Reptile's Custom Kernel-Module Launcher

10 Nov 2024

Introduction

> “In REPTILE version 2.0, the original developer of REPTILE altered how the Kernel-level component is loaded, switching from using insmod to a custom launcher. The launcher Mandiant observed UNC3886 use throughout their operations, based on the custom launcher, was updated with a new function to daemonize a process.” — Mandiant, Cloaked and Covert: Uncovering UNC3886 Espionage Operations, 2024.

This analysis will examine how the Reptile rootkit loader bypasses the standard Linux `insmod` command for loading Kernel modules and will explore methods for detecting the use of this custom loader.

Analysis

Within the Makefile, a random 32-bit hexadecimal value is generated (stored in the `RAND2` variable) each time the Kernel module is compiled. This value serves as the encryption key for the Reptile Kernel object, making it difficult to identify the module through simple hash searches or hex value hunts on the filesystem. The encrypted Kernel module is stored in `reptile.ko.inc`.

```
RAND2 = 0x$(shell cat /dev/urandom | head -c 4 | hexdump '-e"%x"')
[..]
reptile: $(LOADER)
        @ $(ENCRYPT) $(BUILD_DIR)/reptile.ko $(RAND2) > $(BUILD_DIR)/reptile.ko.inc
        @ echo "  CC      $(BUILD_DIR)/$@"
        @ $(CC) $(INCLUDE) -I$(BUILD_DIR) $< -o $(BUILD_DIR)/$@
```

The custom loader is defined in loader.c. Here, the encrypted Kernel object is loaded into a statically allocated array named `reptile_blob`:

```
static char reptile_blob[] = {
#include "reptile.ko.inc"
};
```

The core of the custom launcher involves decrypting `reptile_blob`, copying it to `module_image`, and then calling `init_module` to load it:

```
len = sizeof(reptile_blob);
do_decrypt(reptile_blob, len, DECRYPT_KEY);
module_image = malloc(len);
memcpy(module_image, reptile_blob, len);
init_module(module_image, len, "");
```

The decryption process is handled by the `do_decrypt` function (source code here) :

do_decrypt

`do_decrypt` and `do_encode` are the same function, as defined in a macro before.

```
static inline void do_encode(void *ptr, unsigned int len, unsigned int key)
{
        while (len > sizeof(key)) {
                *(unsigned int *)ptr ^= custom_rol32(key ^ len, (len % 13));
                len -= sizeof(key), ptr += sizeof(key);
        }
}
```

This function decrypts the Kernel module by:

- Performing a bitwise XOR operation between the data and a rotated value derived from the key and len.
- Applying a custom_rol32 function to further obfuscate the data by rotating bits.

custom_rol32

```
static inline unsigned int custom_rol32(unsigned int val, int n)
{
        return ((val << n) | (val >> (32 - n)));
}
```

This function performs a `rotate left` operation on a 32-bit integer. The val is shifted left by n bits, and the overflowed bits are “rotated” back to the right end. This is a common bitwise operation used in encryption algorithms to add an extra layer of obfuscation.

The decrypted Kernel module is then loaded using the `init_module` system call, bypassing the standard `insmod` command:

init_module()

The init_module system call loads a Kernel module into the Linux Kernel at runtime. Essentially, it allows a program to insert a Kernel module programmatically, functioning similarly to the insmod command executed from the command line.

The function takes three key parameters:

- `module_image`: This is the compiled Kernel module code that will be loaded into the Kernel.
- `len`: Specifies the size (in bytes) of the module_image, informing the Kernel how much data to read.
- `param_values`: Allows passing initialization parameters to the module, similar to how you would provide parameters when using insmod from the command line.

In the Reptile rootkit, a macro is defined to simplify the process of invoking `init_module`. By using the macro, the code bypasses standard C library functions and directly interfaces with the system call:

```
#define init_module(module_image, len, param_values) syscall(__NR_init_module,
module_image, len, param_values)
```

This direct syscall approach allows the custom loader to insert the Kernel module without relying on higher-level functions, which can help avoid detection mechanisms that monitor typical command-line usage patterns.

Detection

The custom loader circumvents standard Linux detections that rely on monitoring for insmod commands, such as this Elastic Security rule (source):

```
process where host.os.type == "linux"
and event.type == "start"
and process.name == "insmod"
and process.args : "*.ko"
```

However, using **auditd_manager** rules can still enable detection by monitoring the actual syscalls used to load Kernel modules. The detailed process to enable and configure auditd_manager is explained on the `Kernel Driver Load` rule page from Elastic (source).

```
driver where host.os.type == "linux"
and event.action == "loaded-kernel-module"
and auditd.data.syscall in ("init_module", "finit_module")
```

We are not monitoring for a command (`insmod`) but for a syscall (`init_module`), which the custom reptile loader uses under the hood. This might be enough to catch the loading of a malicious Kernel module on a server. If such a module is loaded, it might taint the Kernel and give us also a heads-up that something fishy is going on. I wrote about Tainted Kernels before.