unlike a virus, a worm is a standalone executable instead of a piece of code embedded within a file. Worms have evolved from being used by malware creators to gain notoriety to being used for financial gain. Of worms detected in 2007, 34.96% of them were used to create *botnets*.[29] A botnet is a collection of compromised computers that can be accessed for malicious use by an attacker. These computers have been previously compromised and left in a state that allows an attacker to control them remotely.

There are many subcategories of worms, like the mass-mailer previously discussed. Another subcategory is the *octopus*. An octopus is a type of worm that exists as different pieces on more than one machine. One function might be on one computer, while other functions are on various other computers in a network. The total functionality of the worm is not realized until they all communicate with each other and harness the capabilities of the separate pieces working together. An octopus could be utilized to perform larger operations that might be more noticeable if done on one isolated machine. This is not a common piece of malware, but it is expected to be used more in the future. A *rabbit* can be used to refer to a worm that exists as one single copy at a time. It jumps from one computer to another as it executes its payload then leaves. Some researchers use “rabbit” to refer to a piece of malware that quickly replicates itself until the CPU load has increased to a state that can crash programs not made to run in environments with a heavy CPU load.

Logic Bombs

A logic bomb is a piece of malicious code that is intentionally hidden within a legitimate program. These occur on large projects where code reviews cannot be adequately done. For example, a program might be designed to delete itself after being used X number of times. Before deletion, it could slip a piece of code in to the system with malicious intent. Roger Duronio[32] was charged with

using a logic bomb against his former company UBS[33] in June 2006. He was convicted and sentenced to 8 years and 1 month in prison, and $3.1 million in restitution to UBS. Logic bombs are related to *Easter Eggs* – pieces of code slipped in with no malicious purpose. These can be pieces of code that give credit to people that helped develop the software, or just something special that the designers wanted to insert. For example, Microsoft's Excel 97 had a built-in flight simulator[31] that is not advertised and can only be accessed through a specific sequence of keystrokes.

In order for a piece of software to be considered a logic bomb, it has to be unwanted and unknown to the user. This means that a known functionality built in to a program that intentionally limits usability or removes itself after a trial period is over is not considered a logic bomb. The payload of a logic bomb is only executed when specific conditions are met at a specific time. They generally delete or corrupt data, print a specific message, or have other unwanted effects.[14] It is possible that a logic bomb will not be discovered or ever used if certain conditions do not occur.

Trojan Horses

A piece of software that convinces a user to run it due to some appealing characteristics, but has malicious consequences, is considered a Trojan. There are two distinct kinds of Trojans with a few subcategories. The first is a complete 100% Trojan. It might be named differently, or packaged and advertised differently, but there is nothing in it other than the malicious code. The second kind of Trojan is a modification of a legitimate piece of software. It still maintains some or all of the functionality as the advertised piece of software, but in addition it executes the piece of malicious code. The goal of a Trojan is to install unwanted software on a computer so that it can be remotely accessed and controlled without the user's consent. Trojans have become more lucrative because now they are often designed specifically to steal banking information. The use of Trojans has increased significantly. Figure 2 shows the increase in Trojans seen at PandaLabs[34]. In 2007, 70% of malware seen at the laboratory were Trojans.

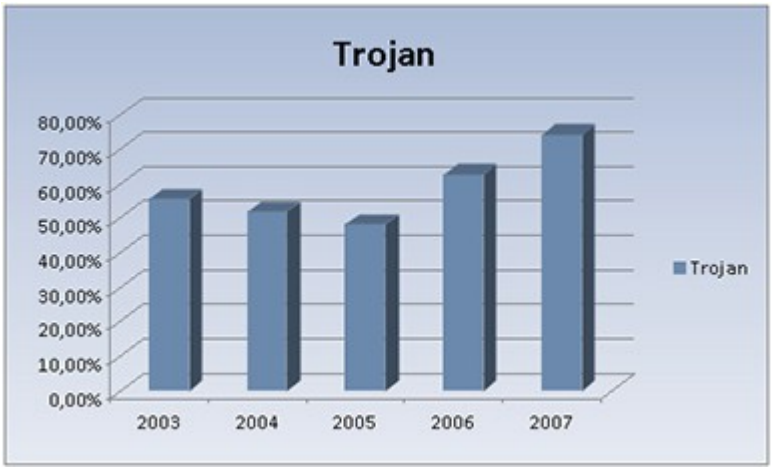


Figure 2: The percentage of Trojans seen at PandaLabs.[34]

A subcategory of a Trojan is a *rootkit*. A rootkit is a program that mimics the functionality of a legitimate program with some other malicious intention. For example, by modifying the “ps” command on a Linux system (a command that shows running processes) to hide specific processes while displaying the remaining processes, a rootkit could hide evidence of itself and other pieces of malware.

In addition to the rootkits, *password-stealing Trojans* are another subcategory of Trojans. They

reside on the infected machine waiting for passwords to be entered. When they are, the password is sent to the attacker so that he/she can have "legitimate" access to what the user is logging into. These are often accompanied by keyloggers so that each keystroke can be recorded. The Lydra.AO Trojan[51] was detected on October 5, 2008. It contains a keylogger to steal people's account and password information. In addition to having a keylogger, it gets information from the computer such as the computer name, IP information, and operating system version. This information is then sent back to the author.

Another type of Trojan is a *backdoor*. A backdoor is an application that opens a network port so that a remote connection can be made to the system. The hacker that created the Trojan can then access the infected machine at any time. This can be combined with rootkits to hide the backdoor from the user so that no suspicions are raised. An example of a backdoor Trojan is Redvoz.A.[50] is an example. It was first detected on October 1, 2008. It installs itself on a compromised machine then connects to a server. An attacker can then connect to the server and remotely control the machine. A backdoor can also be built into an application intentionally. Some programs create backdoors for troubleshooting purposes. Many of these are not meant to go live to the public, and some are not meant to be discovered, but they become problematic when they are.

Exploits

Exploit code is created to target a specific vulnerability. It can be executed locally or remotely on the target depending on the targeted vulnerability. Exploit code is not always malicious. A "white hat" hacker can create exploit code to do legitimate penetration testing. In this respect, the severity of the exploit can vary depending on the intentions of the attacker. Exploit code is used to break into specific programs or systems to gain information or elevate privileges. It is also sometimes traded amongst both penetration testers and hackers.

Many exploits have been produced for Microsoft applications[35][36][37][38]. Four different exploits were discovered at PandaLabs on September 10, 2008. The MS08-055 exploit[35] takes advantage of uniquely crafted URLs that reference OneNote files. If the exploit is successful, it gives the attacker access to the computer with the access privileges of the currently logged-on user. The MS08-054 exploit[36] gives control of the user's computer to an attacker through a vulnerability found in Windows Media Player Version 11. A link to a maliciously created audio file executes the exploit. The MS08-053 exploit[37] gives an attacker remote control of the computer with the currently logged-on user's rights by exploiting a vulnerability in Windows Media Encoder 9. The MS08-052 exploit[38] uses a vulnerability in the Microsoft Windows graphics device interface (GDI) to give an attacker remote control of the computer with the privileges of the logged-on user.

Kits (Virus Generators)

A kit is a program used to create viruses automatically based on the decisions the user has made. They are usually menu driven so that very little background knowledge is needed. Instead, the user selects the specific kind of attack and potential targets; then, the virus is automatically created. Most often, these are faulty or unsuccessful viruses. However, occasionally a virus is generated that works and is quite destructive.

A kit that generates malware does not have to be used exclusively to create viruses. YTFakeCreator[18] is a hacking tool that was first detected on September 5, 2008. It is used to create fake YouTube pages that people can then use for other attacks. This can be coupled with any type of malware; YTFakeCreator only facilitates the delivery of the malware. When a website created by this

tool is accessed, it looks like a real YouTube website, but an error is displayed. The error can be customized using the tool. If the error message link is followed, then the malware specified by the creator of the site can be downloaded to the host system. If coupled with another tool that creates malware, a malicious user without any knowledge of malware creation or computer security can gain access to a delivery mechanism and cause the malware to be delivered.

Spammer Programs

The main purpose of spammer programs is to generate incoming traffic to a particular website. They are normally used for advertisement purposes; they target mobile devices through e-mail, newsgroups, or even text messages on cell phones with messages regarding the website they are advertising. Another purpose of a spammer program is a phishing attack. The attacker might send a message saying that you need to visit a particular login site, and a link is supplied. The link then goes to a duplicate fraudulent site that is made to look like the original. If any personal information is put in, it is then given to the attacker. This technique can be used for identity theft.

The Spammer.AJR[56] malware was detected on October 12, 2008. The malware sends a message to a user advertising rogue anti-malware software. A link to a website imitating YouTube[57] is given in the message. If the link is followed, adware is downloaded to the user's computer. This adware warns of malware on the computer and advertises different anti-malware solutions. The anti-malware solutions do not provide any useful service, and are not free of charge.

Keyloggers

A keylogger simply logs the keystrokes inputted into a compromised system. Everything typed by the user is stored so that the attacker can see everything that he/she types. This often includes user names and passwords, I.D. Information, social security numbers, or credit card information. This is another piece of malware used for identity theft. A keylogger could also be combined with a rootkit so that the compromised system does not show any signs of being compromised, and the attacker's method of transferring the keystrokes back can be hidden.

Keyloggers can be categorized differently depending on the attack type. For example, McAfee classifies Keylogger.c as a Trojan with a subtype of keylogger[39]. Keylogger.c captures a user's keystrokes then mails them back to a configured e-mail address using its own SMTP engine. Hacktool.Keylogger was identified by Symantec on February 13, 2007[40]. It is a keylogger with a defined method of transmission of "...can be installed as part of a threat, such as a Trojan horse." A keylogger can be bundled with various kinds of malware, or use different kinds of malware, to be installed on a machine. Once installed, they all act the same by logging a user's input in a manner that the attacker can later use it.

Adware and Spyware

Adware and spyware are software installed on a computer system with the intention of monitoring the user's habits and reporting them back to the company that installed them. They usually do not have any direct malicious intent. However, they can lower security settings on web browsers so that the system becomes more vulnerable to other pieces of malware. Often, these programs are not effectively built and use a lot of resources, negatively impacting system performance. Adware and Spyware make up an unusual group; it is debatable whether they are even considered malware, since there is no malicious intent. There is conflict about their classification and removal. Many companies

use adware or spyware for directed advertising, and would likely see reduced revenues if they were no longer legal.

The VirusResponseLab2009[41] adware is an example of adware that does not directly harm the computer, but does harm the user. The adware displays screens to the user warning them of threats that do not exist on their computer. To resolve these threats, the user is instructed to purchase and download the antivirus software from their website. After it has been purchased, it displays a user interface with fake information and pops up windows displaying fake attacks at random times. The adware does not do anything to the computer aside from installing itself, but it does trick the consumer in to paying for software that does nothing useful. This type of malware has been called "scareware."[42] A scareware vendor based out of Texas was sued in October 2008 under Washington's Computer Spyware Act. The act "...punishes individuals who prey on user concerns regarding spyware or other threats. Specifically, the law makes it illegal to misrepresent the extent to which software is required for computer security or privacy, and it provides actual damages or statutory damages of $100,000 per violation, whichever is greater."[43] Bruce Schneier, "an internationally renowned security technologist", has promoted the lawsuit as "a good thing."[44]

Current Malware Trends and Statistics

Vulnerabilities

Malware has traditionally targeted vulnerabilities in a specific host operating system. At one point, a large portion of the population could be found using the same operating system, so it offered a wide attack vector. In order to deliver their malware to the broadest range of people, malware creators targeted the most common base operating systems (generally Microsoft Windows). This has begun to change in the last few years. As operating system usage begins to diversify, it is becoming less effective to write malware for one specific operating system. Instead, malware creators are targeting applications. The year 2007[22] saw a significant increase in vulnerabilities of browsers, office applications, audio and video multimedia players, pdf readers, antivirus products, and compression applications.

Many of the new application-specific vulnerabilities require some sort of interaction from the user. This can be anything from opening an e-mail or attachment to visiting a particular website. The use of social engineering attacks has increased to get the user to take the actions required by the malware developers to infect the host system.

Microsoft Office applications were frequently targeted in 2007[22]. Severe vulnerabilities were found for many of the applications, including Outlook, Word, PowerPoint, Excel, Visio, FrontPage and Access. It is a widely-known fact that Microsoft updates their software every second Tuesday of the month. Exceptions might occur when more critical updates are pushed earlier than expected. Malware developers take advantage of this by releasing their malware on the Wednesday after release so they will have maximum impact. This predictable schedule makes Microsoft applications a more tempting target when considering how many people use the programs.

The corporate climate has changed considerably with the increasing use of the Internet. Now, nearly every employee can have access to the Internet to browse from time to time. It was once possible to offer adequate protection by securing DNS, mail, and web services and putting up edge network security solutions like firewalls. However, with the extensive amount of traffic a corporation can generate, this is no longer sufficient. Recent trends are showing that browsers are a focal point of hackers[22]. While blocking content from untrusted sites prevents a large amount of malware, the number of infected trusted sites has been increasing. In 2007, it was found that embassies, banks,

online stores, and ISPs had trusted sites that had been infected to deliver malware to the people visiting them. As with browsers, multimedia applications are, by default, installed on many different operating systems (Windows, OS X, Linux, Unix, etc.) and offer a large client base for attack. These vulnerabilities are often executed by enticing the user to open a multimedia file through email or other advertisements. These files will then install the particular piece of malware on the system, leaving it vulnerable to the attacker. Often, they do play some sort of media so that the person being attacked does not realize that something unwanted is going on on their machine.

Current Trends

The most recent documented trends from April to June 2008 [21] show a significant increase in the use of Trojans and the infection of trusted sites to deliver malware. Instead of using attention-grabbing techniques that are very easy to detect, many pieces of malware aim to work silently without the notice of the user. Trojans work perfectly to accomplish this task. A large increase in the use of Trojans has been to steal banking information. There has been a 400% increase in the use of Trojans to steal banking information in the year 2007. The most active families have been Brazilian banker Trojans and Russian banker Trojans (1.0 and 2.0). The Brazilian bank Trojans (Banbra, Bancos) are designed to steal passwords primarily from Brazilian and Portuguese banks (and less frequently Spanish banks) then transmit the information through FTP or email. The Banbra family of Trojans is created with Delphi, while Bancos is created with Visual Basic. Instead of the normal kit-developed Trojans, these were each created individually. The Russian banker Trojans, on the other hand, are largely created using Trojan kits. Because the core remains the same, many functions have not changed, making them easier to detect. The Russian banker Trojans 2.0 family (Sinowal, Torpig, Bankolimb) is more dangerous because they are updated more often. This family has one distinct characteristic in common. The banking information targeted is given to the malware through a configuration file. This particular functionality is easier to locate, but it is also much easier to modify the target since the malware itself does not need to be modified, only the configuration file. They also employ polymorphic and stealth techniques to try and make detection more difficult.

The use of botnets and zombie networks to spread spam has also been increasing. During the first months of 2008, it was estimated that between 60 and 94 percent of all emails sent were spam.[21] The United States is currently the largest producer of spam emails, responsible for 34.21% of all spam. Another avenue of attack has been recently added to the spammers; social networks are now being targeted. At the beginning of the year, spam images were beginning to appear on the Flickr network. The Twitter network has also seen an increase in spam. Various advertisers have created identities and asked to be added to a profile. The advertised product can usually be found in the name, and the information in the profile contains nothing but advertisements.

With the increase of spam, an increase in phishing attacks has also been noted. Increasingly advanced techniques improve both content and detection evasion capabilities and allow spam and phishing emails to get through. Phishing kits are also improving. Web page templates of any major bank or organization can be found in a phishing kit, so that someone wanting to perform an attack does not need knowledge of how to create one. With the abundance of freely hosted web servers and email accounts, it is now possible to perform lucrative phishing attacks with no upfront knowledge or investment. Easy creation and high profit margin of these attacks means they will likely increase in the future.

The number of computers that have been made part of zombie networks or botnets has also been increasing. The Shadowserver Foundation[19] attempts to track the use of zombie networks and botnets. An infected system communicates with a Command and Control (C&C) server to receive its

orders and attack instructions. Tracking the C&C servers and what machines communicate with them gives an estimate of currently active zombie networks. If a C&C goes down, the associated bots with that server are removed from the count. In addition, entropy values are put in place to keep from inflating the numbers by counting inactive machines. The Shadowserver Foundation has seen an increase from 100,000 to 450,000[20] in machines that were part of a botnet from June 2008 to August 2008.

Initial reports from the third quarter of 2008 (July – September) show an increase in Adware. As Figure 3 demonstrates, Trojans still represent the largest segment of newly detected malware. However, the increasing amount of fake anti-virus programs has caused the percentage of Adware to increase from 22.03% in the second quarter to 37.49% in the third quarter.

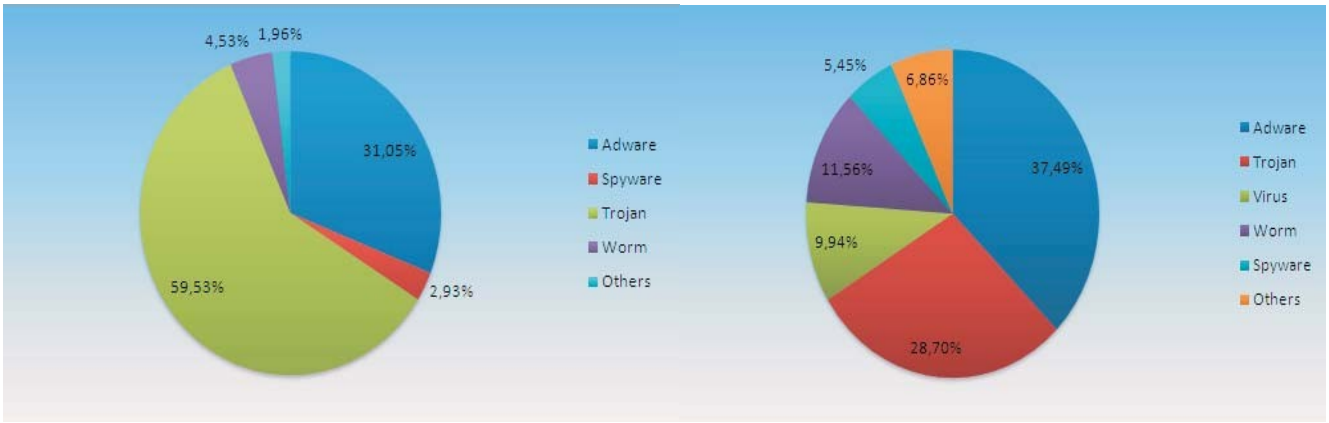


Figure 3: Left: Distribution of new malware detected by PandaLabs in the third quarter of 2008 based on malware type. Right: Distribution of malware detections made by PandaLabs sensors in the third quarter of 2008.[53]

Virtualization

Types of Virtualization

Virtualization is used to abstractly represent computer resources so that a type of virtualized system can take advantage of them. There are over a dozen types of virtualization, ranging from full platform virtualization to something as commonly used as the Redundant Array of Independent Disks (RAID). For the purposes of this research, the types of virtualization examined will be those currently used for malware research, or types that may help to facilitate malware research in the future.

Platform Virtualization

Platform virtualization is performed on a given hardware platform by the host software on the system. A virtual computer environment (virtual machine) is created for the guest software, possibly an application or an entire operating system, to be run in. There are generally no requirements that the software being run in the virtual machine be compatible with the software outside of the virtual machine. The guest application will normally need access to the hardware of the system. This is accomplished by allowing the guest system to interface with the virtual machines hardware as if it was directly accessing the actual machine's hardware. In full virtualization, the virtual machine supports the hardware so that the guest application can operate normally as if it were installed natively; no modifications are needed to get it working properly. Hardware-assisted virtualization allows for the application to be run in the same manner, but the hardware has specific built-in features that facilitate the use of virtualization. It can be used to ensure that the guest applications are run in complete isolation. Paravirtualization does not allow the guest application to be run unmodified. Instead, a layer is added between the guest application and the host system (such as the Hypervisor). The guest application then has to be modified to use the API established by the new layer. System calls are made to the virtualization layer instead of the host system, and these can differ depending on the API used.

Application Virtualization

Normally, platform virtualization is used for entire operating systems. That is why much effort is put in to adding support for various hardware architectures and system calls of different operating systems. Application virtualization is not capable of supporting an operating system, but is meant for single applications. In effect, application virtualization tricks the application into thinking it has been installed directly on the system, when actually it has been encapsulated in some manner, preventing it from being installed directly on the hardware. This is accomplished by adding a virtualization layer between the application to be installed and the underlying hardware, similar to paravirtualization, but on a different scale. When the application makes a call to anywhere outside of what has already been loaded for the application, it is intercepted by the virtualization layer and the call is redirected to a virtualized location. Any call to the disk, registry, or any other piece of hardware is instead sent to an area maintained by the virtualization layer so that the user can control what is accessed and how the accessed data is maintained. Instead of virtualizing the hardware so that an operating system can access it, application virtualization virtualizes aspects of the operating system so that can applications can access the operating system in a manner that is controlled by the user.

Benefits of Virtualization

At a corporate level, malware is an expensive problem. It can lead to time loss due to computer inefficiencies, data loss, or even hardware corruption that necessitates the need to buy a new system altogether, often losing many irreplaceable components (dated software, for example). Malware detection that takes advantage of virtualization could benefit a home user as well. The user can test whether or not a piece of software is malware by installing it in a manner that isolates it so that it can be tested before allowing it to interact with the actual system. This could help to determine the negative consequences of software that has not been thoroughly tested, whether the consequences are intentional or not.

Cost Benefits

The consumer cost benefits of using virtualization are clear. The cost of time to repair a system after the malware has delivered its payload, the cost of replacing any hardware or software, and any personal costs incurred are avoided if the malware can be detected before it is permitted to harm the system. There is additional cost benefit for using application testing. A system that verifies software through virtualization can be used for more than detecting malware. Program interoperability, system utilizations, memory costs, and many other facets of the software can be tested in addition to the presence of malware. This means that various programs can be thoroughly tested by a third party application instead of needing to be scrutinized by the user. This is particularly beneficial for freeware, where there are many different open source solutions to the same problem, but determining which one is best for the user can be difficult. This is all possible because the home user does not have to meet any special or additional hardware requirements in order to test the software. Virtualization allows any other program to run on the same machine without buying a new system to test with. This can be extended to the corporate level. Many corporations make use of testbeds for large applications. While this would likely still be necessary, the time invested on the testbed and hardware usage could be minimized by testing single computer applications in a virtualized environment. A single application is not the only thing that can be tested or implemented virtually. More than one system could be virtualized, or even an entire network. Then, the interaction of various computers over a network can also be monitored to see impacts on issues such as bandwidth. Unnecessary hardware costs can also be removed by creating an environment where hardware will not be damaged during application testing.

There is also a measurable benefit to corporations utilizing virtualization due to the reduction in power requirements.[17] Condensing several physical servers down to one physical machine with several virtual machines can save thousands of dollars over the course of a year, depending on how many machines can be reduced. The reduction in systems can be continued further by converting large servers to smaller virtualized machines and using more efficient data storage techniques such as storage arrays. It is difficult to measure additional savings such as the reduction in cooling requirements because they would differ determining on the installation area. However, less cooling would be needed due to fewer machines in use, lowering the power needs of the cooling system ,and possibly allowing for a smaller cooling system.

Program Monitoring

With many different software development tools, it is possible to monitor the execution of code in a line-by-line manner. This allows the programmer to see what exactly is happening at every line of code. The ability to pause the execution sequence to examine various parameters, or modify various

parameters, and then continue with execution, is used to determine exactly where particular effects are occuring. The same capabilities can be used in a virtual environment. Instead of looking at each line of code, an application can be launched in its normal working environment and then monitored to see exactly what it is doing. This includes the ability to pause the application and look at various processes, registers, and opened files, or gather state information. This information can then be used to determine if the program is performing in a manner that is agreeable to the user or not. The working environment can also be modified to show what kinds of future effects the program might have. For instance, the current date could be modified to be many years in advance to see if anything unusual changes with the application (such as a virus with a set time to deliver its payload, or a date malfunction). Ie essence, the user would be able to see everything that the program would do to his or her computer without the program actually having access to their system.

Performance Isolation

Depending on the technique, using virtualization to aid in malware detection is only a reasonable strategy if the use of virtualization allows the user to continue using the system for his or her normal activities. To this end, it is important to determine the impact that virtualization would have if it were hosted on the user's own computer instead of on a remote system[8].

In order to examine the performance isolation capabilities of virtualization, many different types of virtualization (full virtualization, paravirtualization, and operating system level virtualization) were tested to see how applying stress in the virtual environment affected the performance outside of the virtual machine.[8] In total, 6 tests were run on each virtualization environment:

1. Memory Intensive – loops constantly allocating and touching memory without freeing it
2. Fork Bomb – loops creating new child processes
3. CPU Intensive – tight loop containing integer arithmetic operations
4. Disk Intensive – running 10 threads of Iozone, each running an alternating read and write pattern
5. Network Intensive (Transmit) – 4 threads each constantly sending 60K sized packets over UDP
6. Network Intensive (Receive) – 4 threads each constantly reading 60K sized packets over UDP

These tests were all performed on the various virtualization environments: Vmware Workstation (full virtualization), Xen (paravirtualization), Solaris Containers and OpenVZ (operating system level virtualization). To test the performance isolation properties, SPECweb (SNMP based) was used to report metrics of interest. Each virtual machine then had an Apache web server configured to report on the requested metrics. When performing the stress tests, a Virtual Machine (VM) was put under one of the above tests (the “misbehaving” machine) while the others remained under normal conditions. The results of the misbehaving machine and all other VMs were then compared to the baseline of all of the VMs working without one of them misbehaving.

The results showed that even when one of the virtual machines was misbehaving, the other VMs did not suffer a significant performance loss. The virtualization technologies add a layer between the VMs and the operating systems that is capable of isolating the virtual machines. This also controls the amount of resources used so that one VM can not consume all of the resources and cause the others to become unresponsive or unstable. This throttling results in the other VMs operating normally, but it causes the VM that is working rigorously to suffer severe performance losses.

Software Efficiency

Different types of virtualization require varying degrees of resource management and

utilization, processor and memory utilization, and resource usage. To ensure a high degree of isolation, entire operating systems can be duplicated. When examining potentially harmful code for malware, software isolation is necessary. However, for it to be feasibly used at a consumer level it needs to be done without using so many resources that the average computer is not compatible with the software.

The Feather-weight Virtual Machine (FVM)[3] uses namespace virtualization to significantly cut down on the resources used in the virtualization. The benefit of this approach over others is that it is also compatible with Windows, and maintains software isolation. Two techniques are used to reduce the resource cost of virtualization. The first is the use of namespaces. Namespaces are already highly developed in the Windows environment. When a virtualized resource tries to perform an action, such as access a file in a particular directory, the namespace is prepended to the directory so the unique virtual file, which is a copy of the original, is accessed. This same method is used for processes, memory accesses, hardware interfacing, etc. Everything in a particular VM has its own namespace. This also allows more than one virtualized resource to operate; each one can have a unique namespace and thusly unique resource abstraction. The Windows environment ensures that differing namespaces will maintain isolation so that one VM will not be able to interact with other VMs.

The second technique is the copy on write technique. If the VM is trying to access a resource with a read-only command, it is not of high concern if that particular resource is shared or copied and unique to that VMs namespace. In the case that it is going to be modified, it is copied in to that VM's namespace. This saves time copying unnecessary items (those that are only going to be read from). The copy on write technique and namespace virtualization are accomplished by intercepting system calls at the kernel level. This is where the namespace rewriting is done, as well as where the copy on write decision is made. The FVM and its virtualization layer operate at the kernel level instead of the user level, making them harder to exploit and easier to maintain separation.

The FVM achieves very efficient virtualization. Tests of the prototype have shown latencies of creating and starting VMs of less than one second. Run-time virtualization was demonstrated to be less than 20% of the total execution time of the tested applications. The FVM was created with the purpose of being resource- and time-efficient, particularly for the use of intrusion and malware detection systems that need to harness the capabilities of virtualization to do sandbox-like testing.

Honeypots and Sandboxes

Overview

Honeypots allow for their administrators to monitor current attack trends. In addition to being a useful tool to gain relevant information regarding malware, honeypots provide a system architecture that is meant to be exploited but easily recovered from. One of the values of a honeypot is that it can be exploited to find new types of malware, but it can be taken offline at any time (it is not a production unit) to be examined to determine the extent of the damage. However, a honeypot also offers an environment that is intentionally susceptible to malware, but made specifically to not redistribute malware. This offers a relevant environment to study to aid in creating a system that can efficiently detect malware while remaining resistant to being compromised. In the event the system is compromised, it is set up to be easily recovered while maintaining enough integrity that malware is not redistributed and no useful information is returned to the attacker.

A honeypot is a system that is never intended to be legitimately accessed[23]. Anything interacting with a honeypot can be considered unauthorized. This helps in forensic examinations of honeypots because there are no false positives. A honeypot can be used in a production setting, or a research setting, or a combination of the two. In a production setting, the honeypot mirrors the actual system that the company is running. Honeypots are highly monitored, so that they can be examined at any time to determine who is trying to access them, what they are doing, and how they are doing it. The security levels of a production honeypot mimic those of the actual network so that information gained regarding attacks on the honeypot can be applied to the actual network. When an exploit is used on the honeypot, it is reported to the administrators so that the vulnerability can be addressed and fixed on the legitimate machines. They do not work very well as decoys because most attacks are automated and will be launched on any machines that are found. However, they are useful for finding vulnerabilities on a system that can be taken down and examined at any time, which is something that is much harder to do on a working system. The goal is to determine how a honeypot becomes compromised so that the vulnerability discovered can be fixed on the legitimate system, preventing it from being compromised by that vulnerability. A research honeypot is deployed to gain more information about the nature of attackers and malware. Instead of looking for specific vulnerabilities that need to be addressed in a legitimate system, a research honeypot aims to discover current trends and attack vectors so that defense mechanisms can remain updated. A research honeypot is generally monitored more heavily than a production honeypot because the administrators are looking for more information than what the live machines are vulnerable to. A research honeypot can also determine where the attacks are coming from, what kinds of tools are being used, or the current trends of malware. The difference between a production honeypot and a research honeypot is based on how they are used as opposed to how they are setup or where they are deployed. It is possible for a company to utilize a honeypot to fulfill both roles.

A second distinction between honeypots[23] is the level of interaction allowed between the attacker and the honeypot. Low-interaction and medium-interaction honeypots do not have operating systems installed. Instead, available services are simulated. The degree to which they are simulated determines if they are low or medium-interaction honeypots. A medium-interaction honeypot will simulate the services so that the attack can progress through its stages; this allows the honeypot monitors to gain more information from the attack. Low- and medium-interaction honeypots take less time to monitor and administer because there is not an operating system that can be taken over and used against the honeypot administrator. It is critical to ensure that a honeypot is not taken over and used in a malicious manner. If this were to happen, it could harm the network and the data collected could be

contaminated and unusable. A high-interaction honeypot has an actual operating system installed and provides enough services that it can be fully attacked. This allows administrators the ability to collect detailed information about the attacks, but it is more time-consuming to monitor. The services provided can fully interact with an attacker up to and, including, being compromised. This leads to high-interaction honeypots being taken offline for forensic examination more often, or at the least rebuilt to get rid of the malware that exploited the system.

A *sandbox* uses some of the same techniques that a honeypot uses. A sandbox is used to allow questionable or unknown software to execute in a safe environment.[54] A honeypot is made to attract malware by appearing to be vulnerable to the attackers. This creates an environment where a certain degree of malware is expected to be executed. A sandbox also allows the execution of malware, but the environment is more restrictive because the administrator of the sandbox determines what will be executed, as opposed to a honeypot where anything could be executed. A sandbox can allow a program to execute in a line by line basis, monitoring exactly what happens at each moment of execution.

Honeynet

A honeynet is an unused network that consists of various honeypots. All traffic to and from a honeynet is suspect. The honeypots can be different system configurations running various operating systems. The honeypots in a honeynet are not emulated and are representative of "off the shelf" production models. They are not secured, but their security is not diminished. A honeynet sits behind a firewall that archives all network transmissions so that attacks can be analyzed at a network level. Instead of the firewall protecting the honeynet from the Internet, the firewall protects machines on the Internet from potentially compromised machines in the honeynet.

The Honeynet Project[24] released a revised set of suggested standards in October, 2004. Three important definitions and requirements are offered: Data Control, Data Capture, and Data Collection. Data control ensures that malicious interactions can occur with the honeynet, but the results of any compromises will not harm non-honeynet systems. Data control takes priority over data capture, and should not be detectable by the attacker. Data capture captures all inbound and outbound activities to the honeynet, this is also done in a manner that is undetectable by an attacker. Data collection is done in a distributed environment when several honeypots are providing data. Data captured is securely forwarded to a centralized location for analysis and archiving. This is important to insure the integrity of the data collected and to prevent attackers from viewing the collected data and determining that the system is a honeypot and not an operational system.

A honeynet is effectively nothing more than a networked set of honeypots. However, it has to be carefully controlled to collect as much useful information as possible while hiding the fact that it is not a production system from an attacker. The defining aspect of a honeynet is its architecture[25]. To ensure data collection, data capture, and data control, an interface is added between the honeynet and the Internet. This is generally a layer 2 device with no IP information so that it will be hidden from an attacker. The Honeynet Project refers to this interface as a honeywall. Figure 4, below, shows the suggested architecture of a honeynet. The honeywall can have an interface with an IP address for remote administration as long as it is not part of the honeynet. The interface between the router and the honeywall is bridged, as well as the interface between eth0 and eth1 so that no additional IP information is provided. The honeywall is responsible for allowing malicious attacks against any machine in the honeynet and stopping any machine in the honeynet from attacking anything outside of the network.

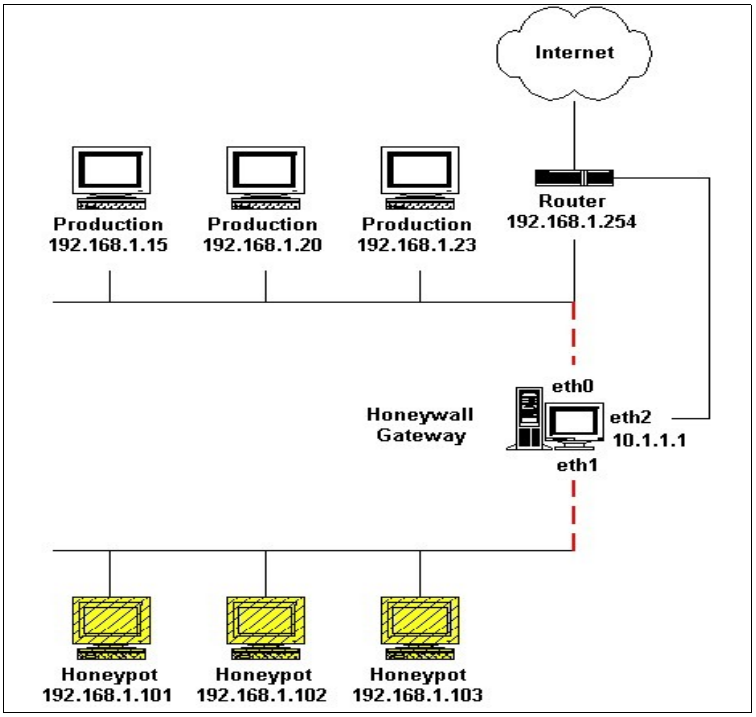


Figure 4: The Honeynet Project's proposed architecture for a honeynet.[25]

To monitor the current activities of the blackhat community, the Irish Honeynet Project[26] followed the steps initially outlined by the Honeynet Project. Data control is accomplished with a firewall as the only point of entry and exit to the honeynet, called a honeynet sensor. In order to collect useful information regarding what the malware is doing, up to five outbound connections are allowed through the firewall. This allows the collection of information regarding what tools the malware is downloading to the compromised machine and who it is trying to communicate with. If the five outbound connection threshold is reached, any additional attempts at an outbound connection will fail. This failure will trigger an alarm to the administrators, notifying them that a honeypot has been blocked. The honeynet sensor is also responsible for data collection. To provide more useful data collection, the honeynet sensor collects data at multiple layers to “paint the big picture”. The Irish Honeynet Project ran for a year, gaining statistical data on attacks. In February 2004, 1800 attacks were recorded from 1121 unique IP addresses. The IP addresses came from 65 different countries. A total of 45 different ports were targeted by the attacks. The wide range of sources and destinations showed the variety in attacks that a system needs defense against.

Defensive Techniques: Malware Detection

Signature-based Techniques

Signature-based malware detection techniques have been used extensively over the last few decades, since malware has started to become more prevalent. They work on the premise that one can find a unique sequence of characters (signature) in a piece of malware that will only be found in that specific malware. One can then scan through everything on a machine and coming into the machine for that signature. If it is found, it is treated as malware; if not, it is considered to be clean. These techniques were initially effective because there was a limited amount of malware. Also, in comparison to present machines, there was a smaller amount of legitimate software. With the explosive growth in both malware and legitimate software, however, it is harder to find unique signatures that are efficient to search for.

When using a signature-based technique of malware detection, there are several variables that will determine if the technique will be successful enough. The first variable is if the list of signatures is comprehensive enough to detect known malware. After developing all of the signatures, they need to be stored in a manner that is quickly accessible, memory efficient, and computationally easy to locate. A common solution to this is using a hash. The key is something that is efficient to create but yields an accurate representation of the signature. The value is then usually the exact sequence of characters that makes the signature. The length of the signature needs to be considered as well. If it is too short, it will match too many strings and not provide useful information. If it is too long, it will be computationally inefficient and a full system scan will take too long. This is particularly true of systems that have low memory, processing power, or battery life. In an attempt to make a useful signature-based malware detection system for mobile devices, Smobile Systems[9] studied a method of using filter-hash strings. A signature-cut method is used to determine the smallest effective substring of the signature to use for the filter-hash. This is the set of bytes in the signature that are the least likely to appear in a legitimate piece of software. The aim is to eliminate legitimate software as quickly and easily as possible to keep from wasting computational time. This filter-hash created from the cut signature yields a key that is more general and matches to more signatures. However, it is computationally efficient and cuts out enough legitimate software that it saves time. If there is no hit from the filter-key, then computation on that particular signature ends and the next sequence can be examined. If there is a match, the process continues to a DJB hash. A byte by byte comparison is done before moving to the DJB hash. This is a more complex process, but it results in a stronger hash value. The chance that the subsequences have the same DJB hash value is very low. The possible duplicates are not in the range of legitimate character sequences, so it is considered to be more accurate. In order to positively identify the signature as malware, the starting index of the cut signature is moved back to the beginning of the initial sequence. The full hash of the signature is then taken with the DJB method and the resulting signature is compared to the computed DJB value. If they match, it is considered malware.

Another method of the signature-based malware detection systems is the scanning method. Commercial antivirus software usually offers various scanning modes. The modes can range from a full system scan, examining all files and data in memory, to only scanning files that have changed since the last scan date. A changed file refers to a new file, a file that has been overwritten, or a file that has been modified during the particular time frame. It is also possible to scan only incoming information, such as email, instant messages, or Bluetooth transmissions. The scanning method and detection algorithm needs to be able to handle different file sizes, different signature sizes, and various key and

value sizes while still effectively comparing the varying sized signatures and scanned strings to determine if they are malicious.

Signature-based malware detection has been effective in the past, but as more pieces of malware are being released it is becoming harder to keep up with the attackers. Methods are being developed to evade signature-based techniques. For example, polymorphic viruses can modify themselves as they are redistributed so that the signatures can change for the same virus. In addition to modifying themselves, the sheer number of viruses is becoming problematic. The assumption that signature-based methods can detect malware is based on the assumption that a unique signature can be found for every piece of malware. As more legitimate and illegitimate software is created, the unique strings are more difficult to find, and some begin to overlap with each other. This creates a problem of false positives. For signature-based methods to overcome this, they must first determine if a piece of software is believed to be malware through signature scanning. They then use another method that ensures it is not in fact a piece of legitimate software that shares the same signature as a piece of malware.

The problem of finding unique signatures is not the only or arguably largest problem of signature-based detection methodologies. The nature of a signature means that the virus has been studied enough that a unique or quasi-unique signature has been extrapolated. In order for this to be true, the assumption has to be made that the malware has first been found. That makes these systems ineffective at detecting new malware that strongly differs from anything currently released. It is possible to detect a zero day attack if it happens to have an overlapping signature with an older attack, or is close enough to it that the malware detection system will still flag it. This weakness means that signature-based systems are usually coupled with other detection techniques for critical systems.

While signature-based methods are becoming more criticized, they are still used in many of the most common antivirus products. Symantec, Trend Micro, and McAfee currently use signature based scanning methods to look for and get rid of malware. However, they have noted that this method (while not as on the decline as people believe) needs to be supported through other methods. Symantec's senior director for product management was quoted in 2007 saying “Everyone agrees signature-based defense is not enough.” Symantec plans to use more whitelisting technologies to try and help the signature-based suite. Trend Micro is also looking forward to new technologies. While still believing that there is utility in signature based methodologies, they intend to invest in more heuristic-based approaches. McAfee continues to use signature-based approaches as well, combining them with host-based intrusion prevention mechanisms. While McAfee is working to add more prevention mechanisms on top of their current suite, they still feel that signatures are needed: “I can understand why some would think signatures are dying, but it does go back to someone not really understanding what a signature is. Some cleaning and repair can't be done without them,” said David Marcus, McAfee's security research manager.[15]

Heuristic Techniques

A heuristic technique makes use of all available information to determine if the suspect data is malware or not. The premise is that there is a fundamental difference between legitimate software and malware. If something is examined long enough, a normal working pattern can be developed. When things change enough from this studied and developed normal behavior model, then a heuristic malware detector will determine that it is malware. This technique is beneficial because it does not rely on having already seen the malware that is attacking the system. It relies on being able to identify a normal working pattern, and identifying when the system has strayed far enough from the normal working conditions to know that something is wrong. Many pieces of malware are written to mutate their programming so that they operate the same way but are identified differently. These mutations

will often throw off signature-based methods, but a heuristic method can still detect this malware.

A heuristic approach can work on many different levels. The system can be monitored on a file by file level; it looks for variables such as creation and modification times, file location and use, file density, or how the file is interacting with the operating system. It can also be used system-wide; looking at all running processes, currently opened and closed ports, TCP and UDP connections, or CPU and memory loads. Additionally, a heuristic approach can look at the network level to see if anything is out of the ordinary. In this case, it would look at things such as the current amount of bandwidth used, active machines, server loads, and activity to time of day comparisons. A heuristic technique can effectively be used at any level where enough data can be collected so that a state of normalcy can be defined. When enough information is collected that the malware detection system knows how the system should be working at a particular time, it can make an educated decision as to whether it is currently operating within the scope of what is expected. Due to the need to collect a significant amount of data, a heuristic system often needs time to collect enough information before it can be considered reliable. In this "down time", another malware detection system is used to prevent attackers from abusing the system and to try and weed out any false positives that might be triggered by the heuristic system during its learning process.

When a heuristic technique is comparing the current state of the file(s), system, or network, a probability model is used to identify if it is malware or not.[11] If the data in question has something different from what it believes to be normal, its probability of being malware is increased. As these differences add up, the probability increases until it reaches a certain threshold. The threshold can be whatever is defaulted in the scanning system or something that the administrator has set. It can be increased to try and avoid false positives or decreased in a more sensitive situation. The decision process used to determine whether or not to increase the probability of being malware has to be carefully created in order to create a useful heuristic model. It is often assisted with decision trees or matrices, and a database of abnormal behavior can be created to gain confidence in the probability.

False positives can be a problem in heuristic models. Due to the nature of being probability-based, they can never be completely certain that the content in question is malware; they can only be very confident. This is why heuristic models are often used in combination with other models. When combined with a signature-based method, it can be useful for determining if a suspect file is infected with something that has been seen before. This then makes it possible for the malware detection software to identify malware with certainty. In addition, the heuristic model can help to detect malware that the signature-based techniques cannot find due to a lack of relevant signatures.

General Decryption

As malware developed, a segment of encrypted viruses became particularly hard to detect.[11] The malware would come in two pieces; a small segment to decrypt/encrypt the entire package, and the payload. In addition to the encryption challenge, the viruses could also be polymorphic or metamorphic. Polymorphic malware would modify itself enough to evade signature detection before encrypting itself (with a new key) and propagating, while metamorphic malware would create an entire new generation of itself before propagating, often not encrypting itself.

It is easy to detect when something could be a virus of this variant because it is easy to spot the pattern of a decryption engine and an encrypted payload. The challenge is figuring out what it does so that proper defensive and cleaning steps can be taken. A general decryption scanner completes this task using three components. The first is an emulation environment where the virus is permitted to run while being monitored. The second is a virus signature scanner used to detect what kind of virus it is once it has been decrypted, and the third is an emulation control module. The virus is then run in the

emulation environment while the control module follows it step by step. After the virus has decrypted itself, but before it has done any damage, the signature scanner is used to try and detect the type of virus. With this information, the general decryption engine can disinfect the machine. Even if the malware were able to infect the emulation environment, it would not do any harm to the actual system because it is contained.

In the event the signature scanner cannot determine what the malware is, or if it is metamorphic, a heuristic approach is used to confirm that it is malware and to try and identify what kind. The heuristic technology observes and scores the behavior of the package. The score is then used to determine if the package is malware. Based on a configured threshold on a numerical rating based on the observation. It is possible that while the package contains the encrypted malware structure of a decryption header followed by an encrypted payload, it is legitimate software. Using the heuristic model can determine if it should be allowed to execute in a real environment or if its execution should be limited to an emulator so that it can be monitored to determine exactly what it does.

This method targets a specific family of viruses, but makes use of both signature and heuristic-based malware detection techniques. This increases the chance that it will be able to determine if a piece of software is malware or not. By isolating the package to an emulated environment, it keeps the malware from propagating and morphing itself. This technique is also useful because it does not require the malware detection software to be able to figure out the encrypted payload. Given the various kinds of encryption and strengths of encryption, it would not be feasible to try and decrypt every package needs to be examined without knowing any information about how it was encrypted or what the key is. Letting the package run so that it decrypts itself saves computer resources and time.

Integrity Verification

Before using or executing a file, it is possible to examine it for something wrong using the previous techniques. However, instead of doing this, another method is to ensure that what you are about to execute is correctly built as you had expected. This can be done by checking the integrity of the file before using it. The most common method of doing this is by using cryptographic hash functions. The two current popular hash functions are the SHA-1 and the MD5. The hash function takes the file as input and outputs a shorter representation of the file that is pseudo-unique to the file. Then, when you want to use the file, you use the same process to reproduce the hash of the file. If they are the same, this indicates that it is safe to use because it has not been modified. Had it been modified, a different hash value would have been produced. Each time a legitimate modification of the file is made a new hash value is created so that only unknown modifications will cause an alert.

The initial problem with these types of utilities are the collisions. This occurs when two varying inputs result in the same output. This is problematic because it means it is possible to change the file, yet produce the same hash value. In the current hashes, this is resolved by trying to ensure that the collisions are not meaningful. If two files produce the same hash output, one of them is a combination of unusable characters that would never be used as a legitimate file. Another way of dealing with collisions is to use more than one hash. While there might be collisions in both of the hashes, they should not share the same collisions. This means that even if one is modified in a manner that it produces the same hash value, the second hash used will differ and the file will not be accepted as unaltered unless both hash values match.

Another problem with integrity verification through hashing is that it is difficult to ensure that the hash value itself cannot be modified. If the file in question is modified, and the program that maliciously modified it also has the capability of rehashing the file, it could just modify the hash value to match the modifications it made so that no alarms will be raised. Often, to combat this problem, the

hash values themselves are kept in a secure read-only location. Then, in order to update them or modify them based on legitimate changes, a secure procedure has to be completed in order for the hash values to be changed. This can be accomplished by using a keyed hash function. That is a hash function that will yield the same results, but needs a key to produce the hash value. Then the key can be securely stored and the hash value cannot be easily derived.

Classically, integrity verification has been used to determine if a file has been modified. However, it has not been strictly used to determine if the file is infected, malicious, or even where the file has been changed. It could be used just to ensure that the file has not changed for any reason, including a legitimate one. Cryptographic hashing is used to find where the virus is in the file. Finding where the virus is in the file is referred to as virus localization. When trying to determine virus localization, it is important to know the infection techniques the malware can use on the file - rewriting, appending/prepending, and embedding. When a virus uses a rewriting technique, it just replaces portions of an executable file with itself. Appending and prepending malware inserts itself at the end or beginning of the file, respectively. If it is appended, a jump instruction can be put at the beginning so that the malware is executed first before giving control back to the original executable. An embedded virus inserts a pointer to itself in the executable that will load the malware when the file is executed. The malicious code is not actually in the executable, but in some other file. If the original file can be accessed, as well as the differing file, it is possible to break the two files into blocks of a single segment size (such as a per line basis for source code) and then hash each block. The differing blocks would show where the virus was located. This technique is too inefficient to be used regularly because of the expansion of the hash sizes (which are commonly too large to begin with). The resulting hashes would be unusably large for a file as small as a few kilobytes. However, some assumptions can be made. In order to find virus localization, you can assume that you only need to find approximately one-quarter of the file differing; any more than that, and the localization question is a moot point because too much of the file is infected for it to be legitimately used.

The blocks investigated can also be limited based on the infection techniques. Only the first portion of a file needs to be examined for a prepending virus. The file can then be inverted so that the last blocks appear first until the first block is seen last. The same technique used to find a prepended virus can be used to find an appended virus. In a rewriting virus, the fact that rewritten data is usually in adjacent blocks can be used to eliminate at least half of the blocks that need to be investigated. Once a difference is found, the adjacent blocks can be examined. The problem of investigating an embedded virus is more difficult because it can be located anywhere. However, instead of trying to develop a novel technique, or scan the entire file, the two previous localization techniques can be combined to try and find the embedded virus.

IMDS: Intelligent Malware Detection System

The IMDS[10] is an approach to malware detection aimed at Windows Portable Executables (PE). The PE file format is used to provide a common format that can be used across different Windows operating systems. Many new viruses are taking advantage of the cross compatibility of the PE API. This allows them to be executed on many different versions of Windows. The IMDS consists of three core modules: the PE Parser, an Objective-Oriented Association (OOA) rule generator, and a rule-based classifier.

Instead of using a disassembler, the PE Parser is used to construct the API execution sequence of the PE file. If the file is compressed or otherwise modified by a third party application, the modifications have to be undone for the PE Parser to work. Once the API execution sequence has been obtained, the API database is queried to generate a 32-bit global identification (ID) that represents the

static execution sequence of the corresponding API functions. These global IDs are used as signatures for the PE files. They are stored in a signature database with six fields: record ID, PE file name, file type (malicious or not), called API sequence name, called API ID, and total number of called API functions.

This data is then mined by the OOA algorithm to generate class association rules which are recorded in the rule database. After examining enough legitimate and illegitimate pieces of software, a pattern develops that differentiates the two by the sequence of API calls used. The data mining helps to find the patterns so that it is possible to distinguish the legitimacy of the software. The rules generated and the selected API calls are passed to a malware detection module that then produces the final report and decision as to whether or not it is malware. Many tests were run across 29,580 executables, of which 12,214 were benign and 17,366 were malicious. When comparing popular anti-virus software, the IMDS was able to detect the following polymorphic viruses: Beagle, Blaster, Lovedoor, Mydoom and variants of those viruses (up to 15 variations total). It was the only malware detection solution that successfully detected them all; Norton AntiVirus, MacAfee, Dr. Web, and Kaspersky failed to detect them all. When detecting unknown malware, the IMDS was able to classify 92%. The next two MacAfee and Kaspersky, only discovered 75.3% and 78% respectively. In addition to detecting more malware and its variants, the IMDS had fewer false positives and a more efficient data mining algorithm.

The SpyProxy Projet

SpyProxy[6] is an application developed to ensure that a web page is not doing anything malicious to the user trying to access it. When the user goes to a web page, the request is intercepted by SpyProxy. The page is then accessed in its virtualization environment and the effects the web page have on the environment are measured. If anything unusual happens (such as spawning a process or trying to download unwanted files), the web page is blocked. All of this is accomplished with only adding 600ms of latency to the initial page loading time.

When a HTTP request is made and is intercepted by SpyProxy, it first checks the cache to see if it already has data pertaining to the web page. For this purpose, the application Squid Web cache was used to do additional filtering. If the page is cached, then processing is done from the cache, otherwise it goes to the web to fetch the content. A static analysis is then performed to determine if a dynamic analysis using the VM is necessary. The dynamic analysis is a VM with a browser that is set to respond to the application. It can request the web page in the VM and then determine if it is safe or not. This is more time consuming, so the static analysis is used to try and filter out content when possible. The static analysis determines if the web page is passive or active. A passive web page contains no content that will be executed, where an active web page contains things that will be executed such as ActiveX or Flash. While a page can be determined to be passive, it is still possible for it to be malicious. For instance, there have been vulnerabilities found in PNGs and JPGs. For this reason a web page will only pass static analysis if it is passive and only contains HTML code. It is possible that an exploit can be found in the processing of the raw HTML code, so for this reason the static analyzer can be completely turned off if necessary. If the web page passes the static analyzer it is sent to the user to be viewed without going through the dynamic analysis. To ensure the safety of the content, the analysis is done starting at the root of the web request.

The threshold of the static analyzer is set to err on the safe side. If the web page is considered active or it contains any non-HTML content, it is sent to the VM for dynamic analysis to determine its safety. The web browser in the VM then fully renders the web page from the root directory. While it is

rendering, the VM is checking what has been modified to determine if it is safe or not. If nothing malicious has occurred, the web site is considered safe and sent to the user. The VM has all unnecessary services disabled and the web browser is the out of the box installation with no additional add-ons; this is to try and limit the possibility of something happening due to something unrelated to the web page access. If the web page is safe, it should not access the file system outside of safe zones (cache for example), spawn unexpected processes, or crash the browser or the operating system. These kinds of events are trigger events. If any of these things are triggered in the VM, the web page is considered malicious and is subsequently blocked.

The dynamic analysis can only detect malicious content if the malware is triggered in the sandbox-like environment. However, if the malicious content does not always execute itself, it is possible for it to not execute during the dynamic analysis. This would then allow the page to be accessed by the user, where it could possibly be triggered. The developers of SpyProxy have suggested fixing this error by duplicating any dynamic user interaction with the web page in the VM before the results are passed back to the user. If the malware is set to trigger on a particular user action (such as data being input in a field), then this setup would help to catch malicious content that is not initially triggered. Another limitation to the SpyProxy setup is that the web page must have completed rendering before it can be determined safe or not. If a web page does not terminate until some user interaction has occurred it could time out and be considered unsafe. To try and avoid delayed attacks, the VM moves the time forward to detect any changes in the content. If the web page times out without completing, it is considered unsafe. This could become problematic for interfaces that wait for user input, but it could be solved with the aforementioned strategy.

To test the effectiveness of the system, 100 known malicious web sites were gathered from Google, black lists, and SiteAdvisor. The sites used varied methods to deliver malicious content. Some tried to force the malware on the user by asking them to install it. In this case, SpyProxy automatically accepts the request and installs the software. If any trigger events go off after the software is installed, it is considered unsafe. SpyProxy was able to detect 100% of the unsafe sites and accurately blocked them all. SiteAdvisor mis-categorized 20% of the malicious sites as safe. All of the tested sites were unsafe, so there is no data regarding false positives.

VMM-Based Sensors for Monitoring Honeypots

The nature of a honeypot has led to the adoption of virtualization. Using virtualization allows for honeypots to be destroyed when compromised, and to be restarted from a saved state without going through the overhead of rebuilding the honeypot. This has led to Virtual Machine Monitors (VMM) being used in the implementation of honeypots. In addition, the VMM can also be used to monitor the honeypot for malware.

The two most common ways to monitor a honeypot are by examining network traffic, or by placing a monitoring application within the honeypot. Using network traffic to monitor for attacks limits the amount of information that can be gained due to the lack of specific system information that is obtained. Placing the monitoring application inside the honeypot allows for more thorough data collection and thusly more fine-grained intrusion-detection. However, it also puts the monitoring application in a vulnerable position. If the system is compromised, it is possible for the monitoring application to also be compromised, losing all meaningful data. Using the VMM to place sensors[52] that monitor the kernel of the honeypot has the advantage of being able to get detailed data from the honeypot, but removes the vulnerability of being accessible through the honeypot. The virtual machine barriers allow for the sensors to monitor the honeypot from a location inaccessible to any potential malware within the honeypot.

A design decision was made to only monitor the kernel for malicious activity based on two assumptions. The first was that the monitors should be lightweight and applicable to various operating systems and applications. The second was that any meaningful attack that is aimed at making persistent changes must involve the kernel in some way. To achieve efficient kernel monitoring, "invocation points" were established in parts of the kernel execution code where a trap is inserted. When an event of interest occurs, an interrupt is triggered.. The interrupt handler of the VMM is then modified to call the appropriate event handler. The event handler investigates the state of the kernel and any information available through the trap. It determines if more invocation points should be created, alarms should be raised, the event should be logged, or if execution should continue as normal. If execution should continue, the trap is temporarily removed and replaced with the original execution instruction. The kernel then resumes execution at the step just prior to the trap being triggered. When execution has passed where the trap was, the trap is replaced so that the same event will cause another interrupt. The traps and associated event handlers are referred to as sensors.

An example of a sensor is given that operates with *xinetd* configuration files. The observation was made that this configuration file is often modified to associate a shell with access to an opened port so that an attacker can connect to and control the compromised computer. The trap is inserted to detect whenever the /etc/xinetd.conf is opened for modification. When the trap is triggered, the event handler examines the registry, temporary variables, and the symbol table to determine if the modification is malicious. Based on the outcome of these variables, the execution is permitted or denied. An example of this sensor, including the trap and the associated event handler, is available in Appendix A.

A small amount of sensors can provide a large amount of security. For the evaluation of this prototype, only four sensors were used. The first was a socket sensor. The invocation point was in the sys_bind handler that controlled what processes were listening on what points. The event handler ensured that only authorized processes were binding to the appropriate authorized port. The second sensor detects sensitive inode access by putting the invocation point in the virtual file system subroutine used to find the directory entry for the accessed file. At this location, the event handler can accurately get the absolute path of the file and the associated inode. The sensor raises an alarm if the file is listed as sensitive, but has a mismatched inode with the one on file, or the file is sensitive and is being opened for modification. The third sensor monitors for stream redirections redirecting stdin, stdout, and stderr to a socket. The invocation point is within the exec system call. The sensor first determines if the spawned program is an interactive shell. If it is, the sensor checks the file descriptors to determine if redirection is to a socket. The sensor then checks to determine if the socket is directed to an external IP to ensure local programs interacting with each other do not trigger false-positives. The final argument capture sensor records arguments passed to spawning processes of interest. To do this, two different sensors are used. The first sensor is in the exec system call handler to determine if the spawning process is one of interest. If the process is of interest, the sensor activates another sensor that is stored within the loop. It copies all of the elements from the argument's array to the new process's stack. Once the initial arguments array has been recorded, the sensor is deactivated to prevent the sensor from unnecessarily recording all information about the running process. By capturing these arguments, it is possible to find what machines are being externally accessed. This could be used to determine if malicious tools are being loaded on to a compromised machine, where those tools are coming from, and what invocation points are being used to get the tools.

Using a Sandbox to Find Hidden Code in Packed Binaries

Monitoring code execution and reverse-engineering techniques yield information about

malware that is valuable in identifying and stopping it. To increase the impact of their malware, attackers have utilized different packing techniques to hide the malicious code. Compression and encryption techniques can be used to obfuscate the malicious binary. The only readily available execution instructions are those that unpack the hidden portion. There can be several layers of encryption of compression, causing the package to load the unpacking routines into memory, then go through a cycle of unpacking and reloading memory instructions until the malicious binary is executed. The several layers of embedded information make accurate disassembly and reverse-engineering difficult.

Instead of trying to determine the malicious contents by reverse-engineering the package, the fact that at some point the malicious code has to be loaded in to memory can be utilized. The malware can be allowed to execute in a sandbox so that it can be monitored to find the malicious segments. The Renovo[55] project finds the malicious portion of packed executables by monitoring memory during execution in a sandbox.

The sandbox uses a fine-grained emulation environment that allows for step by step code execution. In particular, the execution environment is looking for instructions that change the flow of execution. The section of code that is responsible for unpacking the malicious code is referred to as the hidden layer. Many hidden layers can exist to further obfuscate the binary. When the execution flow changes from a point that was within the bounds of the initially loaded memory to an area in memory that was written to after initially loading the instructions, the execution flow has changed layers. The location of the instruction pointer in the newly written section of code is the entry point to the next layer. To monitor the change of layers, the memory is monitored. When the program code is initially loaded, there is a section of program code, a section of packed data, and a section of reserved empty memory. When the reserved memory has not been written to, it is considered clean. When that memory is written to, the areas that were written are dirty. When the instruction pointer moves to a dirty region, that location is the entry point to a new layer that is potentially the packed data. This process is illustrated in Appendix C.

A table, similar to the memory page table, is maintained and referred to as the shadow memory. This table associates a bit with each memory address that keeps track of clean and dirty memory. Within the emulated execution environment, kernel modules monitor the creation and execution of processes. This allows Renovo to continue to monitor memory addresses associated with the malicious binary even if it spawns different threads to unpack the executable. Once the dirty memory has been accessed and the entry point recorded, the shadow memory can be wiped to clean and the process restarted to continue going through several hidden layers. To prevent an infinite loop, a time-out period can be set at which point the sandbox will stop analyzing the executable. This prevents the malware from just continuing the cycle to waste system resources. It also stops the monitoring of memory in the event that the malware is not packed in a hidden manner. If it is an executable with no additional layers, it is possible that no dirty memory will ever be accessed.

This approach to monitoring the execution environment based on access to recently-written memory addresses allows for a dynamic approach to unpacking malicious executables. The sandbox environment ensures that it is done in a safe manner that cannot spread to the host machine. This same process could be used for testing unverified software to monitor where the memory writes to and where it executes from.

Defensive Techniques: Aiding Malware Detection

Virtual Introspection

In order to virtualize the operating system environment, all semantic information regarding the operating system has to be stored on the hosting machine and manipulated by the Virtual Machine Monitor (VMM). Components like RAM, running processes, or registry information is stored in a manner that is accessible by the host of the VM. When doing a forensic analysis looking for malware, it is often difficult to analyze a live machine because the infection could have software that destroys the system when it detects that it is being scanned, or have stealth-like qualities to hide it from the analysis. When doing a forensic analysis, third party tools have to be used to avoid this, but it often leads to a forensic copy being made of the hard drive(s) and examined on a different machine. The most common way of doing this is to shut down the machine and then copy it over. However, it can be done while it is running using network utilities – the downside is that it is possible to modify the system and lose vital evidence. Any modifications made to the system could ruin the presentation of evidence in a legal setting.

Due to the way the VMM stores semantic information on the hosting system, it is possible to perform virtual introspection[7] of a compromised VM without modifying its behavior. All of the tools needed to extract volatile information that would be lost when the computer is shut down (open ports, running processes, loaded modules, etc.) can be used without modifying the VM. The information stored on the hosting system can be read and reconstructed in the same manner the VMM reads and reconstructs it for use within the VM. This read-only approach would not modify anything within the VM. After it has been reconstructed so that the live data from within the VM can be examined outside the VM, this data can be analyzed without the fear of tampering with evidence. Virtual introspection would allow for the examination of volatile information without the concern of altering information or ruining evidence. Once the volatile information has been recovered from the VM, the forensic process can occur as usual. Not only is this beneficial to forensic investigations, it is also beneficial to malware detection. It allows for a suspect system to be analyzed in an environment where modifications that could possibly alert the malware to forensic activity can be made without concern.

Semantic View Reconstruction

Stealthy viruses have an advantage when facing traditional malware detection systems. They can see how the malware detection system is operating because they are running within the system that they are attacking.[2] This means that malware can detect the various anti-malware processes and try to disable them while gaining control of the system. The malware detection system also has the advantage of gaining a semantic view of the system due to running “in the box”, or within the system. This allows the malware detection software to examine running processes, files, or kernel modules , which aids in their detection capabilities. If instead of the operating system running locally, it was run inside a virtual machine, it would be possible to run the malware detection system outside of the system, thereby gaining an “out of the box” view of the system. This would also remove the malware's capability of examining the malware detection systems before trying to launch its payload. The problem is that by running “outside of the box”, the malware detection system loses the semantic view of the hosted system. Instead of seeing running processes, the malware detection system sees memory pages, registers, and disk blocks.

To gain the benefit of both worlds, the VMwatcher works with the Virtual Machine Monitor

(VMM) to reconstruct a semantic view of the VM.[2] The goals of the VMwatcher are to unobtrusively inspect the VM for low level (VMM level) state information without using anything within the VM. Then, it reconstructs the semantic view of the guest externally so that a malware detection system that could run inside the system could be run outside of the system on the reconstructed view without any modifications. The goals are generalized so that the VMwatcher is generic enough to work with different VMMs (VMWare, Xen, UML, etc.) and different host operating systems. The prototype was implemented on a Windows and a Linux system successfully.

The VMwatcher operates under the assumption that the VMM can provide high levels of isolation between the VM and the hosting system. This means that it is possible for the VM to be highly contaminated with malware but the hosting machine remains uncontaminated because nothing can be passed between the VM and the hosting machine. Under this assumption, if malware targets the VM specifically for the purpose of detecting and evading malware detection systems, it will not be able to evade the VMwatcher. The VMwatcher should not even be detectable by the malware, and any modifications the malware makes to the VM should not effect how VMwatcher operates (such as rootkits). This makes detecting stealthy malware much easier. No operations from the VM are used for semantic reconstruction; all of this information is provided within the host file system so that the VMM can load the VM. Using this same information, it is possible to recreate the view from inside the VM outside of the VM. With this reconstruction, standard malware detection applications can be run on the reconstructed view without being hindered by the effects of the malware. For example, if a rootkit were to be installed to prevent malicious processes from showing, the view outside of the box is reconstructed from the files running the VM, so this information cannot be hidden from the VMwatcher. This gives the VMwatcher the capability to allow malware detection systems to run without being affected by malware.

Using the VMwatcher, new malware detection techniques can be used. Aside from being able to operate on a view of the host without being modified by the host, a comparison analysis can be done. To detect stealth malware, a system scan can be done inside the box and outside, then the results can be compared. Anything differing between the two would likely be caused by stealthy malicious software. The view comparison can be done on a system level or a lower file or process level. Due to the capabilities of the VMwatcher to reconstruct the semantics of the VM, any internal component (volatile memory, NIC properties, swap information, etc.) can be analyzed both inside and outside the box, and the results can be compared.

The prototype created was able to run with four different VMMs: VMware, QEMU, Xen, and UML, and it supported variations of Windows, Red Hat, and Fedora Core for semantic reconstruction. 10 different Windows rootkits and a dozen different Linux rootkits were used to test the comparison-based methodology. This methodology attacks the nature of rootkits by showing what has been hidden inside the box. The prototype was able to detect all rootkits by scanning inside and outside the box, then comparing the results. The scans were done with standard unmodified malware detection software (Symantec AntiVirus, for example). If the malware detection software was able to detect the malware in the box, then the benefit is irrelevant. However, the rootkits were able to hide themselves from the malware detection system inside the box. When the same scan was done outside the box with the same malware detection system, the rootkits were found.

Controlled Program Execution Using Manitou

The hypervisor controls access to hardware from either below the operating system (intercepting system calls between the hardware and OS) or above the operating system (intercepting system calls between software running in the OS and the OS). In order to control what has execution

privileges on a system, the Manitou[1] project implemented the hypervisor below the operating system between the OS and the paging calls to the processor.

Most major processors (including AMD and Intel) have implemented the paged memory and per-page executable and writable bits. Manitou leverages these bits to determine what is permitted to run. In order to remain generic enough to run with any OS or supported software, Manitou examines each memory page as it goes to the processor. In doing this, Manitou does not have to understand the semantics of the OS or the software it supports. Instead, it runs in the hypervisor below the OS, monitoring each request sent to the CPU.

To determine if a request is valid or not, a cryptographic hash is taken of the page. The hash is then stored in a table. If the hash of the page appears in the table, the page's executable or writable bit is set according to whichever the page is requesting. Manitou only allows execution or writing; it does not allow both. This is done to prevent the execution from modifying something maliciously to then allow non-approved pages to execute malicious code. If the hash is not found in the table, the execution bit is not set and a page fault is returned to the OS. The OS then handles this in its own manner, usually by terminating the faulty application.

There are a few suggestions to determine if a page's hash should be allowed in the table or not. The first is to have a signature-type body governing the software. If it is digitally signed by a supported body, then the hash is allowed in the table. This could be convenient for large companies or someone with a network large enough that a Public Key Infrastructure can be established. For a home user or a situation where that is not optimal, the user is allowed to add a hash to the table. This is done through an isolated channel between the user and Manitou instead of being managed by the OS. The goal is to have no interaction with the OS in case it has been compromised. A user can assign a specific group to the hashes being added to the table so that they can all be correlated to a specific application. This makes it easier to manage the table at a later time.

If a hash is added to the table that is determined to have negative consequences to the system, the hash can simply be removed from the table. This does not remove the software from the system but it does revoke its execution and write privileges. It will remain on the system, but will be incapable of accessing the system in any way. The ability to revoke an application's privileges allows for users to make mistakes when installing software. The current paradigm of software installation makes it difficult to get rid of malware once it has already been installed. By using virtualization, the user is permitted to make mistakes that can compromise the system, recognize the error, and then revoke that application's rights. Once the application can no longer access the system, removal of the malware can be performed in an easier manner than if it was still permitted to access the system.

Program Isolation Using Solitude

Instead of implementing the hypervisor below the OS to limit access to hardware (such as with Manitou), Solitude[5] creates an isolation file system (IFS) that can be generated on demand for any application that will isolate the application from the base file system. In a manner very similar to the FVM, Solitude uses a copy-on-write scheme to create different isolation environments for the applications. Using the assumption that most malware enters a system from a network connection, Solitude suggests creating separate IFSs for each application that accesses the network. One of the differences between Solitude and the FVM is that Solitude has a policy for sharing files between different IFSs or an IFS and the base system.

The copy-on-write mechanism contains all writes that try and access the base system inside the IFS. If a program later needs to be removed, the IFS can be removed and the base system will be unchanged. Copy-on-write also limits the propagation of attacks through shared updates. Outside of

providing a safe sandbox-type environment for untrusted applications to run in, Solitude also aims to create an environment where post-intrusion analysis and recovery is simplified. The last benefit of copy-on-write is that configuring read access is not necessary when reading is allowed between the IFS and the base system. Reading is such a common (yet fairly secure) occurrence that properly configuring read access to every part of the base system would be a difficult and error-prone process. While read access is allowed to the base system, Solitude still offers the functionality to block read access to specific parts of the base system when necessary, if explicitly supplied by the user. To accomplish the IFS isolation model, the isolation environment is mounted in a predefined portion of the file system. The untrusted application to be "jailed" is then *chroot*'d to the isolation point. The application's privileges are then restricted based on how they would have been in the base system. If it normally would not be allowed root access, then it is not allowed access to the base of the mount point. The Vserver secure chroot barrier is then added. This contains a special flag on the parent directory that helps to prevent the jailed application from escaping.

Maintaining an IFS for varying applications helps to stop cross-contamination of software. For instance, if a piece of software is installed that contains malware to exploit a vulnerability in a web server, the malware will not have access to the web server and will not be able to contaminate it. However, there are many instances where a jailed application needs to have access to other applications. For instance, if newly downloaded imaging software is installed in one IFS, an image created by this software would be contained within the IFS. But if a different viewer resides in another IFS, the viewer would not have access to the file. There are two solutions that Solitude provides to get around this problem. The first is to simply install both applications in the same IFS. Then each application would have the same access rights and would be able to view files created by either application. The second option is to use the file sharing policy. It is possible to allow a file access to the base system from within an IFS. To keep the policies simple, there is no mechanism to allow for sharing directly between different IFSs. Instead, the sharing has to be done to the base system and two different IFSs can share the content through the base system. Six different sharing modes are available - three for reading and three for writing.

The default read mode (Rshare) allows for transparent reading from the base system. When the application in the IFS tries to update the object being read, it is copied into the IFS and the copied version is modified. The Rsnapshot mode allows for copy-on-write to be maintained in both directions. As soon as the IFS is created, the file being accessed is copied in to the IFS before being modified. All further modifications made by the jailed application are done to the copy in the IFS. Conversely, all modifications done by the base system to the file remain in the base system and are not made in the copy maintained in the IFS. The third option is Rdeny. This hides the particular file in the base system from the IFS so that the jailed application does not know of its existence. An Rdeny overrides any of the write settings. The Wisolate mode is the default write mode. It employees the standard copy-on-write policy that isolates all modifications to the IFS environment. The Wcommit option allows for a file in the IFS to be committed back to the base system at a later time. The commit can only be initiated by the base system to prevent malicious software from committing its self. The Wshare option allows for immediate writing to a file in the base system. This is to be used with applications that are trusted, or to applications writing to terminal sessions or directories like /dev/null.

The sharing policies of Solitude allow for the potential spread of malware. To make the propagation of malware trackable, easier to analyze, and then easier to eventually recover from, Solitude implemented a taint policy. If a file transitions from the IFS to the base system it is a tainted file. If a process interacts with that file, the process becomes tainted. Any file that process then interacts with is marked as tainted and all children processes of that process are considered tainted. The tainting algorithm works at the kernel level and is capable of marking any kernel objects as tainted.

Once an object is marked as tainted, it becomes logged. Any files that are committed are logged, and the processes that access files to become tainted (or taint a file) are logged so that the taint propagation can be analyzed. None of the interactions that are isolated to the IFS are logged. The assumption is that if anything malicious happens within the IFS the entire environment can be removed to remove the malware. The purpose of the logging system is largely for analysis and later recovery. When the taint begins through a commit there is a commit ID and an IFS ID that is also logged in the records. This allows an analyzer to determine which IFS caused the malware propagation.

To recover from malware, the first option is to remove the IFS environment. This will remove all changes the malware has made to the system if it was not allowed to write to the base system. However, if it was allowed to write to the base system, the changes can still be undone. The commit and IFS ID can be supplied to generate a taint dependency graph. The same taint algorithm used to determine if a file should be tainted can be used to find all files and processes that were tainted because each execution of a taint rule is logged. It is then suggested that each file be inspected for malware. A previous project referred to as Taser can be used to roll back files to a snapshot before the taint was registered to automate the process. The snapshots are taken as soon as the IFS requests the file.

Collecting Malware with Honeypots

The Nepenthes[45] honeypot takes a low interaction approach to honeypots, but incorporates a modular design to allow for more flexibility. The Nepenthes honeypot emulates vulnerable services. It uses a modular design so that services can be changed depending on the network traffic they interact with. The core Nepenthes daemon manages network connections with other modules. The other types of modules include vulnerability, shellcode parsing, fetch, submission, and logging modules. The vulnerability modules emulate the vulnerable services that are to be attacked. The shellcode parsing modules then parse the incoming payload to the vulnerability modules to extract the malware information. From the extracted information, the fetch module downloads the malware from a remote location. The submission module then determines how the fetched malware is to be stored (on a hard disk or removable media, database, network location, etc.), while the logging modules gather information about the emulation to get an overview of malware attack patterns.

The modular approach allows for Nepenthes to be efficient locally or in a distributed environment. It was demonstrated emulating 16,000 IP addresses on one physical machine. The vulnerability modules also allow for the creation of custom modules with specific reactive, logging, or reporting mechanisms. For instance, a general FTP server can be represented on a port. Then, depending on how the attacker interacts with the port, the vulnerability module can display itself as a Unix protocol or a Win32 protocol.

Worms can be detected by an Intrusion Detection System (IDS) due to the port scans they perform to find more vulnerable targets to infect.[46] This makes detecting many worms trivial to an IDS. For example, the Blaster worm[48] triggered a Snort[49] alert before a signature was created to detect the worm.[46] Bots differ from worms in that they do not automatically perform a port scan after infecting a computer. They wait dormant until they receive a command from Command and Control (C&C) servers. These commands typically come over an IRC channel. This makes detecting bots more difficult than detecting worms. Appendix B demonstrates the number of C&Cs operating over selected IRC ports that have been closed due to bots.[47]

Due to the flexibility and customization available with Nepenthes, it is much easier to detect malware, including bots.[45] The emulation of vulnerable services allows Nepenthes to monitor IRC channels as well as network traffic to detect C&C commands, as well as port scans. The emulated modules allows for efficient detection of malware. After detection, the malware can be collected and

stored in a customized manner. The binaries can then be sent to a sandbox to be analyzed. The binaries can also be scanned by standard anti-virus software to determine if the current signatures can detect the malware. The binaries collected by Nepenthes were scanned by four different commercial anti-virus engines. Figure 5 shows the percentage of malware identified based on the total number of malicious binaries collected and the malicious binaries collected in the last 24 hours.[45]

	AV Engine 1	AV Engine 2	AV Engine 3	AV Engine 4
Complete set (14,414 binaries)	85.0%	85.3%	90.2%	78.1%
Latest 24 hours (460 binaries)	82.6%	77.8%	84.1%	73.1%

Figure 5: Four anti-virus engines used to scan collected malware. The first row depicts the percentage of malware detected from the entire set of collected malicious binaries. The second row shows the percentage of malware detected from binaries collected in the last 24 hours. All binaries collected were malicious; the percentage not detected represents malware the anti-virus scanner could not detect.

The customization allowed in containing and storing malware permits many different analysis techniques that can yield useful statistical information regarding the network. This can be used to determine if any malware is currently traveling through the local network, or the types and percentages of malware attacking the network externally. It can also be used to determine if attacks are coming from a specific source, or what areas of the network are most prone to attack. The modular design of vulnerability emulation could be used in other ways to raise alerts in a production environment when an attack is occurring.

Comparison of Defensive Techniques for Specific Malware

Table 1 below matches types of malware with the studied defensive techniques. The columns represent the specific types of malware examined in this study, while the rows are the specific computer defense mechanisms examined. The intersection of the columns and rows displays how well the defensive mechanism defends against the specific type of malware. A "✓" means the defensive mechanism works well against the malware. An "X" means the defensive mechanism does not defend well against the specific type of malware. Some of the defensive mechanisms studied allow for a new approach to malware detection, but are not actually a malware detection system. In the cases where a defensive technique would increase the malware detection capabilities of a third party malware detection system, it is marked with a "+". Defensive mechanisms that can be utilized to benefit a system after malware has compromised it are marked with a "-".

When matching specific types of malware to a specific defensive mechanism, some correlations do not fall perfectly into one category. When this occurs, a number accompanies the rating in a subscript that correlates with an explanation below the table. The ratings given assume that sufficient information is supplied to the malware detection system. For example, it is assumed that signatures are supplied or that heuristic approaches have collected enough data to build accurate baselines.

	Viruses	Worms	Logic Bombs	Trojan Horses	Rootkits	Exploits	Spammer Programs	Keyloggers	Adware	Spyware
Virtual Introspection$_1$	+	+	+	+	+	X	X	+	+	+
SpyProxy$_2$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Semantic View Reconstruction	✓	✓	X	✓	✓	X	X	✓	X	X
Controlled Program Execution$_3$	X	X	-	-	X	X	-	X	-	-
Program Isolation$_3$	X	X	-	-	X	X	-	X	-	-
VMM-Based Sensors$_4$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Vulnerability Emulation	✓	✓	X	✓	✓	✓	X	✓	X	X
Signature Scanning	✓	✓	✓	✓	X$_5$	X	✓	X	✓	✓
Heuristics$_6$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: A comparison of malware defensive techniques and selected malware.

1: Virtual introspection allows for a malware detection system to work without any hindrance from the operating system or malware. This makes it an ideal addition to any malware detection system looking for malicious binaries in the operating system. In particular, it would help identify rootkits that could hide themselves from a malware detection system running from within the compromised operating system.

2: SpyProxy loads webpages in a virtual machine. All prompts are answered yes and all executables are run. Diagnostics in the virtual machine then determine if anything malicious is occurring. Malicious binaries and the other types of studied malware can be delivered through a malicious website. Given ideal circumstances, including appropriate diagnostic tools in the virtual machine, SpyProxy could detect all of these malware if they were delivered via web page.

3: Program isolation and controlled program execution can stop malware from running or spreading on a compromised machine. However, these techniques work on executables that the user has given permission to run or identified a unique application layer for. If the malware accessed the system through an attack surface that did not put it within the scope of these defense techniques, such as an exploit, they will not be able to help.

4: VMM-based sensors have the same advantage the SpyProxy project has. They can monitor a virtual machine for any malicious changes at the kernel level in a way that cannot be interrupted by malware. The execution of a malicious binary or exploitation of a vulnerable process can be detected with a properly placed trap. However, it requires that the operating system is running within a virtual machine. It does not trap malware within the native operating system.

5: Signature-based detection methods detect malicious binaries by matching a sequence of characters in the binary with a known malicious sequence of characters. It is possible to detect rootkits with this same manner. However, rootkits are difficult to detect with a signature-based scanning method because they can hide very well from the scanning mechanism.

6: A heuristic malware detection technique could detect any malware that causes the system to measurably change it's working conditions. However, in order to detect the specific types of malware studied different metrics would need to be monitored. For example, network metrics (bandwidth usage, network connections, etc.) would need to be monitored to detect a spammer program while system metrics (running processes, memory usages, etc.) would be needed to detect a virus.

Suggested Methods of Securing Unverified Data

Researching current and past malware strategies has shown many possible areas of attack. A computer on the Internet is exposed through many different interfaces. Any process that is monitoring a port is susceptible to attack. The user's interactions with resources provided by others also opens up many attack surfaces. Browsing the internet, reading e-mails and interacting with attachments, downloading content from third party providers, executing downloaded content in potentially vulnerable software, and exchanging physical storage mediums can lead to compromising the integrity of the computer. In order to meet the needs of a constantly changing aggressive environment, a security system based on verifying data before giving it full rights to execute should meet the following criteria:

1. All attack surfaces should be secured. Any data that interacts with a system should be checked before being given full privileges.
2. The system should be fully customizable. Security systems can fall behind because new attacks and threats target areas that are not yet secured, or cannot be secured using current technology. A customizable system should allow for the fact that new threats will be recognized that cannot be sufficiently guarded against using old techniques.
3. The security system should be based on a modular design. This will allow for a customizable system where new attack surfaces can be secured by creating a new module. In the event an attack is created for a known interface that cannot be addressed by the current anti-malware mechanisms, the anti-malware module for that interface can be updated to compensate for the evolving attacks. A customizable modular design allows for the system to be adaptive.
4. The modular approach to verifying unknown data should be based on the attack surface of the type of data. Instead of developing a security mechanism that governs all aspects of a computer (such as signature scanning), focusing on the attack surface can allow for more specific defense mechanisms. For example, installing a plug-in from an unknown web site should be treated differently from securing network communications over an FTP port.
5. The needs of the user should be considered. A verification system will differ between a home user and a corporate environment. The system should be scalable to meet the needs of different users. Higher security levels with additional modules might be necessary for some users, while others might only want a minimal level of security with the fewest modules possible.
6. There has to be room for error. In an ideal system, malware would be detected with a perfect accuracy rate and no false positives. However, it is still possible that a user would want to bypass security features to allow malware onto their system. In this event, there should be a way to undo the harm the malware has caused. This allows for user and system errors.
7. General security practices should not be overlooked. Increased security can be gained by creating a system that uses specialized security mechanisms based on the type of data and interface. However, a general security system, such as a signature-based solution, can still be built on top of this system to add an additional layer of security.
8. No security system can be implemented effectively without educating the user. User education is as important as a good security system. If a user decides to use a security system in a manner that is not intended, or ignore warnings, the computer system can still become compromised.

These criteria would allow for specific security systems based on the needs of the content and users it is securing. More importantly, as new attack vectors are created due to expanding technologies, specific security implementations can be added that do not negatively impact execution overhead. Each security module is only executed for a particular interface, so every defensive strategy is not employed

for a specific interface.

Securing Internet Browsing

One of the most difficult challenges is that vulnerabilities can be exploited through many different attack surfaces. Of those attack surfaces, some can be exploited in vastly different ways, such as Internet browsers. The SpyProxy project[6] offers an interesting approach to creating an environment in which it is safe to browse the web. Due to the varying mechanisms a web page can exploit a vulnerability in the user's computer (plugins, flash vulnerabilities, image vulnerabilities, malformatted web pages, etc.), it is best suited to a sandbox technique. The SpyProxy project loaded the questionable web page in a virtual environment, then browsed through the pages, installing any requested software, and monitored for any malicious changes to the virtual environment. If anything negatively impacted the virtual environment, access to the web page was denied. This strategy is effective when it is difficult to develop a defense due to the large attack surface. In addition to monitoring for trigger events, sensors[52] in the Virtual Machine Monitor (VMM) can be used to increase the effectiveness of the sandbox. This would allow the web page to be loaded in an environment where all interactions that the web page makes with the computer can be monitored at a kernel level.

For purposes of browsing, it is not critical that the type of malware be identified. The importance is effectively determining if it is safe to browse the website. Testing the web page in a virtual environment does not identify what kind of malware is attacking the host, but can trigger alarms if anything out of the ordinary is happening. If no alarms are triggered, the web page can be permitted to load on the user's machine. Due to the sensitive nature of a web browser, and the difficulties in creating efficient traps for all vulnerabilities, the web browser on the local machine can work within an isolated execution environment[5] to add an additional layer of security. A lightweight virtualization of the browsing environment, such as using the Featherweight Virtual Machine[3] (FVM), can yield a more secure environment.

If the browser is isolated to its own virtualization environment, and the browser is compromised, the virtualized environment for that session can be removed. Any data that is sent from the Internet to the browser should be isolated within the virtualization environment so that any malware is removed with the virtualized execution environment. In the event data needs to leave the isolated environment in order to operate properly (such as downloading media to be played in a different player), the data execution can be monitored with a system such as Manitou[1]. When data is passed from the browsing environment to the base machine for purposes of sharing it across different applications and processes, it can still not be fully trusted. To prevent malware from spreading in this manner. a system like Manitou can allow it to execute with full privileges, including potentially infecting the base machine. Once the infection has been detected, Manitou can revoke the execution privileges of the malware. This does not remove it from the machine, but it does prevent it from doing any further harm. From this point, the system can be restored back to a safe state by backtracking all of the changes the monitored malware made.

Securing E-Mail

E-mail is difficult to secure for the same reasons as securing Internet browsing. When attachments are taken in to consideration, the attack surface is very broad. A Trojan or worm that spreads through e-mail attachments can take on the guise of any program. Due to the unpredictability of attachments and links in phishing emails, the SpyProxy/FVM/Manitou[6][3][1] strategy for securing

web browsing can be extended to e-mail. The functionality of e-mail and embedded content has increased to the point where it is difficult to draw a distinct line between Internet browsing and e-mailing. Both can be associated with potentially malicious downloaded content that is hard to detect without first executing it in a safe manner to determine exactly how your system will react. Using SpyProxy/FVM/Manitou or like programs creates a safe execution environment to initially pass an email through. The SpyProxy design can load the e-mail in a virtual environment identical to the host computer where the e-mail will ultimately be received. If no alarms are triggered in this environment it can pass to a virtualized execution environment on the host machine. If there is an attachment, it can also be loaded and executed in a SpyProxy-type environment before passing it to the host machine. This will allow for preliminary testing for obvious malicious behavior and fine-grain monitoring in a controlled execution environment. If the attachment successfully moves through all of the layers of security, the sender can be added to a white list. Instead of maintaining a black list that denies anyone on the list, a white list can be maintained that reduces the security requirements for users on the list. Attachments can still be monitored in a Manitou-like database so that execution privileges can be removed if malware is detected from someone on the white list. All protections should not be removed when someone is white listed to account for the possibility of a friendly user being compromised and used to spread malware.

Securing Downloaded Content

Content that is downloaded from a third party resource, such as peer to peer media, can also contain embedded malware. However, due to potentially large volumes of data and properties that can be assumed, it is possible to efficiently rule out malware without using a system as large as a secondary emulation environment that passes data to a primary emulation environment with controlled execution monitoring. Downloaded content can be broken into two different categories: executable binaries and data. Due to intrinsic differences, the two should be handled differently.

Executable binaries on a Windows platform can be parsed with a system like the Intelligent Malware Detection System[10] (IMDS). This takes advantage of the fact that most executables follow a well-known format. In addition to a known execution format, there is a distinct measurable difference between malware and legitimate binaries. At some point, the malware has to change the flow of execution, launch a new process, or modify an unusual location in memory in order to compromise the system. By logging the execution instructions and comparing malware to legitimate software, differences emerge that allow for accurate malware detection. In the event the software is being downloaded to a particularly sensitive system, it is possible to limit program execution to an isolated execution environment (such as the one offered with Solitude[5]). However, it should be possible to sufficiently rule out the possibility of malware by parsing the execution instructions to determine exactly how the executable will act on the system.

Non-executable data files have to be treated differently because it is more difficult to parse them due to execution differences based on the intended application. To rule out immediate threats, the methodology used in the Nepenthes[45] honeypot could be applied here. Instead of emulating the entire application, the vulnerable portions could be emulated to see how the data file reacts. The Nepenthes honeypot allows for the creation of vulnerability modules that act like a vulnerable service and can be customized based on the needs of the user. A similar emulation technique could create an environment that mimics a programs known vulnerabilities. The program could then begin the process of loading the file into the emulated environment. If any traps are triggered during the load due to vulnerabilities in the emulated environment, then it can be assumed that the data file is malicious. For more complex programs, a different approach is possible. A local virtualization environment can be

used to monitor program execution with kernel level traps and event handlers such as those used to monitor VMM sensors[52]. The application would be permitted to execute as normal, but kernel level traps would be inserted that launch specific event handlers if something unusual occurs. These event handlers can then check the current semantic view[2] of the machine to verify its integrity. If something malicious is occurring, execution can be denied and the file can be considered malicious.

Securing Open Ports

Before looking at the processes monitoring each open port, it is important to establish which ports should be opened and which ones should be closed. Any unnecessary ports and associated processes or applications should be closed so that only mandatory ports are opened. At each needed open port, an emulated process, such as those used in Nepenthes[45], can listen for network connections. The emulated process can emulate the vulnerabilities of an unpatched or unprotected service. If anything compromises the service, then that networked location can be considered malicious and further communications can be denied for a predetermined amount of time.

If no vulnerabilities are abused, the network communications can be forwarded to an internal process that would normally be monitoring the port. This would allow for full functionality and vulnerability emulation. The “real” service can be nested within its own execution environment[3][5] so that communication can be logged and monitored. Much like the previous execution environments, sensors[52] can be placed in the isolated execution environment to ensure that no malicious behavior is occurring. If the entire transaction completes in a safe manner, the remote location could be added to a white list much like a trusted e-mail or Internet location so that security can be lowered for the next transaction. Lowering the security level could mean letting the network communications start with the real process instead of going through the emulated vulnerable process, but maintain execution in an isolated environment where traps can still be triggered.

Conclusion

Verifying data in a safe manner creates a more secure computing environment. Classically, signature-based malware detection methods were used to scan for malware. These techniques were initially successful due to the generalized use and construction of computers. As computers, and their uses, have become more specialized, signature-based malware detection methods are not as useful. The attack surface of a computer has expanded significantly since the advent of signature-based malware detection methods. Web browsing, e-mail, networked computers, and networked software have created many different attack surfaces. Using one general method to secure all of these, such as signature-based malware detection, is becoming less reliable. The most successful computer security techniques are those that attempt to secure specific attack surfaces[1][2][5][6], as opposed to general computer security systems that hope to achieve security across all surfaces with the same method. To create a reliable computer security system, unverified data should be thoroughly scrutinized before it is permitted to access a computer. The data should be analyzed based on which attack surface it was obtained from. For example, data obtained from web browsing should be verified differently than data obtained from a peer-to-peer program. A computer security system that creates a defensive mechanism based on each attack surface of a computer will yield a more reliable computing environment.

References

[1] L. Litty, D. Lie. Manitou: A Layer-Below Approach to Fighting Malware. *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, October 2006.
[2] X. Jiang, X. Wang, D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, October 2007.
[3] Y. Yu, F. Guo, S. Nanda, L. Lam, T. Chiueh. A Feather-Weight Virtual Machine for Windows Applications. *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, June 2006.
[4] W. Armitage, A. Gaspar, M. Rideout. Remotely Accessible Sandboxed Environment with Application to a Laboratory Course in Networking. *SIGITE '07: Proceedings of the 8th ACM SIGITE conference on Information technology education*, October 2007.
[5] S. Jain, F. Shafique, V. Djeric, A. Goel. Application-Level Isolation and Recovery with Solitude. *ACM SIGOPS Operating Systems Review, Volume 42 Issue 4*, April 2008.
[6] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. *University of Washington, Department of Computer Science & Engineering*, May 2007.
[7] B. Hay, K, Nance. Forensics Examination of Volatile System Data Using Virtual Introspection. *ACM SIGOPS Operating Systems Review, Volume 42 Issue 3*, April 2008.
[8] J. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, J. Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, June 2007.
[9] D. Venugopal. An Efficient Signature Representation and Matching Method for Mobile Devices. *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet*, August 2006.
[10] Y. Ye, D. Wang, T. Li, D. Ye. IMDS: intelligent malware detection system. *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, August 2007.
[11] D. Sanok. An Analysis of How Antivirus Methodologies Are Utilized in Protecting Computers from Malicious Code. *InfoSecCD '05: Proceedings of the 2nd annual conference on Information security curriculum development*, September 2005.
[12] G. Crescenzo, F. Vakil. Cryptographic Hashing for Virus Localization. *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, November 2006.
[13] Wikipedia. Malware. *http://en.wikipedia.org/wiki/Malware* accessed on August 10th, 2008.
[14] TechTarget. Logic Bomb.http://en.wikipedia.org/wiki/Sandbox_(computer_security) *http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci815177,00.html* accessed on August 10th, 2008.
[15] P. Mah. Major AV Vendors: Pure Signature-Based Approach Insufficient. *http://www.techatplay.com/?p=68* accessed on August 11th, 2008.
[16] P. Szor. The Art of Computer Virus Research and Defense. *Symantec Press*, third printing May, 2005.
[17] S. Lowe. IT Cost Containment: The Power Benefits of Virtualization. *TechRepublic. http://blogs.techrepublic.com.com/tech-manager/?p=437*, accessed on August 24th, 2008.
[18] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199609*, acessed on September 8, 2008.

[19] Shadowserver. Bot Counts. *http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.BotCounts*, accessed on September 8, 2008.
[20] BBC News. Zombie plague sweeps the internet. *http://news.bbc.co.uk/2/hi/technology/7596676.stm* September 4, 2008.
[21] Panda Security 2008. QUARTERLY REPORT PandaLabs (April – June 2008). *http://www.pandasecurity.com/homeusers/security-info/tools/reports/?sitepanda=particulares*, accessed on August 10th, 2008.
[22] Panda Security 2007. ANNUAL REPORT PandaLabs 2007. *http://www.pandasecurity.com/homeusers/security-info/tools/reports/?sitepanda=particulares*, accessed on August 10th, 2008.
[23] I. Mokube, M. Adams. Honeypots: concepts, approaches, and challenges. *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, March 2007.
[24] The Honeynet Project. Honeynet Definitions, Requirements, and Standards, version 1.6.0: updated 14 October, 2004. *http://www.honeynet.org/alliance/requirements.html*, accessed on September 12, 2008.
[25] The Honeynet Project. Know Your Enemy: Honeynets, last modified 31 May, 2006. *http://www.honeynet.org/papers/honeynet/*, accessed on September 12, 2008.
[26] K. Curran, C. Morrissey, C. Fagan, C. Murphy, B. O'Donnel, G. Fitzpatrick, S. Condit. Monitoring hacker activity with a Honeynet. *International Journal of Network Management*, March 2005.
[27] Panda Security. Virus. *http://www.pandasecurity.com/homeusers/security-info/classic-malware/virus/?sitepanda=particulares* accessed on September 21, 2008.
[28] Panda Security. *http://www.pandasecurity.com/usa/*.
[29] Panda Security. Worms. *http://www.pandasecurity.com/homeusers/security-info/classic-malware/worm/?sitepanda=particulares* accessed on September 21, 2008.
[30] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199848* accessed on September 21, 2008.
[31] Wikipedia. Microsoft Office. *http://en.wikipedia.org/wiki/Microsoft_Office* accessed on September 21, 2008.
[32] Wikipedia. Logic Bomb. *http://en.wikipedia.org/wiki/Logic_bomb* accessed on September 21, 2008.
[33] UBS. *http://www.ubs.com/*.
[34] Panda Security. Trojans. *http://www.pandasecurity.com/homeusers/security-info/classic-malware/trojan/* accessed on September 21, 2008.
[35] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199862* accessed on September 21, 2008.
[36] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199861* accessed on September 21, 2008.
[37] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199860* accessed on September 21, 2008.
[38] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=199859* accessed on September 21, 2008.
[39] McAfee. Keylogger.c. *http://vil.nai.com/vil/content/v_104141.htm* accessed on September 21, 2008.
[40] Symantec. Hacktool.Keylogger. *http://www.symantec.com/security_response/writeup.jsp?docid=2005-041516-0123-99* accessed on September 21, 2008.

[41] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=200218* accessed on September 21, 2008.
[42] BBC News. Fighting the scourge of scareware. *http://news.bbc.co.uk/2/hi/technology/7645420.stm*, accessed on October 3, 2008.
[43] B. Schneier. Schneier on Security: "Scareware" Vendors Sued. *http://www.schneier.com/blog/archives/2008/10/scareware_vendo.html*, accessed on October 3, 2008.
[44] B. Schneier. *http://www.schneier.com/*, accessed on October 3, 2008.
[45] P. Baecher, M. Koetter, T. Holz, M. Dornseif, F. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. *http://honeyblog.org/junkyard/paper/collecting-malware-final.pdf*, accessed on October 1, 2008.
[46] J. Riden. Using Nepenthes Honeypots to Detect Common Malware. *http://www.securityfocus.com/infocus/1880#ref5*, accessed October 1, 2008.
[47] Shadowserver Foundation. Stats – IRC Ports. *http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.IRCPorts*, accessed on October 7, 2008.
[48] Symantec. W32.Blaster.Worm. *http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99*, accessed on October 8, 2008.
[49] Snort. Snort – the de facto standard for intrusion detection/prevention. *http://www.snort.org/*, accessed on October 8, 2008.
[50] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=200798*, accessed on October 8, 2008.
[51] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=200911*, accessed on October 8, 2008.
[52] K. Asrigo, L. Litty, D. Lie. Using VMM-based sensors to monitor honeypots. *Proceedings of the 2nd international conference on Virtual execution environments*, June 2006.
[53] Panda Security 2008. QUARTERLY REPORT PandaLabs (July – September 2008). *http://pandalabs.pandasecurity.com/*, accessed on October 8, 2008.
[54] Wikipedia. Sandbox (computer security). *http://en.wikipedia.org/wiki/Sandbox_(computer_security)*, accessed on October 8, 2008.
[55] M. Kang, P. Poosankam, H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. *Proceedings of the 2007 ACM workshop on Recurring malcode*, November 2007.
[56] Panda Security. Encyclopedia. *http://www.pandasecurity.com/homeusers/security-info/about-malware/encyclopedia/overview.aspx?IdVirus=201161*, accessed on October 24, 2008.
[57] YouTube. *http://www.youtube.com/*

Appendix A: An example of a sensor used to monitor a virtual machine. The first section of code shows the trap used to trigger the event handler. The second section of code shows the event handler that the trap calls. These two pieces of code make up a functioning sensor.[25]

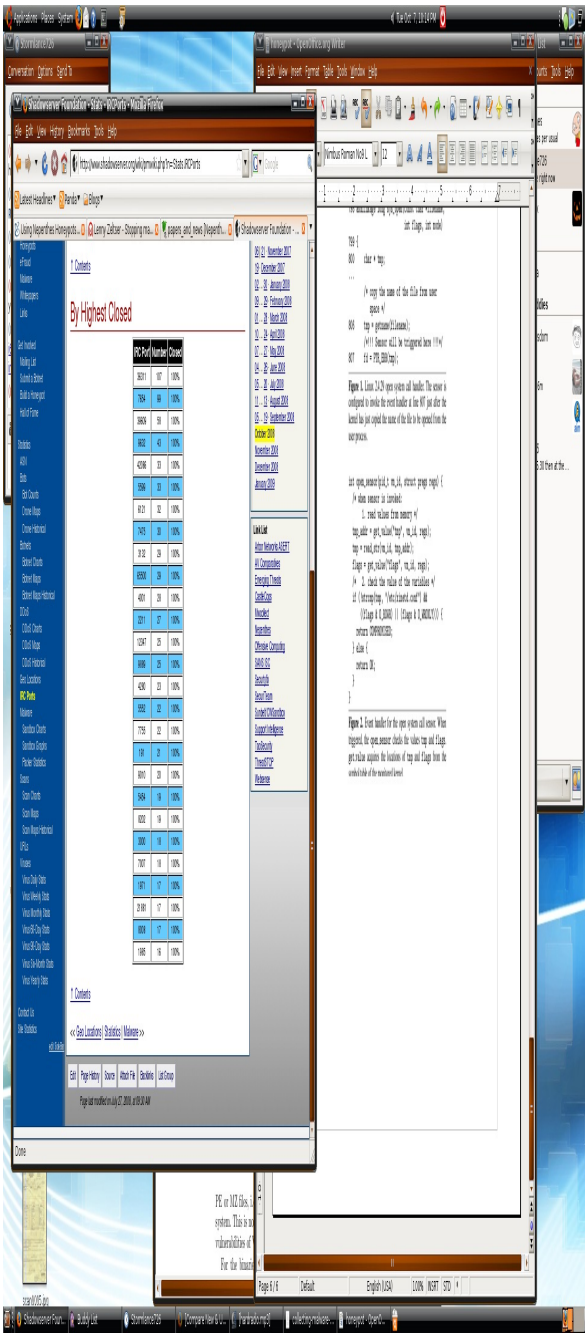
```
798 asmlinkage long sys_open(const char *filename,
                             int flags, int mode)
799 {
800     char * tmp;
...
        /* copy the name of the file from user
           space */
806     tmp = getname(filename);
        /*!!! Sensor will be triggered here !!!*/
807     fd = PTR_ERR(tmp);
```

Figure 1. Linux 2.4.29 open system call handler. The sensor is configured to invoke the event handler at line 807 just after the kernel has just copied the name of the file to be opened from the user process.

```
int open_sensor(pid_t vm_id, struct pregs regs) {
  /* when sensor is invoked:
       1. read values from memory */
  tmp_addr = get_value("tmp", vm_id, regs);
  tmp = read_str(vm_id, tmp_addr);
  flags = get_value("flags", vm_id, regs);
  /*   2. check the value of the variables */
  if (!strcmp(tmp, "/etc/xinetd.conf") &&
      ((flags & O_RDWR) || (flags & O_WRONLY))) {
    return COMPROMISED;
  } else {
    return OK;
  }
}
```

Figure 2. Event handler for the open system call sensor. When triggered, the `open_sensor` checks the values `tmp` and `flags`. `get_value` acquires the locations of `tmp` and `flags` from the symbol table of the monitored kernel

Appendix B: The number of Command and Control servers communicating on selected IRC ports. The IRC Port is the port that the infect machine was listening on, the Number is the number of Command and Control servers communicating over that particular port, and the Closed column represents how many of those Command and Control servers are now inactive.[47]



Appendix C: The process for determining if a packed executable is accessing dirty or clean memory. The point where the flow of execution moves to a dirty regions represents the entry point to a new layer that could be the unpacked portion of the malware.