Linux Kernel Rootkits

Advanced Techniques

Ilya V. Matveychikov
Ighor Augusto

October 24, 2018

release

Ilya V. Matveychikov – Linux kernel addict, security researcher, reverse engineer https://github.com/milabs

Ighor Augusto - Just a guy who thinks bytes come before titles! https://github.com/f0rb1dd3n

This talk is about:

- Programming
- Linux kernel rootkits

This talk is NOT about:

- Malware
- Exploitation

Please note that all the techniques described are publicly available as well as Linux kernel source code.

- bla bla bla

Rootkit (root's toolkit) - kind of software designed to provide continued privileged access to the target while actively hiding its presence.

- affects computers, servers, smartphones ...
- provides privileged access ...
- hides itself ...

Rootkits can be classified according to the environment they are operating (living) in.

- User-mode (UM) rootkits: typically, LD_PRELOAD-based
- Kernel-mode (KM) rootkits: typically, LKM-based
- Firmware-based (FW) rootkits (UEFI)
- Hypervisor (HV) rootkits
- Hybrid rootkits (the mix)

Kernel-based rootkit. Why to have it?

- Why not?
- Altering whole the system: you can do (almost) everything.
- Extremely hard to detect from the user mode.
- Overall, it's a great challenge.

Introduction
A rootkit's purpose

Challenges of writing (a good) Linux kernel-mode rootkit.

- Kernel-mode programming requires an in-depth knowledge of how the OS-kernel and hardware works. Every little 🐞 in the code will became a big pain in the ass.
- The scope of work (in terms of required features) must have been defined before the start of development.
- Fighting against a loooooot of kernel versions, distributions and RHEL «backport hell»-like approach (for example, `kernel-2.6.32-754.2.1.el6` is not actually `2.6.32`).

Constraints of Linux kernel-based rootkit.

- Non-stable (volatile) kernel API. It's hard to manage all the possible versions of the kernel.
- In general, LKM-based rootkit requires to be built for every kernel version. It's (almost) impossible to have the only rootkit's binary that fits all the targets.
- Most likely the rootkit will not survive the kernel update.
- Kernel API is not indented for doing non-kernel tasks. Try to download & execute zipped payload using HTTP(s).

A common subset of (Linux) kernel rootkit features.

- Be able to survive the reboot, update.
- Be able to alter whole the system behaviour (not only the kernel).
- Be able to hide files, directories, processes, network connections, users and other resources.
- Be able to evade against the detection (hide own components, filter kernel & audit logs, restore «taint»-like flags, . . .).
- Be able to provide the payload (keylogger, backdoor/shell, gain privileges, . . .).

Symbol - a symbolic name of some object (function or variable). Treat the symbol as a way to get an object's address by using just it's name (for ex., `sys_call_table[]`).

- Symbols can be exported or non-exported.
- Public kernel API consists of only exported symbols (`EXPORT_SYMBOL()`-like macros is used).
- Private kernel API consists of public kernel API and any other symbols available.

While public kernel symbols are always available it's often required to use private symbols and there are few common ways to find them.

- Read and parse `/proc/kallsyms` file.
- Use `kallsyms_lookup_name()` method.
- Use `kallsyms_on_each_symbols()` method.
- Use signatures and/or by disassembling the kernel's code (from inside the kernel, of course).

NOTE that a) `kallsyms`-interface might not be compiled in and b) `System.map` is mostly useless nowadays because of ASLR.

Write Protect (bit 16 of CR0) - when set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages [1]...

This flag facilitates implementation of the **copy-on-write** method of creating a new process (forking) used by operating systems such as UNIX.

[1] http://vulnfactory.org/blog/2011/08/12/wp-safe-or-not

In case you want to use CR0, use the following to disable write protection (on this CPU):

```
static inline \
unsigned long pax_open_kernel(void) {
  unsigned long cr0;

  preempt_disable();
  barrier();
  cr0 = read_cr0() ^ X86_CR0_WP;
  BUG_ON(unlikely(cr0 & X86_CR0_WP));
  write_cr0(cr0);
  return cr0 ^ X86_CR0_WP;
}
```

Listing 1: Disable Write Protection

In case you want to use CR0, use the following to enable write protection (on this CPU):

```
static inline \
unsigned long pax_close_kernel(void) {
  unsigned long cr0;

  cr0 = read_cr0() ^ X86_CR0_WP;
  BUG_ON(unlikely(!(cr0 & X86_CR0_WP)));
  write_cr0(cr0);
  barrier();
  preempt_enable_no_resched();
  return cr0 ^ X86_CR0_WP;
}
```

Listing 2: Enable Write Protection

In case you want to use CR0, use the following sequence to modify ready-only memory:

```
pax_open_kernel();
sys_call_table[__NR_open] = my_sys_open;
... # system behaviour is undefined
pax_close_kernel();
```

In practice, approach of using WP-bit of CR0 works nearly all of the time. But there are some caveats to be aware of when using this trick in real life scenarios.

- There is a window of undefined system behaviour between `pax_open_kernel()` and `pax_close_kernel()` calls.
- WP is disabled/enabled only for CPU which is calling those methods. So, further memory modification must be done from the same CPU.
- Hypervisor (if any) is able to detect flipping of WP-bit of CR0 register which might be treated as a **sign of attack**.

The better approach is to create a writable mapping of read-only region using `vmap`.

- For each page in region translate it's virtual address to `struct page`. Use `virt_to_page()` for kernel and `vmalloc_to_page()` for modules.
- Use `vmap()` to map those pages to virtually contiguous space using page protection required (`PAGE_KERNEL`).
- Use `vunmap()` to unmap the mapping after using.

```c
void *map_writable(void *addr, size_t len) {
  void *vaddr = NULL;
  void *paddr = (void *)(addr & PAGE_MASK);
  struct page *pages[ ... ];

  for (int i = 0; i < ARRAY_SIZE(pages); i++) {
    if (__module_address((ulong)paddr))
      pages[i] = vmalloc_to_page(paddr);
    else pages[i] = virt_to_page(paddr);
    if (!pages[i]) return NULL;
    paddr += PAGE_SIZE;
  }

  vaddr = vmap(pages,
               ARRAY_SIZE(pages),
               VM_MAP, PAGE_KERNEL);
  return vaddr ? \
    vaddr + offset_in_page(addr) : NULL;
}
```

The better approach is to create a writable mapping of read-only region using `vmap`:

```
size_t slen = \
  __NR_syscall_max * sizeof(sys_call_ptr_t);
sys_call_ptr_t *sptr = \
  map_writable(sys_call_table, slen);
sptr[__NR_open] = my_sys_open;
...
vunmap(sptr);
```

Hooking - range of techniques used to alter the behaviour of some system. Hooking various kernel functions is the base of kernel rootkit's live.

- Hooking system calls by replacing pointers in `sys_call_table[]` and `ia32_sys_call_table[]`.
- Hooking virtual methods calls (vtable-like) by replacing pointers in tables like `struct file_operations`.
- Hooking of kernel symbols by patching their code (will be discussed).
- Registering any kind of callbacks and notifiers (for example, `register_module_notifier()`)
- Registering LSM security callbacks (hooks).

KHOOK[2]- automatic kernel function hooking engine designed to simplify our live. Provides simple API for hooking kernel symbols (functions).

- Uses code patching technique which is based on overwriting target function prologue with `JMP xxx`. Simplest, reliable and 100% working solution.
- Uses in-kernel length disassembler engine (LDE) to get the number of instructions to save before overwriting.
- Allows to make a call to the original function while this function is being hooked.
- For each function hooked a use-counter maintained. This prevents unhooking of symbols which are in use.

[2] https://github.com/milabs/khook

KHOOK provides a set of macros to make the hooker's life a bit easier.

- Use `KHOOK(xxx)` macro for declaring a hook of function `xxx` which has it's prototype declared (somewhere).
- Use `KHOOK_EXT(xxx, typeof(arg0), typeof(arg1), ...)` macro for declaring a hook of function `xxx` which has not have it's prototype declared.
- Use `KHOOK_GET(xxx)` and `KHOOK_PUT(xxx)` macros for managing symbol's hook use counter.
- Use `KHOOK_ORIGIN(xxx, args...)` to call to the original function as it was not hooked.

Use `khook_init()` and `khook_cleanup()` to init and cleanup the engine. Calling to `khook_init()` causes all declared hooks to be installed while calling to `khook_cleanup()` does the reverse.

Add the following includes to your code:

```
#include "engine/engine.h"
#include "engine/engine.c"
```

Add the following options to the linker:

```
ldflags-y += -T$(src)/engine/engine.lds
```

```
KHOOK(inode_permission);
static int \
khook_inode_permission(struct inode *i, int m)
{
  int ret = 0;

  KHOOK_GET(inode_permission);
  ret = KHOOK_ORIGIN(inode_permission, i, m);
  printk("%s(%p, %08x) = %d\n", \
    __func__, i, m, ret);
  KHOOK_PUT(inode_permission);

  return ret;
}
```

Listing 3: Hooking of inode_permission() example

Hiding of processes is the one of the most popular rootkit features. There is no publicly available Linux kernel rootkit which can hide processes well.

This task is not complex by itself but it requires to have a good knowledge of how the kernel works. At least how it manages the processes.

Implementation of hiding processes requires the following to be done:

- Managing the processes lifecycle. Be able to attach/detach some attributes to processes while forking and executing.
- Managing the processes visibility by filtering out `/proc` and some system calls.
- Managing the processes CPU-time accounting.

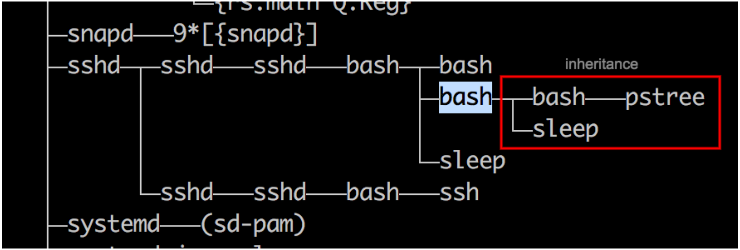
Hook `copy_creds()` function to be able to attach attributes to processes at fork time. Inherit parent process attributes for all direct children, if required.

Hook `exit_creds()` function to be able to detach attributes from the processes at exit time.

In it's simplest form attaching/detaching attributes to processes may be implemented by using one of unused (reserved) bits of `task->flags`, for example: `0x80000000`.

Common Techniques
Hiding of processes: lifecycle

Illustration of the inheritance of attributes of hidden processes.



Hook `next_tgid()` function to be able to filter out `/proc/PID` like directory entries. Just skip all the tasks with “hidden” attribute set from being iterated.

NOTE: There is no reason to hook `getdents()` to filter out `/proc/PID` content. Do not do it.

Hook `find_task_by_vpid()` function to be able to fight against `unhide`[3] by altering some system calls:

- `getsid`
- `getpgid`
- `getpriority`
- `sched_getparam`
- `sched_getaffinity`
- `sched_getscheduler`
- `sched_rr_get_interval`
- `kill`

[3] https://github.com/Enrico204/unhide

CPU utilization is the sum of work handled by a processor unit. It's a good idea to exclude hidden processes from being accounted.

Hook `account_process_tick()` function to be able to exclude ticks spent by a hidden processes from system wide ticks accounting.

Hiding of files and directories is the one of the most popular rootkits features.

Being implemented as a part of Linux kernel rootkit it allows to hide filesystem stuff from being observed by system administrators and other users. Sure, this will work only from the moment LKM is loaded.

Implementation if hiding files and directories is based on the following:

- Filtering the access to files or directories by using their full path (`open()`-like system calls).
- Filtering files and directories from being listed (`filldir()`-like system calls).

To be able to filter out the access to files or directories by using their filenames hook the following non-public kernel functions:

- `do_sys_open`
- `user_path_at`
- `user_path_at_empty`

To be able to filter out files and directories from being listed hook the following non-public kernel functions:

- `filldir`
- `filldir64`
- `fillonedir`
- `compat_filldir`
- `compat_filldir64`
- `compat_fillonedir`
- `__d_lookup`

Demo I

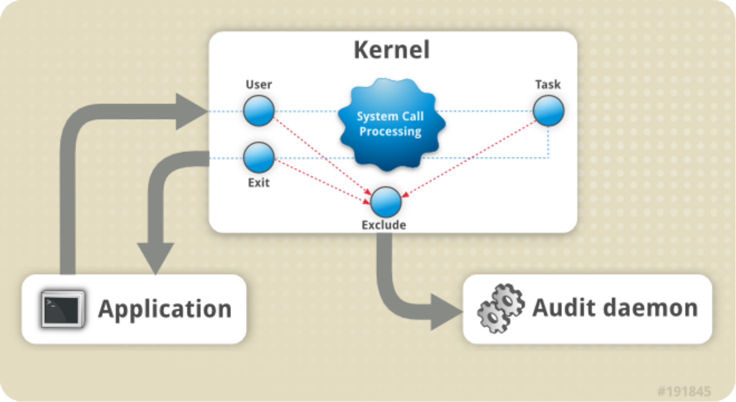
Live demo of Reptile r00kit

https://github.com/f0rb1dd3n/Reptile

The Linux Audit system provides a way to track security-relevant information on your system. It might be useful for:

- Watching file access.
- Monitoring system calls.
- Recording commands run by a user.
- Monitoring network access.
- . . . and so on

The Audit system consists of two main parts: the user-space applications and utilities, and the kernel-side system call processing. The architecture is show below:



There is a way to completely disable (bypass) auditing for the certain task. Use the following:

- Hook `audit_alloc()` function.
- Inside the hook just clear `TIF_SYSCALL_AUDIT` for the task if required.

As the result there will be completely no audition for all tasks without `TIF_SYSCALL_AUDIT`. By design. Really.

```
KHOOK(audit_alloc);
static int \
khook_audit_alloc(struct task_struct *t)
{
  int err = 0;

  KHOOK_GET(audit_alloc);
  if (task_audit_disable(t)) {
    clear_tsk_thread_flag(t, TIF_SYSCALL_AUDIT);
  } else {
    err = KHOOK_ORIGIN(audit_alloc, t);
  }

out:
  KHOOK_PUT(audit_alloc);
  return err;
}
```

Advanced Techniques

Filtering the kernel log

Linux kernel log is a standard way to log the information by the kernel. The information logged can be obtained by user-space programs like `dmesg` or (`journalctl`).

It's mandatory for Linux kernel rootkit to be able to filter-out kernel log messages like the following:

```
[vagrant@localhost khook]$
[vagrant@localhost khook]$ dmesg | grep signature
[    4.877850] vboxguest: module verification failed: signature and/or
[vagrant@localhost khook]$
```

There are 2 ways of getting messages from the log:

- Using `syslog(2)`
- Using `/proc/kmsg`

`syslog` interface is an old-style way to get messages from the kernel. It's implemented internally by `do_syslog()` function.

`/proc/kmsg` interface is the new-style one and it's implemented internally by `devkmsg_read()` function.

There are few types of kernel log messages ...

syslog message

```
"<%u> message-text\n" (no timestamp)
"<%u>[%5lu.%06lu] message-text\n"
```

/proc/kmsg message

```
"%u,%llu,%llu,%c,[,...];message-text\n"
" key=value\n[ key=value\n]" (options, if any)
```

Filtering-out messages from the kernel's log requires us to a) hook proper symbols b) let them do their job and c) post process written out data.

- Hook `do_syslog()` and `devkmsg_read()`.
- Let them do their job by writing messages to user-space processes (like `dmesg`), when requested.
- Having the address (and the length) of just filled user-space buffer do the following . . .
- . . . make an in-kernel copy using `memdup_user()`
- . . . filter it out splitting messages by `newline`
- write out filtered result altering the final length (if changed)

It's a good idea to have a tiny LKM-module (loader) which loads the encrypted payload. That's something we call *Matryoshka*[4].

[4] https://github.com/milabs/kmatryoshka

The technique is pretty simple.

- Write your `payload.ko` in form of LKM without any restrictions.
- Write the `loader.ko` module which will embed the encrypted `payload.ko` as is.
- Use `user_addr_max()` to get the current value of user-space address limit (SEG).
- Extend the user-space address limit to fit the decrypted payload and use `sys_load_module()` to load it.
- Restore the user-space address limit by using `user_addr_max()` and SEG value.

The example of using *Matryoshka* technique is shown below. The module `parasite_loader.ko` hosts the encrypted body of `parasite.ko` and then loads it from inside the kernel.

```
vagrant@ubuntu-xenial$
vagrant@ubuntu-xenial$ hexdump -C parasite/parasite.ko | grep -A1 -B1 H2
000000c0  61 73 69 74 65 2e 0a 49  27 6d 20 77 69 6c 6c 69  |asite..I'm willi|
000000d0  6e 67 20 74 6f 20 61 74  74 65 6e 64 20 48 32 48  |ng to attend H2H|
000000e0  43 20 32 30 31 38 20 63  6f 6e 66 65 72 65 6e 63  |C 2018 conferenc|
vagrant@ubuntu-xenial$
vagrant@ubuntu-xenial$ hexdump -C parasite_loader/parasite_loader.ko | grep -A1 -B1 H2
vagrant@ubuntu-xenial$
vagrant@ubuntu-xenial$ sudo insmod parasite_loader/parasite_loader.ko
insmod: ERROR: could not insert module parasite_loader/parasite_loader.ko: Invalid parameters
vagrant@ubuntu-xenial$
vagrant@ubuntu-xenial$ dmesg
[57459.318494] parasite_loader is saying:
               Hello, I'm the loader.
               I will load the parasite for you.
[57459.319779] parasite is saying:
               Hello, I'm the parasite.
               I'm willing to attend H2HC 2018 conference.
vagrant@ubuntu-xenial$
```

It's possible to get rid of static C-strings at compile time using the following approach (GCC-only, but who cares).

```
$ echo 'printk("hello␣world\n");' | perl
    destringify.pl

printk(({ \
  unsigned int *p = __builtin_alloca(16); \
  p[0] = 0x6c6c6568; \
  p[1] = 0x6f77206f; \
  p[2] = 0x0a646c72; \
  p[3] = 0x00000000; \
  (char *)p; \
}));
```

Listing 4: Static C-string compile-time obfuscation

Demo II

Live demo of some other stuff