Global Information Assurance Certification Paper

A Review of Loadable Kernel Modules

by Andrew R. Jones, for SANS Security Essentials, GSEC Practical Assignment, version 1.2e

Introduction

For a long time, the kernel of any operating system was one large, indivisible executable image. This worked well to start with when kernels were small and advanced features were not needed. As the years have gone by, kernels have ballooned in size. Witness the Linux kernel: The compressed version of the source code for version 1.0 was about 1 megabyte. The compressed source for version 2.4 is over 25 megabytes. (kernel.org) Memory is a precious resource, so the technique of modularizing the operating system kernel was developed. This allows the kernel to be split up into many different pieces, and only those pieces that are needed at the moment are loaded into memory, to be unloaded again when they are no longer in use. These pieces that can so freely be loaded and unloaded are called loadable kernel modules (LKMs).

Anyone who is trained to think about the security implications of every action will immediately recognize this technique of using LKMs as a big problem. Without proper care, arbitrary code can be loaded into the kernel. Let us be clear about the difference between executing arbitrary code (as a user) and loading arbitrary code into the kernel. When executing code as a user process, that code is still bounded by what the kernel allows it to do. The kernel is the final authority on the system. If code can be loaded into the kernel, that final authority can be compromised and absolutely anything that is physically possible can be done. As pertains to installing a rootkit on a compromised system, this would allow an attacker to hide processes, files, sockets, and so on from every single program on the system without even trojanizing the executables. An attacker can build an entire impenetrable (by normal means) environment that a user cannot break out of and is not even aware of.

While most if not all modern operating system include loadable kernel modules, this paper will devote almost all of its attention to Linux. Rootkit LKMs and defensive LKMs are far more readily available for that platform than for any other. So much so, in fact, that some seem to believe that LKM stands for "Linux kernel module". This is not to say that exploits and defensive tools are not available for other platforms. They are just not available in the abundance that they are for Linux.

Theory

The theory of how loadable kernel modules actually do their work is quite simple. The kernel keeps an array of pointers to the locations of system calls, so all a loadable kernel module has to do is replace pointers in that list with pointers to its own definitions of the same functions. As an example, an LKM that wanted to prevent any other LKMs from being loaded would replace the system call to load a module with its own code that would essentially ignore the request, and possibly return an error. A more sophisticated version of the same LKM might make note of the name of the module, return success, and add the name of that module to any listing of all loaded modules, so all would seem as though the module had been loaded correctly.

There is one other option open to LKMs that want to do destruction: instead of replacing pointers, they can simply overwrite the existing system call. As long as the LKM's version of the system call is shorter than the original, no other code will be overwritten. This is effective at evading protective measures that do nothing more than compare before and after snapshots of the syscall pointer list.

Both proactive and reactive defenses against malicious LKMs are available. The most common proactive defense takes the form of "getting there first" -- that is, loading an LKM designed to protect the system before an attacker has the chance to load an LKM designed to do damage. Another option in the Linux kernel is to remove kernel capabilities. Kernel capabilities were introduced in the middle of the 2.2 kernel line, and they include the ability to stop the loading and unloading of any kernel modules.

Proactive measures, naturally, attempt to prevent any harm from being done to the system to begin with. They aggressively block anything that looks like malicious activity in the direction of the kernel. They also create the possibility of creating a virtual environment that will disguise the real system from an attacker -- exactly the same thing an attacker wants to do in reverse.

On the downside, protective LKMs will usually slow the system down, given that they have to execute their own checks before they allow anything to happen, many still rely on filesystem protection to guard the LKM and other critical system files on disk, and they may be too restrictive. Completely removing the ability to load an unload modules from the kernel removes quite a bit of flexibility from a system and may force it into poorer performance. Like any security measure, protective LKMs are not foolproof. As an example, if an LKM allows other modules to be loaded and unloaded, a malicious LKM that is not activated by default could be loaded into the kernel, then activated later through some other means (e.g. the /proc filesystem or a SIGHUP). At that point, it could change the system call table and go undetected.

Reactive measures against malicious LKMs use the tactic of comparing the system call table in memory with a known clean copy on disk. The advantages of this form of protection are the low overhead in terms of use of system resources and the lack of reliance on filesystem protection (given that the clean copy of the system call table is kept on a write-protected removable medium, as all such files should be).

The most obvious disadvantage to this approach is that it only tells the system administrator when the machine has already been compromised. Another disadvantage is that it is a manual process, and manual processes are a pain, and they are more prone to error. Finally, if such a method returns with a clean bill of health for the system, it is not proof that the system has not been compromised. Such methods can also be fooled. Most such modules do their checking when they are first loaded into the kernel, so if a machine has already been compromised and an LKM from a rootkit is currently residing in the kernel, it has probably already redirected the module insertion and deletion system calls to its own versions, and it could replace the system call table with a saved clean copy before loading any new modules, then replace the evil version of the system call table after the newly loaded module has been allowed to initialize and do its checking.

Classic LKMs

On Thursday, the 9th of October, 1997, heroin.c was posted to Bugtraq by Runar Jensen. It is the first known publicly available malicious kernel module, though it is intended for educational purposes, and it includes file, process, and module hiding capabilities. It was based on other published works concerning security weaknesses in the Linux kernel, but it opened the floodgate to more powerful clones.

One of the most famous heroin.c clones is knark, scribed by Creed in the middle of 1999. It builds on the ideas presented in heroin.c by allowing networking sockets to be hidden, and attempts to execute one program to be redirected to another program, most likely a trojanized executable also hidden by knark. It also allows for remote command execution and immediate promotion of any running process to root-level privileges, and it hides the fact that a network interface has been put into promiscuous mode (most likely to sniff the network for passwords and such).

Though our focus here is Linux, it would be good to keep in sight the fact that other operating systems are vulnerable to the same weakness. Little has been written for Solaris that has been made available to the general public, but Plasmoid/THC wrote a proof of concept LKM in 1999 that was supposed to be for Solaris 7 on either the SPARC platform, or on x86. Plasmoid also included a very good primer on writing LKMs for Solaris that could easily be used by anyone with a little bit of system programming knowledge in C to write something malicious. Also, the existence of an LKM detector for FreeBSD, mentioned briefly later in this paper, leads one to believe that malicious LKMs have also been written for that platform. In fact, the picture on FreeBSD seems a bit more complicated than that. Pragmatic/THC has written a paper on it that can be read at http://www.thehackerschoice.com/papers/bsdkern.html.

Protective tools

The time has come to discuss specific protective measures that are available today. Many exist only for the Linux 2.2 kernel, since the 2.4 kernel is still relatively new, and many open source tools are not maintained as well as the rest of us might like. Some are still actively being developed.

StMichael (http://sourceforge.net/projects/stjude/) is a project that was started by Timothy Lawless to address weaknesses in his own StJude package, which is a rule-based intrusion detection system. He recognized that StJude was still based on the assumption of a trusted and safe kernel, which was no longer a safe assumption. StMichael is still in its early stages of development, but it is already a good tool to add to a system administrator's arsenal, and it shows much promise for the future. StMichael falls into the category of proactive tools, as it is an LKM that is supposed to be loaded as soon as possible after system boot, and that constantly guards against changes to the system call table. The current version is 0.03, and that version replaces the sys_init_module() and sys_delete_module() calls with its own versions that do checks in addition to calling the original system functions. In addition to keeping a list of the original system call table, StMichael also does MD5 sums over the first few bytes of the actual function, so it will know if the function was overwritten in place. If the system call table is modified after the insertion of a module, StMichael will detect this and restore the call table to its pristine state. Good things are also in StMichael's future. In a response Mr. Lawless wrote to an inquiry of my own, he says that version 0.04 will address the following issues:

-- run-time patching of kernel functions (not syscalls) by hostile code
Extendig [sic] the use of MD5 (and probably SHA -- read below) to fingerprint the functions that StMichel [sic] is dependent on for opperation [sic] within the kenel. [sic]
This includes (but is not limited to) the "printk" (printf, but in kernel-space) kmalloc, kfree, memcpy internal kernel functions.
If one of these is corrupted, StMichael will attemtp [sic] to produce a log message (even if silent) if printk is not corrupt, sync the filesystems, and reboot.
If sync or reboot are corrupt, it will enter a busy loop. If printk is corrupt, it will not output the warning.
[...]

-- Addition of the SHA checksuming [sic] function.
This is being done, and combined with md5 checksums, to reduce the potential for birthday attacks.

-- More frequent re-validations [sic] of the kernel (Lawless, "Re: Experimenting with St. Michael")

Farther down the road is protection of files marked by the Linux second extended filesystem (ext2) as immutable. Mr. Lawless is also planning to roll StJude and StMichael into one package for those who want both, but will continue to develop StMichael as a standalone

module for those who don't want or can't use StJude.

The next tool under discussion is also a proactive tool very similar to StMichael in its mode of operation and capabilities, only it's a little more advanced in its development. It's called lsm, which is short for "loadable security module", and it's available at http://packetstorm.securify.com/linux/security/lsm.tar.gz. It already has the protection of immutable files that StMichael will have sometime in the future, and it includes raw disk protection for those who try to go around the standard system calls to access the hard drive. It is, however, quite restrictive. Once loaded, it does not allow any kernel modules to be loaded or unloaded. If that sounds attractive, it's not a whole lot more to just build a static kernel and be done with the entire problem. The performance of most systems will not be so adversely affected by this limitation, but it would definitely be a bad thing for a system that uses a lot of kernel modules (e.g. device drivers), but uses them only infrequently. It also has the potential to make adding and removing hardware or services a real pain.

Along the same lines, lcap, available from http://pweb.netcom.com/~spoon/lcap/, is a utility for Linux that allows the superuser to remove kernel capabilities, including the loading and unloading of modules. This is a proactive defense, but not a very practical one. The startup sequence for the system can be modified by an attacker to insert a module before lcap is called. This is true for many other defensive utilities, but what makes lcap potentially more restrictive is that, according to Patrick Reynolds, "To make capability bounding sets at all useful, you have to disable CAP_SYS_RAWIO, which governs access to /dev/mem. Be advised that doing so will break X and any other user-space program that needs raw access to memory or I/O ports." (Reynolds) This may or may not be a problem for a server on which X would not normally be run, but it would never fly for a desktop machine.

Carbonite is the first reactive tool on our list. It is an LKM extension of the userspace forensic tool cryogenic, and it can be found at http://www.foundstone.com/rdlabs/proddesc/carbonite.html. Cryogenic queried /proc for a list of running processes instead of relying on *ps(1)*. Carbonite takes this to the next level by reading the kernel's task list directly and writing the output to a file. This precludes the possibility of hidden processes. This tool has not yet been ported to the 2.4 kernel, and may never be.

Kstat is an LKM available at http://www.s0ftpj.org/tools/kstat.tgz that compares the current system call table to a known good copy in a disk file. This disk file is often called System.map and is provided with a Linux distribution. RedHat keeps it in /boot. It can also be generated at any time from a known-good kernel with the *nm(1)* command. If there are differences between the two, those differences are reported. As another reminder that exploits and tools exist for platforms other than Linux, sec_lkm.c (http://www.softpj.org/tools/sec_lkm.c) is a tool for FreeBSD that does very much the same thing that kstat does.

Miscellaneous

There are other concerns with LKMs that have not been touched on in this paper, and that ought to be mentioned, if only in passing. LKMs, both from rootkits and of a protective nature, are just as susceptible to bugs as all other software. LKMs in a rootkit are even more likely to be buggy, since they probably have not undergone close scrutiny by the large open source community. No matter the origin of the LKM, though, buggy code in something that gets loaded into the kernel can have disastrous effects. As an example, after loading lsm into the kernel on a test machine, the test machine spits out kernel debugging information to the screen every five minutes when rmmod tries to unload all unused modules. This is clearly not acceptable in a production environment, and this is also the mildest form of instability that could be introduced. A more serious form could turn into an unintentional denial of service attack. To its credit, Linux continues to run like a champ in this specific case, despite its internally inconsistent state.

Another vector of attack is the loading and unloading mechanism itself. RedHat released a bug fix for a local root vulnerability in the modutils package of RedHat 7.0 and 6.2 in November of 2000. The security announcement states that this release "supersedes [sic] the previous modutils errata packages." (RedHat) Obviously, there were other problems with modutils, as well, and any problem with modutils is potentially very serious.

It bears repeating that the unauthorized and undesired use of any system resources whatsoever on a computer is not acceptable, and by planting an LKM, and anything else that goes with it, on a system that does not belong to the person doing the planting, that person is committing a wrong. It does not matter if the compromised computer is adversely affected or not. The mere use of resources that do not belong to the user is prohibited, even if they are not being used. This is not LKM-specific, but it needs to be repeated on a constant basis.

Finally, there is a twist to the story of LKMs. In his post to Bugtraq mentioned earlier, Patrick Reynolds also commented that modules were not necessary to carry out a similar attack on the kernel. In his own words:

As an aside, more fun with module security... Even if you compile a kernel with module loading completely disabled, a clever attacker could still load custom, module-like code into the kernel using /dev/mem. It's trickier than changing cap-bound, but it's still feasible, because page tables and syscall tables are similarly exposed through /dev/mem. (Reynolds)

Thus, while it is certainly necessary to control the use of LKMs, similar effects can be achieved in at least one other way, and system administrators need to be aware of those other ways in order to guard against them.

Conclusion

Since LKMs provide something close to the ultimate exploit, the kernel is likely to be a battleground over the next few years, and possibly far into the future. The best defense is still to build a monolithic kernel whenever possible, but this is becoming less and less practical. Tools such as those covered in this paper are easier to deploy on a wide basis than having to build a kernel for each and every machine under a system administrator's care.

Let us never forget that tools to protect the kernel against malicious LKMs should still only be part of a much wider intrusion prevention/detection system. Such a system typically needs to include filesystem protection to prevent alteration of system startup files, or the binaries for the tools themselves. Scans external to the machine need to be done periodically, also, to ensure that the list of ports that the operating system is reporting as being open matches up with reality. Two relatively well-known tools to do these two things are LIDS (Linux Intrusion Detection System), and nmap.

A system administrator can also consider LKMs as a tool for setting up a virtual environment to hide the real system from attackers. They could even be used very effectively in setting up a honeypot or honeynet. As always, we need to use a superset of the tools that attackers are using in order to stay one step ahead of them.

The biggest problem with LKMs is the trust that the kernel puts in anything it can read from the disk that the superuser says to load into memory. The kernel cannot be allowed to place such whole-hearted trust in something that can so easily be compromised. The author hopes to see a scheme put into place in the next couple of years to allow for authentication between the kernel and any module it is told to load. What form that would take is for someone else to devise, and hopefully someone will do that devising very soon. Until then, we all need to watch our kernels with a hawk's eye.

Bibliography

Clemens, Jonathan. "Knark: Linux Kernel Subversion". Intrusion Detection FAQ. SANS Institute. <http://www.sans.org/newlook/resources/IDFAQ/knark.htm> (9 June 2001)

Creed (psuedo.). knark-2.4.3.tgz. Ported by Cyberwinds (pseudo.). Packet Storm. <http://packetstorm.securify.com/UNIX/penetration/rootkits/knark-2.4.3.tgz> (10 June 2001)

Foundstone, Inc. "Carbonite v1.0 - A Linux Kernel Module to aid in RootKit detection." Foundstone, Inc. <http://www.foundstone.com/rdlabs/proddesc/carbonite.html> (10 June 2001)

FuSyS (pseudo.). "KSTAT - Kernel Security Therapy Anti-Troll". s0ftpr0ject2000. 19 August 2000. <http://www.s0ftpj.org/tools/kstat.tgz> (10 June 2001)

Jensen, Runar. "Malicious Linux modules". 9 October 1997. Bugtraq. <http://www.dataguard.no/bugtraq/1997_4/0059.html> (9 June 2001)

kernel.org. <ftp://ftp.kernel.org/pub/linux/kernel/> (9 June 2001)

Lawless, Timothy. "Re: Experimenting with St. Michael". E-mail to Andrew Jones. 6 June 2001.

Lawless, Timothy. "Saint Jude, The Model". Version 1.0. 2 November 2000. SourceForge. <http://prdownloads.sourceforge.net/stjude/StJudeModel.pdf> (9 June 2001)

Lawless, Timothy. StMichael_LKM-0.03.tar.gz. 2 June 2001. SourceForge. <http://prdownloads.sourceforge.net/stjude/StMichael_LKM-0.03.tar.gz> (10 June 2001)

Paul (paul@ihaquer.com). lsm.tar.gz. 2 May 2001. Packet Storm. <http://packetstorm.securify.com/linux/security/lsm.tar.gz> (10 June 2001)

pIGpEN (pseudo.). "LKM Detector". s0ftpr0ject2000. 18 April 2000. <http://www.s0ftpj.org/tools/sec_lkm.c> (10 June 2001)

Plasmoid/THC (pseudo.). "Solaris Loadable Kernel Modules". Version 1.0. 22 December 1999. Packet storm. <http://packetstorm.securify.com/groups/thc/slkm-1.0.html> (9 June 2001)

Rauch, Jeremy. "Introduction to Linux Capabilities and ACL's". 28 August 2000. SecurityFocus.com. <http://www.securityfocus.com/frames/?focus=linux&content=/focus/linux/articles/capabilities.html> (9 June 2001)

RedHat, Inc. "Red Hat Linux General Security Advisory". 27 November 2000. <http://www.redhat.com/support/errata/RHSA-2000-108.html> (11 June 2001)

Reynolds, Patrick. "Linux capability bounding set weakness". 22 June 2000. Bugtraq. <http://pweb.netcom.com/~spoon/lcap/bugtraq.txt> (10 June 2001)

Spoon (pseudo.). "LCAP". 17 September 2000. <http://pweb.netcom.com/~spoon/lcap/> (10 June 2001)

Last Updated: May 26th, 2017

Upcoming Training

SANS Stockholm 2017	Stockholm, Sweden	May 29, 2017 - Jun 03, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
Security Operations Center Summit & Training	Washington, DC	Jun 05, 2017 - Jun 12, 2017	Live Event
Community SANS Ottawa SEC401	Ottawa, ON	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC401: Security Essentials Bootcamp Style	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Secure Europe 2017	Amsterdam, Netherlands	Jun 12, 2017 - Jun 20, 2017	Live Event
Community SANS Portland SEC401	Portland, OR	Jun 12, 2017 - Jun 17, 2017	Community SANS
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Minneapolis SEC401	Minneapolis, MN	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Phoenix SEC401	Phoenix, AZ	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Mentor Session - SEC401	Macon, GA	Jul 12, 2017 - Aug 23, 2017	Mentor
Mentor Session - SEC401	Ventura, CA	Jul 12, 2017 - Sep 13, 2017	Mentor
Community SANS Atlanta SEC401	Atlanta, GA	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Colorado Springs SEC401	Colorado Springs, CO	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANSFIRE 2017 - SEC401: Security Essentials Bootcamp Style	Washington, DC	Jul 24, 2017 - Jul 29, 2017	vLive
Community SANS Charleston SEC401	Charleston, SC	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Fort Lauderdale SEC401	Fort Lauderdale, FL	Jul 31, 2017 - Aug 05, 2017	Community SANS
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event