WILLIAM & MARY
CHARTERED 1693

Colonial Academic Alliance Undergraduate Research Journal

Volume 1

Article 5

2010

A Hybrid Root-kit for Linux Operating System

Subrata Acharya Dr.
Towson University, sacharya@towson.edu

Brian Namovicz
Towson University, bnamov1@students.towson.edu

Jonathan Wiseman
Towson University, jwisem2@students.towson.edu

1. Introduction

Hacking has been around almost since the first computers were connected together. Every day many new vulnerabilities/exploits are released and many computers become compromised. This is good for an attacker because there is a constant stream of new vulnerabilities/exploits that can be leveraged to break into computers. However, with newly published exploits comes a newly released patch for those exploits (usually). This is the reason that attackers have developed '*back-doors*' commonly referred to as root-kits.

A root-kit is a post-compromise tool that an attacker uses to maintain access and often collects information from users such as passwords, credit card information, social security numbers, and other sensitive information. The importance of a root-kit is that once the vulnerability which was used to exploit the system is patched, the attacker can still get back in through a '*back-door*'. The purpose of this paper was to explore the area of root-kits by taking the role of an attacker and actually developing a root-kit that targets the *Linux 2.6 kernel*. By doing this we were are able to gain a great amount of insight into the internal workings of the kernel as well as its shortcomings with regards to security by developing a *Linux Kernel Module (LKM) key-logger*. We also look into some common techniques used by root-kits for providing a back-door to the attacker. Then we investigate some simple techniques that root-kits utilize for stealth (it is imperative that the users/administrators do not know the system is compromised). Finally, we look at a simple and elegant solution to infecting a compromised computer with the root-kit we developed.

2. Hooking System Calls in Linux

One of the most common methods that root-kits have used over the years to subvert systems is a technique known as hooks. Hooks work by modifying function pointers to point to a malicious version of a function. By doing this an attacker can gain complete control of the execution flow of a particular call. Commonly targeted functions are system calls because any user-mode program that does anything of interest (file access, memory allocation/management, read/displaying data) must use system calls to communicate its intentions to the kernel.

System calls are referred to by a table in memory called the system call table. This table contains pointers to the functions that the user-mode program is trying to use. For example a user may want to open a file using the *fopen()* function in the C library. When the user calls *fopen()*, the C library processes this call and passes the information to the operating specific implementation of the *open()* system call. The *open()* is the user-mode interface for the kernel mode system call *sys_open()*. *Open()* does some checks on the arguments to make sure the user is allowed to open the file and then calls *sys_open()*. The call to *sys_open()* causes the system to lookup the address of the *sys_open()* function in the system call table and switch to kernel mode briefly while the function completes and returns a file descriptor. The return value is passed up the function call stack and eventually the user-mode program is given a handle to the file they opened with the *fopen()* function.

A system call hook would intercept this request by changing the function pointer in the system call table to point to a malicious version of *sys_open()* instead of the real one. This function could simple collect the data passed and then continue to pass the arguments to the real *sys_open()* or could completely rewrite the way that *sys_open()* opens a file, the possibilities are nearly endless. This entire process can be done easily with a *Linux kernel module (LKM)*. This technique has been known for a long time and is a favorite of hackers. Due to the widespread abuse of system call hooks in the *2.4 Linux kernel*, the kernel developers decided that the *2.6 kernel* was not going to allow *LKMs* to have access to the system call table. To prevent system call hooks, the *2.6 kernels* no longer export the *sys_table* symbol. This means that in the *2.4 kernel* a *LKM* could refer to the *sys_call_table[]* array and directly manipulate the functions it pointed to, now in the *2.6 kernel* the *sys_call_table[]* array has no meaning.

However, this only makes things difficult, not impossible. There have been a number of ways that people have successfully gained access to the system call table. This is usually accomplished by reviewing the kernel source and finding what other globally exported functions are near the system call table in memory and using search techniques to find the array. The problem with this is that these functions that are supposed to be near the system call table are not near it in all versions of the *2.6 kernels* so this is not a very elegant solution.

Our next thought was, "*the kernel has to know where the system call table is stored somehow, if we can find out how it knows, maybe then we can find it too*". After a little bit of research we found that there is in fact one reference to the system call table in the operating system. In the file */boot/System.map* there is data which the kernel uses on boot-up to know where everything is supposed to be at in memory, this includes a reference to *sys_call_table*. This means that if we can get a handle to the system call table by grabbing this address from the */boot/System.map* file and putting it into our *LKM* source code at compile time. This handle is simply an unsigned long pointer that is set to the address from the file. This pointer can then be referenced using array notation just as the *sys_call_table[]* was used in the *2.4 kernel*. This method is much more reliable, compatible, and elegant than the searching methods that others have previously used.

Now that we have a method to hook system calls, how can we use this to log keystrokes? Well everything in the Linux operating system is in some way represented as a file. This includes the standard input/output/error *(stdin/stdout/stderr)*. Processes interacting with this files use the same standard file descriptors, *stdin* = 0, *stdout* = 1, *stderr* = 2. Knowing this, we could make a system call hook for calls to *sys_read()*. Pseudo code for the evil function:

```
evil_sys_read(parameters)
    if (file descriptor == 0)
        log the buffer parameter
return real_sys_read(parameters)
```

This method was implemented and it worked like a charm. The same method could even be used to see *stdout/stderr* also and log both sides of the interaction. The problem was that it was not able to capture passwords, which is one of the main goals of a *key-logger*. By looking into source code for the *getpass()* function in the C library it seems that extra care is taken when passing password data from a *TTY* device to the process. It appears that data is read directly from the input queue of the *TTY* into the process. This seems to bypass the *sys_read()* function that we were intercepting. It appears that our key logging solution would have to exploit something at a lower level.

This effort was not the solution we were looking for; however it was not a complete loss either. Most sources that we found on the Internet proclaim that it is impossible to reliably hook system calls in the *2.6 Linux kernels* via the system *call table*. These rumors were certainly disproved by our experiments.

3. Linux Pseudo-terminals

With system call hooks not providing us with the data we wanted, it was time to dig deeper and try to attack the *TTY* devices themselves. On most *Linux* distributions a number of different type of terminal devices are available. The most common and widely used is the *pseudo-terminal* devices. These devices allow a configurable number of terminals to be created as they are needed by different processes. This is a distinct advantage over previous terminal implementations, all of which had a set number of terminals available (usually

64), which were always on and waiting to be connected to a process. The way that the *pseudo-terminal* system works is by separating duties to a single *pseudo-terminal master (ptmx)* and a configurable number of dynamically created *pseudo-terminal slaves (pts)*. Each of these *pts* devices are named after their *index* and placed in the *pts* directory of the *dev file system*. For example, the first *pts* would have the *index* 0 and the device node would be */dev/pts/0*.

These file nodes, like all files can among other things be *opened*, *read*, and *written* to by either their owner or the *root* user. For example, a *root* user can use a command such as:

```
echo "hello world" > /dev/pts/0
```

The result is the words '*hello world*' is printed onto the terminal that uses */dev/pts/0*. Note that the text is not put into the input queue of the terminal, meaning that the text is only displayed and cannot be used to run commands as a user in another terminal. We need to know how this entire system works in order to potentially exploit it with a function hook.

The first question is how are the pts devices created dynamically? Well there is a function in the C library which a process is supposed to use to allocate a terminal for itself *(getpt() function)*. This function works by calling *open()* on */dev/ptmx*. The open function for the *ptmx* device is specially written to do the proper things involved with allocating a new *pts*, once it does this, it returns a file descriptor to the *pts* that it created for the calling process.

So now we know that each time a user remotes into the target computer, the *SSH* process forks off and drops privileges to the correct user, and then probably uses the *getpt()* function which opens */dev/ptmx* which returns a file descriptor to a *pts* (*/dev/pts/X*). This *pts* is connected to the bash shell that the user is likely using. Good, so now we understand enough about how the *pseudo-terminal* system works to give processes access to *pts*.

4. Final Pseudo-terminal Key-Logger

Now that we know how to use hooks to intercept system calls and we know how the pseudo-terminals work, we can develop a decent method to log keystrokes. The first challenge is how we set hooks on the pseudo-terminals because they are created dynamically. Well we know that the pseudo-terminal master (*ptmx*) device is responsible for creating *pseudo-terminal slaves (pts)*. So if we hook the open call for */dev/ptmx*, we can find when a new pts is being created and then set the appropriate hooks on the pseudo-terminal slaves.

The next challenge is how do we set hooks on the *pts* device? The *pts* devices are just a type of *TTY* device that we can find code for in the *Linux kernel* source. After reviewing the code and finding how everything works, we found that there are two possibly useful functions that we would like to create hooks for. In the *tty_struct* structure, there is a *tty_operations* structure which points to the driver specific functions that would be mapped to the file node's file operations structure (defines what the file does when the user attempts to *open*, *read*, *write*, etc). So it would be great if we could just hook the functions for *reading*/*writing* (*tty_read*, *tty_write*) to the terminal right? There is a problem. One of the other security changes between the *2.4 and 2.6 Linux kernel* was to make the *tty_operations* structure constant, presumably to thwart simple key-logging hooks like this. However, previously we discussed a way to create hooks despite the kernel developer's best efforts to stop us. A similar technique can be implemented to bypass the constant declaration of the structure.

In the previously mentioned hook technique, the problem we had was getting a handle to the correct location in memory that stored the function pointers; in this case we already have it. The problem is we cannot directly write to that location due to the structure being constant. But let's think about this, we have kernel level access; at kernel level almost anything is possible right? What if instead of manipulating the pointer in the context of the structure, we manipulate the pointer in the context of a memory address? It turns out this works really well, regardless of the structure being constant. It is impossible to stop us from overwriting the bytes that define the *tty_read*/*tty_write* function pointers in the structure. To do this we create a regular pointer and set it to the address of the *tty_operations* structure and by reading the source code, which is freely available, we can determine the offset of the correct pointers in the structure. These pointers can now be addressed using simple array/index notation rewritten to the addresses of our malicious functions. The hooks are in place.

The final task was to find which function (*tty_read* or *tty_write*) was going to give us the data we wanted (most importantly passwords). As indicated in the Phrack article on writing a LKM Key-logger, the *tty_read* function is what we wanted, so this was the first function we tried. Our technique for setting hooks for this function worked beautifully, data was being captured and we event wrote the code to write the captured text to specific files for each different *pts* device. There was one problem, no passwords.

As previously mentioned, the problem with the system call hooks was that they couldn't collect masked passwords. But this time we are seeing everything that goes through the *TTY* right? Not quite, because of the nature of *TTYs* they work in full-duplex mode. When you type a character and you see it on the screen, in the background the process that is displaying the characters is reading the data from the *TTY*, this is called echoing. In the case of masked passwords, no data is being read from the TTY, echoing is turned off. So it turns out that the data we are looking for isn't available via *tty_read* because the data is never read.

So we had to deviate from what the source told us should work and try to hook the *tty_write* function. This was very straightforward, the code was nearly identical to the *tty_read* hook. Once this code was implemented, like magic, we were collecting masked passwords. Now anytime a user used a program like *su* (switch user) or *passwd* (changes their password) we would log the inputted password. We finally had a working LKM key-logger that theoretically works on all of the *2.6 Linux kernels*. To our knowledge this has not previously been done. Although we focused on the pseudo-terminal *TTYs*, this technique attacks the *TTY* infrastructure as a whole, not just pseudo-terminals. It could easily be ported to work with other types of *TTY* devices. A particular interest may be the *TTYs* used for local access, as these often are not pseudo-terminals and are likely to be used by a user with high privileges. Also because these terminals are accessed locally, the *TTY* hooks would capture the login itself. This usually wouldn't happen in our case because the pseudo-terminals are usually spawned for remote users after the authentication occurs through another mechanism such as *SSH*.

5. Maintaining Access

The back-door is perhaps the most critical part of any root-kit. This is the part that allows the hacker to regain access to the victim. When we started this paper it was suggested that we base our paper off of an existing root-kit. The existing root-kit that we found was a user-mode program that listened on a raw socket for a signal (a specially crafted *ICMP* echo request). Upon receiving the signal, the program would connect-back to a hard-coded *IP* address and port with a *TCP* socket. The *STDIN*, and *STDOUT* file descriptors for the program were closed and then redirected to read/write to the socket. This provided a simple shell over the network. This was a good start.

It became apparent that this rudimentary back-door would need some modifications. First, using a hard-coded *IP* address was not desirable. It may not always be possible for an attacker to have the same *IP* address. So we modified the program to connect-back to the *IP* address that sent the signal.

The other obvious addition that needed to be made was a good system for triggering the back-door on the attacker's side. For this we simply wrote a *C* program that uses raw sockets to specially craft the signal (*ICMP Echo w/ ID = 0xDEAD*). When given the *IP* address of the target, the *ICMP* packet is crafted and sent. Once the target receives the signal it attempts to connect to the source address of the *ICMP* packet, this method of making the compromised computer connect back bypasses the majority of firewalls because egress connections are usually not filtered. To handle the connect back, the attacker simply uses a program like *netcat* or the newer *ncat* (packaged with *nmap*) to listen on the specified port and wait for the shell to connect.

6. Stealth

A major aspect of any root-kit is the ability to hide its presence on a machine. Initially it was difficult to find resources that provided guidance on how to implement these aspects. People in general shun the creation and use of root-kits. Most of the leads that were found were forum sites. On these forums there would be a person posing a question to the community on how to create different aspects of root-kits. Inevitably the rest of the posts would be along the lines of, "*There is no legitimate use for that, you can only use that in a root-kit, and I'm not going to tell you anything*." This trend was repeated across many different forums.

Without resources to work from this aspect of the paper was turning out to be very difficult. The turning point was noticing that the programs that were responsible for functions like *ls*, *ps*, and *netstat* all resided in the *bin* directory, which root has full access to. With access to these programs and where they resided a new possibility emerged, the idea of altering the results of these functions by changing them.

It was decided that we would focus on *ls*, *ps*, and *netstat*. These programs were deemed to be the most common tools that would indicate to the user that something on the system is amiss. Hiding the root-kit from these common programs would effectively hide the root-kit from most users.

The first attempt in changing these programs was to see if source code for each could be obtained and altered. The altered program would then replace the original program. Finding the code for these functions again proved difficult. While looking for the source code another idea emerged, we could place an intermediate program in place of the original program and have our program filter the data given to the user.

This method is not unlike a man in the middle of an attack. We would place our program in between the user and the intended program, like *ls*. The user would interact with our program instead of directly with the original program. To the user, our program would act and appear just like the original program. Our program would then interact with the original program in the same way a user would. Our program would then filter the results to hide anything we didn't want displayed to the user.

A side effect of using this method is that it is actually fairly simple to execute. A program could be written in C and use the "*system ()*" function to call the original function. Also because we were replacing the original program, by changing the name of the original program and naming our program to match the original name, we did not have to change anything to make it so our program would be executed anytime the user entered in the command like '*ls'*. Our program would automatically be in the execution path.

Once this tactic was decided upon the next step became determining ways to filter the results of the original function. Several methods were investigated but it was ultimately decided that three methods would be used. The three methods utilize normal aspects of the original function calls and normal aspects of the *Linux OS*.

While coming up with these ideas took a fair amount of time and effort, the results were very simple. This became a theme of this section of the paper. A purposeful attempt was made to keep this part of the paper as simple as possible. The idea for this was to investigate what someone could do without very much specialized knowledge of the *OS*. Every technique used could be done by any person with a very basic knowledge of the *Linux OS* and of the C programming language.

The first method used was to use the parameters of the original function. This method was used to overcome *ls*. The function call *ls* has a parameter -- *ignore=*. This parameter will take a character string and cause *ls* to ignore any files with that character string. The altered program takes input from the user and appends -- *ignore='pwn3d*'* to the end. That command is sent to the original program, which has now been renamed and moved. The original *ls* executes with the ignore parameter and displays the output to the user minus any files with that character string. To the user it appears identical to what they would expect, the only difference is that any files with the character string "*pwn3d*" are hidden from the user.

The second program we targeted was ps. Because parts of our root-kit are normally in a running state it would be revealed by the *ps* command. The *ps* command posed a difficulty. The original thought was to utilize *grep* to *grep* out the line containing the process associated with the root-kit. The problem with this method is that *grep* is then displayed as a process. To correct this we could also *grep* out the *grep*. This wasn't acceptable however because then all currently running *greps* would be hidden. This method would hide any *greps* that were running whether it was associated with what our program was doing or not. This was a problem.

The best way to make sure someone doesn't discover you're hiding something is to make sure you only hide exactly what you intended to. Humans are good at pattern recognition. If you hide one thing in a group of many, a person is not likely to notice the one thing missing. If you hide multiple things, and every one of those things is the same, you create a pattern. If a user normally uses *grep*, and performs *ps* and doesn't see any of the *greps*, it will cause the user to investigate. We don't want to have anything that would pique someone's curiosity. We want to hide exactly what we intended and nothing more.

With *grep* not being useful in this case an alternative method needed to be employed. The method chosen was to output the results of the *ps* to a file. Any function call can have its output piped to a text file using '>'. Once the results of the *ps* are in the file we could then read in the data, filter it, and then display it to the user. Because *grep* was not running at the time of the *ps*, the only data we needed to filter was data directly associated with the root-kit.

Our program took in the parameters entered by the user. The original *ps* program, which was moved and renamed, was then used with the parameters entered by the user. The output of the legitimate *ps* was piped into a text file. The text file would be written in the temp folder. This was done because files in the temp folder are not often examined and would at some point be deleted even if our attempts to delete it failed. Once the output was written to the file, the data from the text file was read back in. During the process of reading from the text file the line associated with the root-kit was ignored. The data was then outputted to the user and the text file was deleted.

The third program targeted was *netstat*. This program was targeted because our root-kit listens on a raw socket for a signal and then establishes a TCP connection to provide the attacker with shell access. All of these would be revealed by the *netstat* command. In this case we could use *grep* to filter the results for us. *grep* would not show up in a *netstat* so we didn't have to worry about hiding it. As with the parameter in *ls* that excludes certain data, *grep* has a parameter that will cause it to ignore certain items. The parameter *–v 'char-string'* will cause *grep* to read all of the data except anything matching that character string. *grep* is necessary here because *netstat* does not have its own parameter that allows for the excluding of specific data.

Our program worked by taking in the input from the user. The user's parameters were then sent to the original version of netstat with *grep –v '32569' | grep –v 'raw'* appended to the end. The number 32569 is excluded because the shell access program attempts to connect back to the attacker on *port* 32569. The term raw refers to a raw socket, which is used to listen for the signal which establishes the connection back to the attacker, this could raise some suspicion. As a result of the inclusion of the *greps*, the output to the user included no reference to the port in use or any raw sockets.

As stated earlier, the intent in this part of the paper was to see what could be done using simple means. By creating our own programs to intercept requests for common programs from the user we were able to effectively hide the root-kit's presence. Most users would not be able to detect the root-kit. This part of the root-kit is by no means sophisticated and as such a professional could probably easily detect it if they were looking for it, but that's not the point. The targets of most attacks are normal users without extensive technical skills. With the push for people to use Linux, especially with *netbooks*, a whole new community of targets has arisen. Using the methods we did, it is possible that any person with basic knowledge of programming and the Linux OS could effectively attack a lot of people. We only targeted three common programs. Using the same methods as above it could be possible to target many other programs.

7. Infection

There is more to a root-kit than the programs that it encompasses. The attacker has to have a good way of getting the root-kit onto the compromised server and quickly installing/configuring it. After it is installed, everything must quickly be cleaned up as to not accidentally tip off the system administrator to the malicious software. In our case, all of our programs were written in C/C++. This means that each of our programs would have to be individually compiled and then moved to the correct locations once it is on the victim computer. Obviously a script would be ideal for compilation/installation steps but a few questions remain. How do we ensure the script can run on the target computer? How do we get the script and the source code onto the target computer?

In order to ensure that our installation script can run on nearly any target computer we have to reduce the number of dependencies for the root-kit. The first choice we must make is what language do we write the script in? We could use *perl*, *python*, *ruby*, *java,* etc. The problem with programming/scripting languages is there are dependencies for them. *Perl*, *python*, *and ruby* need their respective interpreters, *java* needs the *java* runtime environment, etc. The solution we came up with was to use *bash* scripting. The reason to use *bash* scripting was simple; the root-kit is designed for *Linux*, and the *bourne-again shell* (*bash*) is the standard shell on nearly all *Linux* distributions. By using *bash* we can almost completely eliminate the language dependency issue. The other dependency issues that we run into are the requirement of *gcc*, *make,* and *kernel* module dependencies (for key-logger). *gcc* and make are extremely common defaults on most *Linux* distributions so again these are not much of a concern. The kernel module dependencies however have potential to be a problem. In order to compile a *Linux Kernel Module (LKM)* certain dependencies must be installed, and they are often not installed by default. We were unable to find a good solution to bypassing this dependency so for our purposes we will assume that the target has the dependencies installed, and if not the root-kit would be tailored differently (without a *LKM* included).

The other question we had to answer was how to get the script and the source code onto the target. We decided that to minimize the number of file transfers, we could include them into the bash installation script. This technique is nothing new; it is common among malware programs often called '*droppers*'. The technique uses the '*echo*' command (simply repeats parameters to *stdout*), in conjunction with the redirection operator to write a source code to files. So now, almost all the steps for infection have been identified, except one, how do you get the script on the target machine? This can really vary depending type of root access that the attacker has gained. If the attacker has an actual shell, the *wget* command can be used to download the script from a web server. If the attacker instead has compromised the target through some type of web or database exploit the script could be written to the victim by other means.

Once the installation script is on the target machine, infection is simple; the script needs executable permissions (*chmod* 755). Then the script needs to be run. When run, the script has a few defined phases that it goes through:

1. Write the source code out to source code files.
2. Compile the source code into the binaries (*gcc/make*).
3. Move necessary files (newly compiled and soon to be replaced).
4. Add reboot persistence (modify start up scripts)
5. Clean up (remove left over files, and installation script itself).

8. Conclusion

This paper encompasses a wide range of techniques involved in root-kits. The key-logger made use of sophisticate methods to attack the machine at the kernel mode level. The stealth part attacked the machine at the user mode level. This is a perfect example of how security must encompass everything. Security techniques monitoring the kernel alone would miss the user mode changes. Monitoring at the user mode only would miss the kernel level changes. Attacks can occur at all levels and in a nearly infinite number of combinations. Finally, the installation script showed that delivering an attack such as this could be done using simple and elegant methods. This paper was useful in that it exposed us to these different aspects and made us consider the range of possible attacks. By doing that, we began thinking about the entirety of the security challenge.

References

Colbert, J., A. Rubni & G. Kroah-Hartman, *Linux Device Drivers*, 3rd Edition., O'Reilly Media, Inc, 2005.

Stevens, W., *UNIX Network Programming*, Volume 1, Second Edition, Pearson Education Inc., 2002.

Writing Linux Kernel Key-logger, Volume 11, Issue 59, Phrack Inc. 18 (14).

ftp://ftp.kernel.org/pub/linux/kernel/v2.6, *Kernel Developers, Linux Kernel 2.6.29.1 Source Code*, 2009.