Implementing Rootkits to Address Operating System Vulnerabilities

Manuel Corregedor and Sebastiaan Von Solms
Academy of Computer Science and Software Engineering, University of Johannesburg
Johannesburg, South Africa
{mrcorregedor, basievs}@uj.ac.za

Abstract—**Statistics show that although malware detection techniques are detecting and preventing malware, they do not guarantee a 100% detection and / or prevention of malware. This is especially the case when it comes to rootkits that can manipulate the operating system such that it can distribute other malware, hide existing malware, steal information, hide itself, disable anti-malware software etc all without the knowledge of the user. This paper will demonstrate the steps required in order to create two rootkits. We will demonstrate that by implementing rootkits or any other type of malware a researcher will be able to better understand the techniques and vulnerabilities used by an attacker. Such information could then be useful when implementing anti-malware techniques.**

Keywords: Rootkits; vulnerabilities; malware; security

I. INTRODUCTION

Anyone who uses a computer for work or recreational purposes has come across one or all of the following problems directly or indirectly (knowingly or not): viruses, worms, trojans, rootkits and botnets. This is especially the case if the computer is connected to the Internet. The current commercial products used by home users to detect and prevent malware are either signature based or heuristic based [1][2][3].

The statistics in [4] show that although malware detection techniques are detecting and preventing malware, they do not guarantee a 100% detection and / or prevention of malware. The reason for not being able to achieve a 100% detection and / or prevention of malware is because malware authors make use of sophisticated hiding techniques in order to prevent malware from being detected by means of signature based techniques. Such techniques are either entirely or partially based on code obfuscation [5][2][6]. This has resulted in the emergence of malware known as polymorphic and metamorphic malware. Metamorphic malware makes use of code obfuscation in order to change its code structure in such a way that very little bytes remain that can be used as a signature [7][8]. Polymorphic malware encrypts its payload using different keys each time to make it undetectable. The polymorphic malware then makes use of a decryptor to decrypt its malicious payload in order to execute it. Polymorphic malware however can be identified by the signature of its decryptor [7][8] which is why it makes use of metamorphic decryptors to avoid being detected. Metamorphic and polymorphic malware poses serious challenges for anti-malware software specifically signature based techniques, however rootkits are a more serious malware threat.

A rootkit is a malicious program or set of programs that tries to hide its existence on an infected computer by attacking the Operating System (OS) by using one or a combination of the following: modifying program binaries, hooking call tables such as the System Service Descriptor Table (SSDT) and the Interrupt Descriptor Table (IDT) to hijack the kernel's control flow, modifying legitimate code to force a call to rootkit code or by using DKOM (Direct Kernel Object Manipulation) [9][10][11][12][13]. Rootkits are designed to fundamentally subvert the OS kernel and are capable of obtaining and maintaining unrestricted control and access within the compromised system without even being detected by anti-malware software [14]. Rootkits can also hide other malicious software or activities such as open network connections, running processes and files on disk [10][11][12]. Long life time rootkits are most likely to attempt to hide [11].

This paper will demonstrate the steps required in order to implement two rootkits. The first rootkit can sabotage a Windows OS by causing it to blue screen or to generate an error after each reboot. The second rootkit can disable anti-malware programs and log the keys pressed by the user. The target OS environment is Windows XP Professional 32 bit and Windows 7 Professional 32-bit, we will use OS to refer to these two environments. The home user is the target of our rootkits. The rationale for this paper is that in order to prevent attacks from rootkits a researcher must first understand how rootkits work and what vulnerabilities they expose. Once this is done counter measures can be identified for the vulnerabilities. To our knowledge no other papers exist that cover how rootkits are implemented. We hope this paper will assist other security researchers in understanding how rootkits work and what vulnerabilities they exploit. Such information could then be useful when implementing anti-rootkit techniques.

The remainder of this paper will be structured as follows: in section 2 we will briefly discuss the tools required to implement rootkits, sections 3, 4 and 5 will look at how to get into the kernel, install the rootkits and manipulate the kernel. Sections 6 and 7 will discuss our Sabotager and Evader rootkits and their functions after which in section 8 we will discuss the vulnerabilities of the OS that our rootkits exploit. Sections 9 and 10 will discuss how the vulnerabilities can be addressed.

II. TOOLS

A list of the tools used to develop the rootkits discussed in this paper can be found in the table 1. It should be noted that all tools are freely available from Microsoft. The rootkits we

TABLE I. DEVELOPMENT TOOLS

Name	Description
Windows Driver Kit	Tools and documentation necessary to develop kernel mode drivers.
Windows SDK	Tools and documentation necessary to develop user mode applications.
Sysinternals Suite	Contains useful tools such as Debug view that allows one to view output from the kernel and other useful tools [16].
Windows Debugging Tools	Contains a set of tools you can use to debug drivers, applications and services.

implemented are both hybrid rootkits because they consist of user mode and kernel mode components. This allows us to have access to all of the kernel's data structures and procedures while still having access to the user mode Windows API. We also make use of a user mode component to communicate with the kernel mode component. The rootkits are implemented as kernel-mode drivers. A kernel-mode driver is a loadable kernel-mode module that is intended to act as middleware between the hardware and the OS's I/O manager [15]. It communicates with the I/O Manager by making use of I/O request packets (IRPs). The IRPs are created by the I/O Manager on behalf of a user-mode application that wants to communicate with the kernel-mode driver. There are three ways to get the rootkits into the kernel [15], we will discuss each in the next section.

III. GETTING INTO THE KERNEL

The first possible way of getting into the kernel is to use the Service Control Manager (SCM).

A. Using The SCM

This is the easiest and most supported way to load a rootkit. The rootkit is registered in the SCM database and as such allows you to use the SCM to manage your rootkit by allowing you to start, stop, restart, load, unload or delete the rootkit. You can also specify when you want your rootkit to be loaded i.e. on demand, auto (load when computer restarts), boot (load by system boot loader), system (load during kernel initialization) or disabled [17]. Using the SCM to load the rootkit is also the most easily detectable by an administrator as it leaves a lot of evidence in the registry as well as showing the rootkit as a running service on the system. A rootkit can take measures to obfuscate this information during runtime however an offline analysis by an administrator will pick it up [15]. However an average home user will in most cases not be aware of such evidence being left behind. In our implementation we do not remove any evidence because we require that certain information remains in the registry in order for one of our rootkits to meet its objective, we will discuss this later. Another possible way to load a rootkit into the kernel is to use an undocumented function.

B. Using the System Call ZwSetSystemInformation

The ZwSetSystemInformation is an undocumented function exported by ntdll.dll. It is possible to import this function and use it to load your rootkit. The advantage is that you can load the rootkit without any traces being visible in the registry. One disadvantage is that the rootkit will be loaded into pageable memory which means the memory manager may write your code onto disk. This will cause a bug check if the rootkit requires the code that has been paged to disk to handle interrupts, acquire spin locks, or hook system calls [15]. Another disadvantage of using this approach is that you are not able to easily manage your rootkit as in the case of using SCM. Lastly this approach does not work in Windows Vista and later [15]. Another possible way to load a rootkit into the kernel is to inject code into the kernel.

C. Injecting Code into the Kernel

Injecting code into the kernel can be done by modifying driver code paged to disk or by leveraging an exploit in the kernel [15]. Leveraging an exploit in the kernel can either be done by finding a vulnerability in the kernel itself or other kernel mode drivers. Both techniques have the advantage that the rootkit can be loaded without any traces being visible in the registry. The disadvantages are that you will not be able to easily manage the rootkit and the vulnerability you relying on could be patched. The next section will discuss what method we chose to install our rootkits.

D. Chosen Method

Our implementation which is targeted at the average home user our rootkits make programmatic use of the Service Control Manager. We also gave the service the misleading description of "SDDL subsystem for Windows USB Resource – Microsoft(C)" this coupled with our driver name msusb.sys would be in our opinion enough to mislead an average home user should they (although very unlikely) discover our rootkit. It is also important to decide where you going to store the actual rootkit file. Although the file can be stored anywhere it should be stored in %windir%\system32\Drivers directory in order reduce possible suspicion as this is the directory where the majority of driver files are stored.

Once you have decided on what method you will use to get your rootkit into the kernel the next step is to decide how your rootkit will be installed and executed.

IV. INSTALLING THE ROOTKIT

We packaged our rootkits as an installer for a popular first person shooter game. The installation displays a fake installer window which remains open while the installation is taking place. The installation consists of using the SCM to load the rootkit, start it and set it up to be loaded by the system boot loader. It also performs other activities which we will discuss later. Once completed we display a message informing the user that the installation failed due to missing files as well as displaying a URL to any site of our choosing.

It should be noted that our chosen method of installation is the easiest as it does not need any specific exploit in order to be installed and relies solely on social engineering. It also relies on the fact that since the target is an average home user he/she will in most cases not be able notice any of the evidence left behind by our installation. This means that the user has to rely solely on the security software he/she is running on their system. This is the first problem to which we will refer to later on in this paper. The next section will describe how we manipulate the kernel to perform what we need.

V. MANIPULATING THE KERNEL

Once in the kernel the rootkit can do any one of the following: hook call tables such as the SSDT and the IDT to hijack the kernel's control flow, modify legitimate code to force a call to rootkit code or use DKOM. The rootkits we will be discussing perform system call hooking and some DKOM. In our implementation we hook the ZwSetValueKey and ZwQuerySystemInformation system routines. We will start by looking at what is needed before we can perform the actual hooking.

A. Laying the Foundation for Hooking

Hooking is performed by firstly importing the routines you want to hook from existing system modules. The import is done during compile time by specifying an exact prototype of the routine you want to import and using the keyword NTSYSAPI to tell the compiler that it should not compile your routine definition but should instead import the routine from an existing system module. We also create routines with different names but with the same routine signatures as the routines we are hooking. We then provide the implementation for the new routines. Now that we are able to get the addresses of the routines we want to hook and the ones we want to replace them with the next step is to perform the actual hooking.

B. Hooking the SSDT

The first step to hooking the SSDT is to disable the write protection enforced on the SSDT. This can be done easily by clearing the 17th bit (WP) of the processor's CR0 register in order to allow the CPU and hence the kernel mode code to write to read only memory. In order to perform the hooking we need the address of the SSDT, this address is exported by ntoskrnl.exe as the symbol KeDescriptorTable. The KeDescriptorTable consists of four entries of which one is the address of the SSDT. We then locate the index in the SSDT that contains the address of the routine that we want to hook and swap the new routine address into the SSDT at that index. The swap is performed by making use of the routine InterlockedExchange which performs an atomic transaction. An atomic transaction is necessary because the SSDT is shared across several processors and hence we need exclusive access to it. Once we swap the entries we re-enable write protection.

We have now demonstrated how easy it is to hook a routine in the SSDT, the next section will describe how we use hooking in one of our rootkits to cause the OS to generate an error or a bug check at every boot.

VI. SABOTAGE

As mentioned our rootkits are setup to be loaded by the system boot loader, this ensures that every time the computer reboots our rootkit will be loaded by the system boot loader. Our Sabotager rootkit, as we have named it, causes the OS to generate an error or a bug check which is more commonly known as a blue screen of death every time the computer reboots. This is accomplished by hooking the ZwSetValueKey routine which is used to replace or create a value entry for a key in the registry[18][15]. Once the routine is hooked we do nothing and we do not call the original ZwSetValueKey and as such the OS cannot set any value in the registry, this causes the OS to generate an error or a bug check during startup. The OS however does have measures in place to deal with newly installed drivers that could be causing problems during boot up. We will not be discussing measures such as re-installing the OS or backing up the OS from a stored image as this is usually a last resort that a user would take to restoring their system because by doing so they would in most cases loose all their current personal data and / or programs.

A. Disabling Boot from Last Known Configuration

The one measure the OS has is that it allows you to boot into the last known configuration by pressing F8 during boot up. The last known configuration boot option makes use of the last control set that was used to successfully boot the OS. A control set contains system configuration information such as device drivers and services that were used to boot the OS. Control sets are stored under the SYSTEM key in the HKEY_LOCAL_MACHINE subtree [19]. There could be several control sets in the registry depending on how often the user makes changes to the services and / or device drivers on their computer.

The OS stores the number of the control set that was last used to successfully boot the OS, this value is stored in the registry value called LastKnownGood in subkey SELECT under the SYSTEM key in the HKEY_LOCAL_MACHINE subtree in the registry. Therefore if there is a problem during boot up and the user chooses to use the last known configuration then the OS will use the number of the control set stored in LastKnownGood to boot up the system.

In order to counter this our rootkit simply overwrites the value stored in LastKnownGood with an arbitrary value for which no control set exists. Writing to the registry is achieved by making use of ZwOpenKey and ZwSetValueKey. This causes the last known configuration boot option to fail.

B. Disabling Safe Mode

The OS also allows the user to boot into safe mode by pressing F8 during boot up. Safe mode allows a user to trouble shoot problems that he/she may be having. Safe mode according to [20] only starts the basic files and drivers necessary to run Windows. However our rootkit still loaded because we set it up to be loaded by the Windows boot loader. The Windows boot loader loads all the drivers that are classified as belonging to the boot class category in the registry [15]. This is the reason why we left evidence of our rootkit installation in the registry i.e. in order for the boot loader to load our rootkit. The important thing to take note of is that our rootkit is loaded before the Session Manager. The Session Manager is responsible for starting all the processes necessary for creating a login session for the user [15]. As such we can conclude that the login session cannot be created because we are blocking it from setting any values in the registry, which is something that it requires in order to create the login session. The other load options namely demand, auto and system do not work in terms of meeting our needs.

It should be noted that by disabling the use of safe mode and booting from last known configuration we have prevented Windows XP 32 bit from being able to recover from our Sabotager rootkit. However Windows 7 Professional 32 bit can still make use of the Startup Repair feature and System

Restore. We will discuss the Windows 7 Startup Repair feature next.

C. Windows 7 Startup Repair

The startup repair feature was first introduced in Windows Vista. This feature attempts to automatically diagnose and repair problems that prevent Windows from booting [21]. It also provides access to other features such as System Restore, System Image Recovery, Windows Memory Diagnostic Tool, Command Prompt and lastly to re-install Windows [21]. Windows 7 32 bit failed to automatically repair Windows in the case of our Sabotager Rootkit. The last option that could be used is System Restore.

D. Disabling System Restore

The last obstacle in the way of our Sabotager rootkit is Windows System Restore. System restore automatically tracks changes to a computer at all times and at specific intervals by creating restore points that contain the state of a computer before changes occur. Restore points can also be manually created by a user. This allows the user to restore the computer to a previous state, by choosing a restore point on a date and time prior to when a change was made that may be causing problems [22]. This would mean that the OS could restore the computer back to a point before the installation of our rootkit. In order to counter this, the user mode component of our rootkit deletes all previous restore points on the computer and creates a new restore point after it installs the rootkit. This is all achieved by making use of the routines provided by SrClient.dll. The next section will look at the error messages that the user will receive when the computer fails to boot.

E. Error Messages

In initial testing a bug check was generated also known as a blue screen of death for both Windows XP Professional 32 bit and Windows 7 32 bit however recently with all updates installed up until the 9th of April 2011 Windows XP Professional 32 bit now when doing a normal boot displays a message stating that lsass.exe had failed with the error message "The requested Operation was unsuccessful". The process lsass.exe is the Local Security Author Subsystem created by the Session Manager which plays an important role in performing user authentication during a login [15]. The error message is misleading because there is no problem with lsass.exe. The following were the bug checks generated during testing with all updates installed up until the 9th of April 2011:

Windows XP Professional 32 bit and Windows 7 Professional 32 bit when booting into safe mode both generated a bug check with the title BAD_POOL_CALLER with stop code 0x000000C2 with the first parameter 0x00000007. This indicates that a driver has attempted to free memory which was already freed [23]. The only way to find out which driver is responsible would be to attach a kernel debugger such as Kd.exe to the kernel or to the crash dump file created during the bug check.

Windows 7 Professional 32 bit when doing a normal boot generated a bug check with the title "The video driver failed to initialize" with stop code 0x000000B4. This indicates that the problem is a conflict with the parallel port and the video card [24]. The advice in [24] is to restart in safe mode and change the address of the parallel port to 0378, this however would not resolve the problem. The next section will look at the practical uses of our Sabotager rootkit from an attacker's perspective.

F. Practical Uses

To our knowledge this is the only type of malware which aims specifically to make a user's computer unusable by causing the OS not to boot in this way. Although this at first might seem like it could have very little value from an attacker's perspective let's consider how an attacker could use such an electronic weapon to make money.

The first example we can look at is one where the user takes a computer in for a service at a store for example a hardware upgrade, software installation etc. The technician at the store performs the service but also sets up an installer to install our rootkit in say one month. Within one month the user would start to run into all the problems that we discussed and it would be very likely that the user would bring the computer back to the store where they took it for a service. The technician at the store could then charge the consumer for anything for example a video card, a mother board, memory modules, data recovery etc. The same could be performed by a store that sells computers with pre-installed software on the computer. Another use of our rootkit could be that once the user installs our rootkit we display and print a message informing the user that their computer has been infected and making it seem as if the message has been created by the OS. The message will instruct the user to download a paid for "fix" from some site and warn the user that if they don't the OS will become unusable.

This concludes the discussion of our Sabotager rootkit, the next section will discuss our second rootkit which disables anti-malware programs and logs the keys pressed by the user.

VII. Disabling Anti-malware Programs and Logging Keys

Our Evader rootkit, as we have named it, logs the keys pressed by the user and stores them in a text file, it also disables anti-malware programs running on the computer. This rootkit is setup to be loaded by the OS kernel during initialization. We will discuss each of these features in turn starting with the logging of keys pressed.

A. Logging Keys Pressed

Our Evader rootkit is implemented as a filter driver. Filtering is made possible through Microsoft's driver model that supports a layered architecture such that it allows several drivers to work together in a chain in order to achieve a certain goal. This architecture allows a driver to be injected into the driver stack and to leverage functionality that has already been implemented [15]. The filter driver can manipulate an IRP before passing it on to one of its adjacent drivers [15]. Our rootkit filters the PS/2 Keyboard in order to intercept all the key strokes. The rootkit works as follows:

A request for input by the Windows subsystem is sent to the keyboard driver prior to any key being pressed, when this request occurs an IRP is automatically created by the I/O manager and sent down to the keyboard driver to await input

from the keyboard [15]. Our rootkit registers a dispatch routine for this IRP, this allows us to receive the IRP as it goes down the driver stack. When the IRP passes through our dispatch routine, on the way to the keyboard driver, we register a completion routine for it such that when it completes i.e. gets the keyboard data we will get to access the keyboard data contained in it.

The one challenge faced in this implementation is that when handling IRPs specifically the completion of IRPs the completion routine code usually executes at the Interrupt Request Level (IRQL) of DISPATCH_LEVEL and in order to do things like write files to disk, which we require to do, the IRQL must be PASSIVE_LEVEL[15]. User mode programs and threads run at PASSIVE_LEVEL [25]. In order to overcome this as shown in [15] we can allocate a buffer in memory that can be used to store the keyboard data captured and we create a thread that executes at PASSIVE_LEVEL. The completion routine executing at DISPATCH_LEVEL stores the keys being logged into the buffer. The thread that executes at PASSIVE_LEVEL reads the contents of the buffer and writes them to disk. Access to the buffer is synchronised by making use of a mutex. It should be noted that all keys pressed will be logged regardless of what security measures a user has taken for example using HTTPS on a website.

This concludes our discussion on the key logging function of our rootkit, the next section will discuss how we disable anti-malware programs.

B. *Disabling Anti-malware Programs*

The first step to disable anti-malware programs is to hook the ZwQuerySystemInformation routine as described earlier, We then call the original ZwQuerySystemInformation routine in order to filter the results of that call. We then check the type of information that we are receiving in order to ensure that it is process information that we are receiving. If it is we retrieve the system process information structure. This structure however has been obfuscated and there is no official documentation detailing all of its attributes, specifically the ones that we are interested in such as the process name and process ID. However by looking at sources such as [15] we can get the undocumented versions of such structures. It should be noted that such undocumented structure definitions could differ across different versions of Windows [15].

We then iterate through all the system process information structures, where each structure represents a current executing process on the computer. We then compare each process name against a list of anti-malware program process names that we maintain. We terminate the process if it has a name that matches an anti-malware process name in our list. Once again due to the fact that we are executing at an IRQL of DISPATCH_LEVEL we are not allowed to terminate the process. We overcome this limitation once again by sharing a memory location between the code executing at DISPATCH_LEVEL and a thread executing at PASSIVE_LEVEL. The process ID is stored in this memory location by the code executing at DISPATCH_LEVEL. The thread executing at PASSIVE_LEVEL retrieves the process ID and terminates the process.

In our testing, we have successfully terminated the following anti-malware programs: Avira AntiVir Personal (10.0.0.611), AVG Internet Security 2011 (10.0.1204), BitDefender Total Security 2011 (14.0.28.351), F-Secure Internet Security 2011(10.51 build 106), avast! Free Antivirus (6.0.1000) and Microsoft Security Essentials (2.0.657.0). All anti-malware programs were executing on 32 bit Windows XP Professional with all updates installed up until the 11th of March 2011. Avira was the only anti-malware program that detected our Evader Rootkit by making use of a generic detection routine. However when we changed the extension of our rootkit from .sys to something else Avira did not automatically detect our rootkit. This allowed us to install and execute the rootkit. The next section will look at the practical uses of our Evader rootkit from an attacker's perspective.

C. *Practical Uses*

Our Evader rootkit could allow an attacker to disable the anti-malware program installed on a user's computer and install a rogue anti-malware program that looks just like the original anti-malware program that just got disabled. It could then deploy any other malware and take over the computer, the user would be none the wiser as he/she would still think they were protected. Regardless of whether or not the anti-malware program is disabled, the rootkit will still be able to collect user information through the key logger. This concludes the discussion of our rootkits. The next section will discuss the vulnerabilities that we have identified by looking at how our rootkits have been implemented.

VIII. VULNERABILITIES

Through the implementation of our rootkits we have identified the following categories of vulnerabilities the kernel, sharing memory, the registry, the Windows boot loader, system messages and the user. We will discuss each in turn.

A. *The Kernel*

The first vulnerability is the fact that we can easily clear the 17th bit (WP) of the processor's CR0 register allowing the CPU and hence our rootkits that run in kernel mode to write to linear read only memory [26][15]. The WP flag's purpose is to protect read only user-mode memory from being modified by kernel-mode threads. It should also be noted that user mode threads cannot modify read only memory regardless of whether or not the WP flag is set [26]. The WP flag is also used for implementing the copy-on-write strategy used by operating systems such as Linux [26][27].

The other vulnerability is that we can easily hook the routines in the SSDT once write protection is disabled. Lastly all kernel-mode code is considered to be trusted by the OS kernel allowing our code to do whatever the OS can do since they execute at the same privilege level [15][28].

B. *Sharing Memory*

The vulnerability in this section deals with the sharing of a memory buffer between two threads, where one thread is executing at PASSIVE_LEVEL and the other at DISPATCH_LEVEL. Specifically the allowance of a thread with a lower privilege level to read memory locations written to by a higher privilege level thread. This vulnerability is what allows us to log our keys to disk and to terminate the anti-malware programs.

C. The Registry

In kernel mode it is very easy to modify the registry be it adding new keys or modifying existing keys. It should be noted that the registry is Window's central hierarchical database that it uses to store information that it uses continually during operation such as the profiles for each user, the applications installed on the computer and the ports that are being used [29]. Our rootkit simply blocked write access to the registry which is what caused the OS to not startup. We also used the registry to disable the last know configuration boot up option. It should be noted that our rootkit could do several other things such as disabling user accounts, set up programs to automatically start, change several security settings etc. Several malware uses the registry as an attack vector for example the recent Win32/Afcore family of trojans also known as Coreflood makes use of the registry in order to allow itself to execute when Windows Explorer runs and when Internet Explorer is launched [30]. Other examples of such malware are Vundo [31] and PWS:Win32/Zbot [32].

D. Windows Boot Loader

As discussed our Sabotager rootkit was setup to be loaded by the Windows boot loader by doing so our rootkit will be loaded even if booting into safe mode. The vulnerability in this case is more relevant to Windows XP 32 bit because once safe mode is disabled the user will have no "built in" features to assist him/her.

E. System Messages

This section looks at the system messages displayed and not displayed during the execution of our rootkits. The first thing worth noticing is that we were able to delete all the restore points on the computer without a single warning or confirmation message being displayed to the user. We also noticed that after installing updates on Windows XP 32 bit Professional that the kernel was outputting debugging information which we saw using debug view (dbgview.exe), the message displayed was: "A driver is mapping physical memory 0001F000->0001FFFF that it does not own. This can cause internal CPU corruption. A checked build will stop in the kernel debugger so this problem can be fully debugged."

This message is the result of our rootkit hooking ZwSetValueKey. The user should be made aware of such a problem by means of a pop up warning the user that the program just installed could be performing malicious activities. This could then help the user in establishing what caused the problem.

Lastly all the bug checks generated and messages displayed as mentioned are misleading and do not help the user to resolve the problem. Furthermore some of the solutions proposed to resolve the bug checks would be very difficult for an average home user to perform.

F. The User

This section addresses the last category of vulnerabilities namely the user of the computer. Firstly the installation of our rootkit is only possible if the user executes our installer which is why we used social engineering to trick the user into thinking he/she is executing a file that will install a well known game. It should be noted that we could easily make it seem like the installer is for some other type of software. Secondly the user does not posses adequate knowledge or experience to determine which drivers/services should or should not be running on a computer. This is the reason why we do not attempt to hide the evidence of our rootkits existence.

The last vulnerability is that the average home user relies entirely on their anti-malware software protecting them and as such is given a false sense of security. This false sense of security contributes to a user executing our installer because their anti-malware program indicated that our installer and rootkits were not malicious. In addition to our tests where we disabled the anti-malware programs we also scanned our rootkits using www.virustotal.com on the 17th of April 2011. A total of 42 anti-malware programs were used namely AhnLab-V3, AntiVir, Antiy-AVL, Avast, Avast5, AVG, BitDefender, CAT-QuickHeal, ClamAV, Commtouch, Comodo, DrWeb, Emsisoft, eSafe, eTrust-Vet, F-Prot, F-Secure, Fortinet, Gdata, Ikarus, Jiangmin, K7AntiVirus, Kaspersky, McAfee, McAfee-GW-Edition, Microsoft, NOD32, Norman, nProtect, Panda, PCTools, Prevx 3.0, Rising, Sophos, SUPERAntiSpyware, Symantec, TheHacker, TrendMicro, TrendMicro-HouseCall, VIPRE, ViRobot and VirusBuster.

All anti-malware programs failed to detect our Sabotager rootkit. Only Avira as in our tests was able to detect our Evader rootkit, all the other anti-malware programs were unable to do so. This serves to demonstrate the dangers of a false negative. This concludes our discussion regarding the vulnerabilities we have identified, the next section will look at how some of these vulnerabilities have been addressed in versions of Windows Vista 64 bit and later.

IX. Vulnerabilities Addressed in 64 bit Windows

As mentioned at the start of our paper our rootkits have been written for 32 bit versions of Windows. Some of the vulnerabilities identified in the previous section have been addressed in versions of Windows Vista 64 bit and later. The 64 bit version of Windows Vista and later differs from its 32 bit counterparts as follows [33]:

- 64-bit versions require that all device drivers be digitally signed by the developer.
- 64-bit versions of Windows provide Kernel Patch Protection also known as PatchGuard which helps prevent a malicious program from doing the following [34]: Modifying system service tables such as the SSDT, allocating memory and using it as the kernel and patching any part of the kernel (AMD64 only).

Although the changes greatly improve the security of the 64 bit versions of the OS, as we will discuss in the next section, it is still possible to bypass these security improvements.

A. Bypassing Driver Signing and Kernel Patch Protection

To get around the fact that drivers must be signed what rootkit developers could do, if they had the money, is start a front company and purchase a signing certificate (software publishing certificate) and then use it to sign their rootkits. An alternative method would be to find a vulnerability in a signed driver and exploit that vulnerability in order to load their unsigned rootkits [15].

The first problem with PatchGuard is that it still executes in kernel-mode where the rootkit executes. Although Microsoft have used anti-debugging, anti-detection and obfuscation techniques to make it very difficult for anyone to locate and disable PatchGuard, it is still possible for a rootkit to manipulate PatchGuard. There are technical papers [35][36] [37] that have already demonstrated how to disable or co-exist with PatchGuard, although these are for older versions of PatchGuard it serves to demonstrate that it is possible to disable PatchGuard. It should be noted that when anti-malware programs and malware run with the same privilege level trying to counteract each other, that it is in most case a losing battle for anti-malware programs as new vulnerabilities always emerge that the malware can exploit [12][13][38]. The next section will look at the architecture that we believe future anti-malware programs should be based on in order to deal with the vulnerabilities we have identified.

X. Addressing the Vulnerabilities

In figure 1 we show the current architecture employed by most commercial anti-malware products. The important thing to take note of is the fact that components of the anti-malware program run in both user mode and kernel mode. Anti-malware drivers that run with the same privilege level as a malware's kernel mode driver can be compromised. We propose that all future malware architectures should be based on the architecture presented in figure 2 below. This architecture is very similar to current ongoing research in [11] [13][10][38] that propose the use of hypervisors and virtualization techniques to deal with rootkits. Such an architecture would significantly reduce the chance that malware would be able to disable or compromise the anti-malware program because the anti-malware program would be, at all times, executing at a higher privilege level than the malware. We also propose using an authorisation mechanism, indicated on figure 2 by a lock icon, to prevent PASSIVE_LEVEL threads from reading data from a memory location that has been written to by a DISPATCH_LEVEL thread.

Lastly anti-malware program developers need to start taking some responsibility in assisting and educating the user when it comes to malware threats including social engineering attacks. This is indicated in figure 2 as the Anti-malware Program Advisor component. This component should provide the user with up to date information regarding social engineering attacks as well as a database where a user could refer to for help when in a situation where they are unsure of what to do. Additionally it should provide users with interactive, clear and easy to understand messages when a threat is detected in order to better assist the user. Such a component could significantly reduce the chances of preventing the malware from being installed.

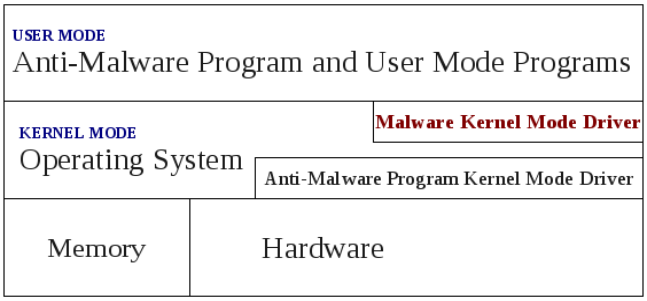

Figure 1. Current anti-malware architecture

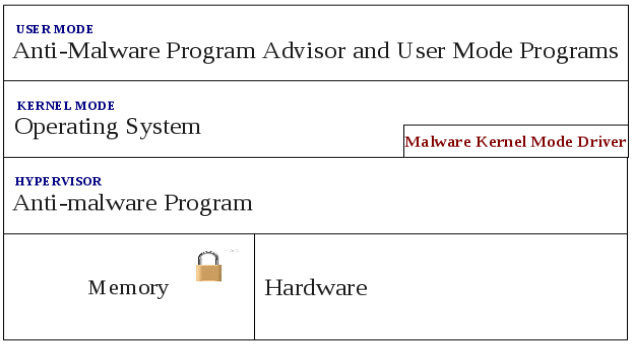

Figure 2. Future anti-malware architecture

XI. Conclusion and Future Work

This paper has demonstrated the steps taken in order to implement two rootkits that could collectively disable anti-malware programs, log keys and prevent the OS from booting. The steps demonstrated what vulnerabilities the rootkits exploited namely the kernel, sharing memory, the registry, the Windows boot loader, system messages and the user. The measures currently used to address the identified vulnerabilities were discussed and weaknesses identified. An architectural model based on current ongoing research was proposed for future anti-malware programs.

We have shown that current anti-malware program architectures have several weaknesses which can be exploited by rootkits when executing at the same privilege level as the OS. This paper has also demonstrated that by developing a rootkit or any other type of malware a researcher will be able to better understand how they work and therefore will be able to counter the techniques used by them and by doing so prevent the malware from causing any damage.

The use of hypervisors to prevent malware is nothing new, however we believe that more focus should be given to assisting the user in making the correct decisions, this is especially true for a home users who do not have policies in place to assist them. Future work will consist of implementing the current proposed architecture, extending the architecture as we identify vulnerabilities, adding to our rootkits and developing additional malware in order to better understand how they work.

REFERENCES

[1] Yin, H., Song, D.: Panorama: capturing system-wide information flow for malware detection and analysis. In Proceedings of the 14th ACM conference on Computer and communications security, pp.116--127. ACM, New York, October 2007.

[2] Zhou, Y., Inge, M.: Malware detection using adaptive data compression. In Proceedings of the 1st ACM workshop on Workshop on AISec, pp. 53--60. ACM, New York, 2008.

[3] Zolkipli, M.F., Jantan, A.: A Framework for Malware Detection Using Combination Technique and Signature Generation. In Computer Research and Development, 2010 Second International Conference, pp. 196--199. IEEE, New York, June 2010.

[4] Volume 9 of the Microsoft Security Intelligence Report (SIR). http://www.microsoft.com/security/sir/default.aspx, 12 November 2010.

[5] Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In ACM Transactions on Programming Languages and Systems, ACM, New York, August 2008.

[6] Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In Proceedings of the 1st India software engineering conference, pp. 4--14. ACM, New York, 2008.

[7] Campo-Giralte, L., Jimenez-Peris, R., Patino-Martinez, M.: PolyVaccine: Protecting Web Servers against Zero-Day, Polymorphic and Metamorphic Exploits. In 28th IEEE International Symposium on Reliable Distributed Systems, 2009. SRDS '09, pp. 91--99. IEEE, New York, September 2009.

[8] Leder, F., Steinbock, B., Martini, P.: Classification and detection of metamorphic malware using value set analysis. In 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), pp. 39--46. IEEE, New York, October 2009.

[9] Sparks, S., Embleton, S., Zou, C.: A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS. In Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, pp. 125 -- 134. ACM, New York, 2009.

[10] Riley, R., Jiang, X., Xu, D.: Multi-Aspect Profiling of Kernel Rootkit Behavior. In Proceedings of the 4th ACM European conference on Computer systems, pp. 47--60. ACM, New York, April 2009.

[11] Jones, S., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: VMM-based Hidden ProcessDetection and Identification using Lycosid. In Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 91--100. ACM, New York, March 2008.

[12] Vasisht, V., Lee, H.: SHARK: Architectural Support for Autonomic Protection Against Stealth by Rootkit Exploits. In 41st IEEE/ACM International Symposium on Microarchitecture, pp. 106--116. IEEE Computer Society, Washington DC, 2008.

[13] S. Josse.: Rootkit detection from outside the Matrix. In Journal in Computer Virology Volume 3, Issue 2, pp. 113--123. Springer, 30 June 2007.

[14] Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating Return-Oriented Rootkits With "Return-less" Kernels. In Proceedings of the 5th European conference on Computer systems, pp. 195 -- 208. ACM, New York, 2010.

[15] Blunden, B.: The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Wordware Publishing, Inc., Texas, 2009.

[16] Windows Sysinternals. http://technet.microsoft.com/en us/sysinternals/default, 15 February 2011.

[17] SC. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/sc.mspx?mfr=true, 11 April 2011.

[18] ZwSetValueKey. http://www.osronline.com/ddkx/kmarch/k111_5yya.htm, 9 April 2011.

[19] What are Control Sets? What is CurrentControlSet? http://support.microsoft.com/kb/100010, 9 April 2011.

[20] What is safe mode? http://windows.microsoft.com/en-US/windows-vista/What-is-safe-mode, 10 April 2011.

[21] Startup, http://windows.microsoft.com/en-US/windows7/products/features/startup-repair, 10 April 2011.

[22] Understanding System Restore. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/app_system_restore_hss_understand.mspx?mfr=true, 9 April 2011.

[23] How to Debug "Stop 0xC2" or "Stop 0x000000C2" Error Messages. http://support.microsoft.com/kb/265879, 11 April 2011.

[24] Err Msg: Stop 0x000000B4 The Video Driver Failed to Initialize. http://support.microsoft.com/kb/240369, 11 April 2011.

[25] What is IRQL? http://blogs.msdn.com/b/doronh/archive/2010/02/02/what-is-irql.aspx, 11 April 2011.

[26] Intel 64 and IA-32 Architectures Software Developer's Manual. April 2011.

[27] Hailperin, M. Operating Systems and Middleware: Supporting Controlled Interaction. Thomson Course Technology, 2007.

[28] TanenBaum, A., Woodhull, A.: Operating Systems Design and Implementation., Chapter 1. , New Jersey, 2006.

[29] Windows registry information for advanced users. http://support.microsoft.com/kb/256986, 14 April 2011.

[30] Wong, J., Williams, J.: MSRT April '11: Win32/Afcore. http://blogs.technet.com/b/mmpc/archive/2011/04/13/msrt-april-11-win32-afcore.aspx, 13 April 2011.

[31] Wong, J.: Vundo Employs Worm Behavior. http://blogs.technet.com/b/mmpc/archive/2009/04/22/vundo-employs-worm-behavior.aspx, 22 April 2009.

[32] Sanico, J.: Got Zbot? http://blogs.technet.com/b/mmpc/archive/2010/03/11/got-zbot.aspx, 11 March 2010.

[33] A description of the differences between 32-bit versions of Windows Vista and 64-bit versions of Windows Vista. http://support.microsoft.com/kb/946765, 5 April 2011.

[34] Patching Policy for x64-Based Systems. http://msdn.microsoft.com/en-us/windows/hardware/gg487350.aspx, 5 April 2011.

[35] Skywing: Bypassing PatchGuard on Windows x64. http://www.uninformed.org/?v=3&a=3&t=pdf, December 2005.

[36] Skywing: Subverting PatchGuard Version 2. http://www.uninformed.org/?v=6&a=1&t=pdf, December 2006.

[37] Skywing: PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3. http://www.uninformed.org/?v=8&a=5&t=pdf%20, September 2007.

[38] Chubachi, Y., Shinagawa, T., Kato, K.: Hyperviosr-based Prevention of Persistent Rootkits. In Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 214 -- 220. ACM, New York, June 2010.