Volatility contest 2023 - Submission document

Abyss Watcher

December 2023

Contents

Abstract

Researches presented in this document should be considered as an unique submission. As specified in the contest submission guideline, this document contains a section *Why should I win the 2023 Volatility contest ?*. Code and ressources are donated under Volatility Foundation, Inc. Individual Contributor Licensing Agreement.

1 Introduction

Memory forensics is a field of digital forensics focusing on the analysis of a system's volatile memory. By extracting the state of a running computer, data and artifacts only available at runtime can be uncovered and reconstructed. Indeed, network connections, processes list, encryption keys or any data that needs to be processed live will be stored in the RAM (Random Access Memory) [11]. Parsing a memory sample (also called *memory dump*) is a complex task, and needs a deep understanding of how hardware, kernel and software operates [7, see /about]. The *Volatility Foundation* [7] is an organization developing an open source and free framework called Volatility. The framework was originally developped in 2007, under the name *Volatility2*, before being rewritten and released in 2020 with a new project : *Volatility3* [7, see /3].

Once a year, the *Volatility Foundation*, organizes a contest to create innovative, interesting, and useful extensions for the framework [7, see /2023]. This contest is a great way to inspire researchers and forensics enthusiasts to develop and create new ressources. In accordance with this, the following document will present a set of extensions to the Volatility framework, including an extended database of kernel symbols and a set of plugins targeting the detection of Linux rootkits. These were conducted by myself (*Abyss Watcher*), a cybersecurity engineering student.

2 Providing a collection of symbols for the most popular Linux distributions

To analyze a memory sample, Volatility needs to know how the memory was organized by the kernel, including addresses for structures, templates, indexes etc., which cannot be detected, for now, with a sole memory sample. These informations are generally gathered by extracting the debug symbols for a kernel, which are provided by the distributor itself (Windows, Ubuntu, macOS...). Debug symbols are generated when compiling a kernel, alongside the code, to gain access to information about the source code, like names, identifiers or variables [16].

Volatility2, the previous framework version, relies on the local compilation of a kernel (assuming source code is available), to generate the required symbols and types. They are wrapped in a "profile". On the other hand, Volatility3

relies on the use of *dwarf2json* [5], which iterates on symbols and types of a kernel, and wraps them in a JSON ISF (Intermediate Symbol File).

Ideally, symbols for every kernel release should be easy to locate and use for Volatility end users.

2.1 State of the art

Due to the variety of available Linux distributions [17], providing appropriate symbols at runtime can become a complex task. Indeed, each distribution provider customizes the Linux kernel depending on its needing, and may release a new version frequently.

Accessing the needed dependencies to generate Volatility symbols may also be a challenge, regarding the fact that each provider arranges its packages sources differently. Finally, it is also quite a challenge for non-initiated users to understand the complete symbols generation process. Having myself, in the past, been confronted to these steps when using Volatility for the first time, I quickly realized it was sometimes more challenging for users to get the appropriate symbols than to analyze an evidence.

In this way, I started searching for existing work and automated projects (like the one made by JPCERTCC [8] for Windows). Here are the most significative I found, at the time :

- https://github.com/p0dalirius/volatility3-symbols (created by p0dalirius) : 637 symbols
- https://isf-server.techanarchy.net (created by kevthehermit) : 1327 symbols

Each provides a collection of symbols (only Volatility3 here), for many distributions. Unfortunately, many ISF I was looking for wheren't available in those repositories. After exploring mirrors of Ubuntu and Debian, I quickly realized that the number of existing kernels was a lot higher than I expected, explaining why they were missing from the existing ISF collections. Searching the Web for more collections, I wasn't able to find a complete one. Thereby, I decided to start my own project.

2.2 Objectives

The main goal is to generate symbols/profiles for any kernel ever released by a distribution, for a specific architecture. Here are the key points :

1. Allow analysts to target the core of their work : examine an evidence memory sample.
2. Symbols for a specific kernel release only need to be generated once, it is unnecessary for every analyst to recreate them.

3. Following point 2, providing end users the ability to generate symbols easily on a per-demand basis is great, but directly providing the resulting symbols allows ressources and time gains.

4. Archive symbols for Volatility use, in case remote sources close in the future.

Even if Volatility3 is now the project to focus on, Volatility2 is still widely used in the memory forensics community. Doing so, I decided to work with both. Furthermore, debug symbols might be missing for a kernel, so the easiest way to do an analysis, is to use Volatility2 and a profile.

2.3 Technical requirements

Here are a few requirements, needed to efficiently generate Volatility symbols :

- High speed Internet connection (to fetch packets)
- A powerful CPU
- At least 14GB of RAM (can be swapped), as dwarf2json will load a lot of data in memory

2.4 Restrictions

I chose GitHub to host all generated files, as it is easy to navigate for users, and runs on high quality servers. However, git, by its nature, will keep track of every file history, even "blobs". Doing so, and as we will see, the repositories quickly become gigantic. To avoid abusing GitHub storage, I chose to only work with :

- Ubuntu (amd64, i386)
- Debian (amd64)
- KaliLinux (amd64)
- AlmaLinux (amd64)
- RockyLinux (amd64)

2.5 Project core components

The project is mainly built with Python3, and uses Docker containers to create a generation environement.

2.5.1 Containers

Containers allow to create a "sandboxed" environement which can be easily customized, created or destroyed. They also avoid messing the host machine with unneeded dependencies and files.

2.6 Needed packages

To generate Volatility2 profiles, the kernel source is needed. For Volatility3, only the debug symbols from the *vmlinux* file [18]. On most APT based distributions (Ubuntu, Debian...):

- linux-image # Volatility2, contains the System.map files ("symbols")
- linux-modules # Volatility2, replaces linux-image if available
- linux-headers # Volatility2, specific kernel headers [15]
- linux-headers-common # Volatility2, common kernel headers [15]
- linux-image-dbgsym # Volatility3, compiled kernel with debug symbols

Equivalent packages are needed for DNF based systems (AlmaLinux, Fedora...).

2.7 Build environment

Creating ISF files for Volatility3, once the source package is downloaded, is quite easy, as it only necessites to be extracted and provided to dwarf2json. On the other hand, generating profiles requires a complex setup. Indeed, kernels also relies on GCC to be compiled. At first, I assumed the latest GCC version would do just fine to compile each kernel (starting from 2.6 to recent ones). Unfortunately, I quickly realized a Linux kernel release was generally designed to work with a specific GCC version.

To be as exhaustive as possible, I decided to create an environment with each major GCC version released to this day. This includes : 3.4, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5, 6, 7, 8, 9, 10, 11, 12, 13. Here, the use of Docker was practical, as I was able to create a Dockerfile with all dependencies, not overlapping themselves.

2.8 Workflow

Here is the high view generation workflow :

1. "Scrape" sources of each specified distribution and architecture, for kernel packages.
2. Determine which kernel symbols haven't been generated yet.
3. Instantiate container (repeat for each kernel).
4. Extract kernel source, determine required GCC version by checking kernel requirements, compile kernel and extract profile from container.
5. Extract kernel debug symbols, provide it to dwarf2json and extract ISF from container.

6. Destroy container.

7. Push files to GitHub repositories.

Scraping is done by using package managers to list packages from every release of every distribution. ZIP compression is used for Volatility2 profiles, and Volatility3 ISF JSON files are optimized (no indents, spaces between keys) and XZ compressed.

2.9 macOS symbols generation

Wanting to include macOS symbols in my project too, I also implemented a scraper and automatic builder, working with any format ever provided by Apple for their debug symbols compressed files (KDK) [1].

2.10 Results

As of December 2023 :

- Profiles and ISF for kernels ranging from 2.6.10 to 6.6.0
- Volatility2 profiles collection contains more than 9530 files : https://github.com/Abyss-W4tcher/volatility2-profiles
- Volatility3 symbols collection contains more than 6770 files : https://github.com/Abyss-W4tcher/volatility3-symbols

JSON files matching banners and ISF are also available, allowing end users to quickly search for the right files to get. They can also be used to specify a remote ISF server in the Volatility3 code, to fetch symbols automatically (in the same manner as Windows).

We can observe a huge gap between ISF and profiles count, as a lot of Ubuntu kernels debug symbols aren't available in the ddebs [14] official sources.

2.11 Perspectives

The project sources aren't available for now, because it needs some specific setup and isn't quite useful for end users. However, I plan to release it in the future, when I will have find a way to pack the environment nicely.

3 Rootkit detection plugins

Abstract

A few adjustments to the Volatiltiy3 framework were necessary, and are discussed in the following pull request : #1039. A Volatility3 in version 2.5.2 (latest *develop* branch) will be needed to execute the following plugins.

Following the finalization of the automatic symbols generator project, I started searching out for ideas of new plugins to develop for Volatility3, with the end goal being to compete in the 2023 contest.

During my researches, I came accross a 2021 BlackHat document [4], written by Andrew Case, a well-known digital forensic researcher and core developer of Volatility, and Golden G. Richard III, which is also a well-known digital forensic researcher. The document presents and explains the tracing functionnalities of the Linux kernel, as well as new Volatility plugins to detect their usage. Indeed, many Linux rootkits take advantage of these capabilities to hijack kernel functions and gain persistence in a compromised system. However, after a bit of open source searches, I couldn't get my hands on the developped plugins, as I guess they weren't made publicly available. Correlating this with the fact that the rootkit detection plugins of Volatility contest 2022 were well-receveived, I decided to start the plugins from scratch, and make them publicly available.

Before starting to develop, I needed to setup an analysis environment, to easily inject rootkits inside VMs and extract memory dumps (with *LiME*). To do so, I took advantage of the Vagrant environment creator, and chose a set of open source rootkits :

- *xcellerator Linux Kernel Hacking*, ports hiding and random generators manipulation rootkits [19] : provides an educational guide on rootkit development, as well as many samples and techniques.
- *KoviD* [3] : Linux Kernel 5+ rootkit, with many features : hide itself, hide files and directories, backdoors...
- *bds_lkm_ftrace* [2] : Linux Kernel 5+ rootkit, with many features : hide itself, hide files and directories, backdoors...

I worked on three version of the Linux kernel :

- Debian-bookworm_6.1.0-13-amd64
- Ubuntu-jammy_5.15.0-87-generic
- Ubuntu-xenial_4.4.0-210-generic

First, I started by implementing and updating rootkits provided by xcellerator [19], to make them run on recent Linux kernels. Then, with a good setup in place, I began the development.

3.1 linux.check_ftrace

The ftrace infrastructure was originally created to attach callbacks to the beginning of functions in order to record and trace the flow of the kernel [9]. To enumerate callbacks, the plugin iterates on each *ftrace_ops* structs contained in the *ftrace_ops_list* list. Basically, *ftrace_ops* stores informations about each hooked symbol, and their corresponding callbacks. A callback is an address pointing to a function in memory, which oversteps the original call for a symbol.

Here are the elements returned by the plugin :

- **ftrace_ops** : offset of the corresponding *ftrace_ops* struct
- **Callback** : callback address and its corresponding symbol name (enclosed in brackets)
- **Hooked symbol** : kernel symbol hooked
- **Module** : name of the module holding the callback, and its corresponding offset (enclosed in brackets)
- **Callback out of kernel .text** : determine if the callback is part of the compiled kernel or not
- (optional) **ftrace_ops_flags** : print flags associated to the *ftrace_ops* struct

Now, let's take a look at a sample output from the *check_ftrace* plugin :

```
$ python3 $vol3/vol.py -f Ubuntu-xenial_4.4.0-210-generic_ftrace_xcellerator.lime linux.check_ftrace
Volatility 3 Framework 2.5.2
Progress:   80.00               Scanning ftrace_ops_list...
ftrace_ops      Callback        Hooked symbol   Module  Callback out of kernel .text

0xffffc03b0020  0xffffc03ae0c0 [fh_ftrace_thunk]       ['tcp4_seq_show']       0xffffc03b0100 [rootkit_ports]  True
0xffffc0392100  0xffffc0390200 [fh_ftrace_thunk]       ['urandom_read']        0xffffc03921c0 [rootkit_char]   True
0xffffc0392020  0xffffc0390200 [fh_ftrace_thunk]       ['random_read'] 0xffffc03921c0 [rootkit_char]   True
0xffffc038d020  0xffffc038b050 [fh_ftrace_thunk]       ['SyS_kill', 'sys_kill']        0xffffc038d100 [rootkit_hidden] True
```

Figure 1: *check_ftrace* : *xcellerator rootkits* sample output

Memory image was infected with three rootkits from [19], each of them hooking one or many symbols from the Linux kernel, using ftrace. As we can see, the plugin successfully detect this behaviour, and is able to provide the user with useful informations and objects addresses for further investigations (e.g. with volshell). One of the rootkit is named *rootkit_hidden*, because it unlinks itself from the *list_head* linked list. However, an existing Volatility3 plugin named *check_modules* is able to obtain the list of modules from the *sysfs* list and compare it to the output of the *lsmod* plugin. The *check_ftrace* plugin directly calls these two plugins to insure efficient modules discovery.

What happens if a rootkit uses a malicious technique to hide itself both from lsmod and sysfs ? Let's take a look at another rootkit, *KoviD* [3] :

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-jammy_5.15.0-87-generic_kovid.lime linux.check_ftrace
Volatility 3 Framework 2.5.2
Formatting...4.12               Scanning ftrace_ops_list...
  |      ftrace_ops |              Callback |             Hooked symbol |  Module | Callback out of kernel .text
* | 0xffffc09ed0d0 | 0xffffc09e5630 [UNKNOWN] |              ['tty_read'] | UNKNOWN |                          True
* | 0xffffc09ecff8 | 0xffffc09e5630 [UNKNOWN] |             ['filldir64'] | UNKNOWN |                          True
* | 0xffffc09ecf20 | 0xffffc09e5630 [UNKNOWN] |               ['filldir'] | UNKNOWN |                          True
* | 0xffffc09ece48 | 0xffffc09e5630 [UNKNOWN] |      ['audit_log_start'] | UNKNOWN |                          True
* | 0xffffc09ecd70 | 0xffffc09e5630 [UNKNOWN] |  ['account_system_time'] | UNKNOWN |                          True
* | 0xffffc09ecc98 | 0xffffc09e5630 [UNKNOWN] | ['account_process_tick'] | UNKNOWN |                          True
* | 0xffffc09ecbc0 | 0xffffc09e5630 [UNKNOWN] |          ['tpacket_rcv'] | UNKNOWN |                          True
* | 0xffffc09ecae8 | 0xffffc09e5630 [UNKNOWN] |           ['packet_rcv'] | UNKNOWN |                          True
* | 0xffffc09eca10 | 0xffffc09e5630 [UNKNOWN] |        ['udp6_seq_show'] | UNKNOWN |                          True
* | 0xffffc09ec938 | 0xffffc09e5630 [UNKNOWN] |        ['tcp6_seq_show'] | UNKNOWN |                          True
* | 0xffffc09ec860 | 0xffffc09e5630 [UNKNOWN] |        ['udp4_seq_show'] | UNKNOWN |                          True
* | 0xffffc09ec788 | 0xffffc09e5630 [UNKNOWN] |        ['tcp4_seq_show'] | UNKNOWN |                          True
* | 0xffffc09ec6b0 | 0xffffc09e5630 [UNKNOWN] |        ['__x64_sys_bpf'] | UNKNOWN |                          True
* | 0xffffc09ec5d8 | 0xffffc09e5630 [UNKNOWN] |       ['__x64_sys_kill'] | UNKNOWN |                          True
* | 0xffffc09ec500 | 0xffffc09e5630 [UNKNOWN] |      ['__x64_sys_clone'] | UNKNOWN |                          True
* | 0xffffc09ec428 | 0xffffc09e5630 [UNKNOWN] | ['__x64_sys_exit_group'] | UNKNOWN |                          True
```

Figure 2: *check_ftrace* : *kovid* sample output (hidden)

As this rootkit implements advanced hiding techniques, it also unlinks itself from the sysfs linked list, while still keeping the callbacks in place. The plugin is not able to detect the module's name, and the callback symbol.

To counter this technique and uncover this behaviour, I developed another plugin which scans memory for structures artifacts.

3.2 linux.check_unlinked_modules

Uncovering modules unlinked from the kernel lists requires to manually scan the memory for artifacts. Each loaded module is assigned a memory range in kernel space [13], typically placed one after the other. Let's assume we have three modules loaded, in order : A, B and C. If B unlinks itself from /proc/modules and /sys/module/, here is what happens :

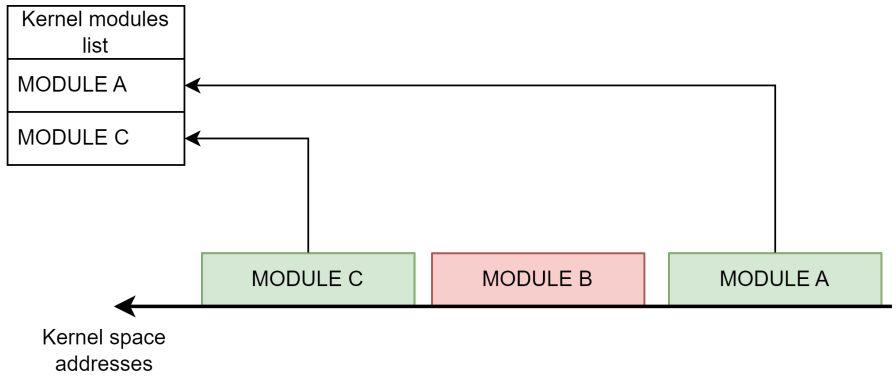

Figure 3: Sample memory modules layout

How can Volatility detect its presence now ? Well, the *module* struct can be identified in memory by looking for specific properties :

```
struct module
{
        enum module_state state;

        /* Member of list of modules */
        struct list_head list;

        /* Unique handle for this module */
        char name[MODULE_NAME_LEN];

        /* Sysfs stuff. */
        struct module_kobject mkobj;
        struct module_attribute *modinfo_attrs;
        const char *version;
        const char *srcversion;

    /* redacted /*
}
```

My first approach, was to create a regular expression matching specific properties of this structure :

```
# Split regex for clarity.
regex_parts =
    {
        "r_enum": b"[\x00-\x03]\x00{7}",
        "r_list_next": b".{8}", # pointer can be NULL
        "r_list_prev": b".{8}", # pointer can be NULL
        "r_name": b".{56}",
        "r_mkobj": b"(?!\x00{8}).{8}",  # valid pointer
    }
# Use a lookahead assertion to allow "overlapping" matches
module_regex = (
    b"(?=" + b"".join([part for part in module_regex_parts.
        values()]) + b")"
)
```

This worked well for my samples, but it wasn't taking into account endianness or x86/x64 formats. Furthermore, the "module" struct was not organized the same way from one kernel to another.

Instead of doing case by case regexes, I decided to create a function generating a regex based on an *ObjectTemplate* and its member :

- iterate over each element
- check if we can make assumption against the element type (e.g. a boolean will be 0 or 1)
- walk recursively in sub-elements (a struct inside a struct), to create sub-regexes

This allows to create a "regex mask" against a struct, flexible and strong against any kernel version. I also took into account potential overrides, if a structure has some pre-defined properties values that can help identifying it. The function design is not targeted against the *module* struct, thereby analysts should be able to use it to scrape memory for any other struct, as long as it exists in the symbol table. It was also tested against an arbitrary sample from the Volatility foundation [6], and by enabling the debug output, the plugin succesfully detects every module struct in memory. After getting a list of offsets from the regex scan, the plugin validates a candidate if its *module→mkobj→mod* points to the candidate offset (itself).

Side note : Some rootkits, like *osom* [12], tend to NULL their module name after hiding :

```
/* Hide the module information from the kernel. Partially
    unloads it. */
void hide_module(struct module *mod){
     list_del(&mod->list);
     kobject_del(&mod->mkobj.kobj);
     mod->sect_attrs = NULL;
     mod->notes_attrs = NULL;
     memset(mod->name, 0, 60);
     return;
}
```

Current Volatility3 modules utilities convert the *module→name* property to a plain string. It returns only printable characters from the module name, which can be misleading (e.g *module→name = 0xdeadbeef*). As a result, the name returned by Volatility3 can be an empty string, being a real problem when two modules have non printable names, as it won't be possible to clearly differentiate them.

When a malicious module hides itself, we still have a leak of an address inside its memory space (e.g. the callback offset from the *check_ftrace* plugin). We can determine the closest modules from this leak (backward and forward), and scan all the memory between them, for potential modules struct. Let's uncover the *kovid* module, based on a known callback from the previous plugin output :

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-jammy_5.15.0-87-generic_kovid.lime -v linux.check_unlinked_modules --leaked-address 0xffffc09e5630
Volatility 3 Framework 2.5.2
INFO     volatility3.plugins.linux.check_unlinked_modules: Searching modules structs from 0xffffc090c000 to 0xffffc09f3000, based on provided address 0xffffc09e5630...
INFO     volatility3.plugins.linux.check_unlinked_modules: Found sysfs non-listed module kovid at 0xffffc09ed4c0
Formatting...
  | Module Address | Module Name
* | 0xffffc09ed4c0 |       kovid
```

Figure 4: *kovid* rootkit module uncovered by *check_unlinked_modules*

It is also possible to bruteforce the modules memory range directly, without a leak, as the *–leaked-address* parameter is optional. By implementing a call to this plugin from the *check_ftrace* plugin, it can now have precise informations about deeply hidden modules too :

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-jammy_5.15.0-87-generic_kovid.lime linux.check_ftrace
Volatility 3 Framework 2.5.2
Formatting...4.12               Scanning ftrace_ops_list...
  |     ftrace_ops |                          Callback |            Hooked symbol |                  Module
* | 0xffffc09ed0d0 | 0xffffc09e5630 [fh_ftrace_thunk] |             ['tty_read'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecff8 | 0xffffc09e5630 [fh_ftrace_thunk] |            ['filldir64'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecf20 | 0xffffc09e5630 [fh_ftrace_thunk] |              ['filldir'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ece48 | 0xffffc09e5630 [fh_ftrace_thunk] |      ['audit_log_start'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecd70 | 0xffffc09e5630 [fh_ftrace_thunk] |  ['account_system_time'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecc98 | 0xffffc09e5630 [fh_ftrace_thunk] | ['account_process_tick'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecbc0 | 0xffffc09e5630 [fh_ftrace_thunk] |          ['tpacket_rcv'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ecae8 | 0xffffc09e5630 [fh_ftrace_thunk] |           ['packet_rcv'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09eca10 | 0xffffc09e5630 [fh_ftrace_thunk] |        ['udp6_seq_show'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec938 | 0xffffc09e5630 [fh_ftrace_thunk] |        ['tcp6_seq_show'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec860 | 0xffffc09e5630 [fh_ftrace_thunk] |        ['udp4_seq_show'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec788 | 0xffffc09e5630 [fh_ftrace_thunk] |        ['tcp4_seq_show'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec6b0 | 0xffffc09e5630 [fh_ftrace_thunk] |        ['__x64_sys_bpf'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec5d8 | 0xffffc09e5630 [fh_ftrace_thunk] |       ['__x64_sys_kill'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec500 | 0xffffc09e5630 [fh_ftrace_thunk] |      ['__x64_sys_clone'] | 0xffffc09ed4c0 [kovid]
* | 0xffffc09ec428 | 0xffffc09e5630 [fh_ftrace_thunk] | ['__x64_sys_exit_group'] | 0xffffc09ed4c0 [kovid]
```

Figure 5: *check_ftrace* : *kovid* rootkit sample output (uncovered)

Here is another sample output, from the *bds_lkm_ftrace* [2] infected memory dump (some DEBUG output is stripped):

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-jammy_5.15.0-87-generic_bds_lkm.lime -v linux.check_ftrace
Volatility 3 Framework 2.5.2
INFO    volatility3.plugins.linux.check_unlinked_modules: Searching modules structs from 0xffffc08bb000 to 0xffffc0995000, based on provided address 0xffffc098e270...
INFO    volatility3.plugins.linux.check_unlinked_modules: Found sysfs non-listed module bds_lkm_ftrace at 0xffffc0990500
INFO    volatility3.plugins.linux.check_ftrace: No func_hash.filter_hash.buckets for ftrace_ops@0xffff8463dd80, skipping...
Formatting...
  |     ftrace_ops |                        Callback |          Hooked symbol |                    Module | Callback out of kernel .text
* | 0xffffc0990380 | 0xffffc098e270 [fh_ftrace_thunk] |       ['__x64_sys_kill'] | 0xffffc0990500 [bds_lkm_ftrace] |                          True
* | 0xffffc09902b0 | 0xffffc098e270 [fh_ftrace_thunk] |   ['__x64_sys_getdents'] | 0xffffc0990500 [bds_lkm_ftrace] |                          True
* | 0xffffc09901e0 | 0xffffc098e270 [fh_ftrace_thunk] | ['__x64_sys_getdents64'] | 0xffffc0990500 [bds_lkm_ftrace] |                          True
* | 0xffffc0990110 | 0xffffc098e270 [fh_ftrace_thunk] |        ['tcp6_seq_show'] | 0xffffc0990500 [bds_lkm_ftrace] |                          True
* | 0xffffc0990040 | 0xffffc098e270 [fh_ftrace_thunk] |        ['tcp4_seq_show'] | 0xffffc0990500 [bds_lkm_ftrace] |                          True
```

Figure 6: *check_ftrace* : *bds_lkm* rootkit sample output (uncovered)

3.3 linux.check_tracepoints

A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime. A tracepoint can be "on" (a probe is connected to it) or "off" (no probe is attached) [10]. Any tracepoint is held inside an array of pointers, ranging from *__start___tracepoints_ptrs* to *__stop___tracepoints_ptrs* symbols offsets. Pointers in this array can be specified in two different formats, depending on kernel :

- Absolute pointer to a *tracepoint* struct (*CONFIG_HAVE_ARCH_PREL32_RELOCATIONS* is False)
- Offset relative pointer, represented by a 32 bits integer (*CONFIG_HAVE_ARCH_PREL32_RELOCATIONS* is True)

The plugin first starts by detecting which format is used, and then iterates on each *tracepoint*. It only processes tracepoints which have at least one probe attached("on"), as it would result in many non-forensic use results [4].

Here are the elements returned by the plugin :

- **tracepoint** : offset of the corresponding *tracepoint* struct
- **Probe** : probe address and its corresponding symbol name (enclosed in brackets)
- **Module** : name of the module holding the probe, and its corresponding offset (enclosed in brackets)
- **Probe out of kernel .text** : determine if the probe is part of the compiled kernel or not

To test the plugin, I used two versions of the same module, developed by Hugo GUIROUX, available here. First one was adapted for recent kernels, attaching a probe to *mm_page_free* and *mm_page_alloc* [4]. Second is the original module, which was targeted for older kernels, attaching a probe to *sched_switch* and *sched_wakeup*.

Let's take a look at the plugin output for a 5.15.0 kernel memory sample :

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-jammy_5.15.0-87-generic_tracepoints.lime -v linux.check_tracepoints
Volatility 3 Framework 2.5.2
INFO     volatility3.plugins.linux.check_tracepoints: CONFIG_HAVE_ARCH_PREL32_RELOCATIONS was determined to be True
Formatting...9.88            Iterating over tracepoints...
  |                tracepoint |                              Probe |                      Module | Probe out of kernel .text
* |  0xffffa7df8820 [mm_page_free] |  0xffffc08a0010 [probe_mm_page_free] | 0xffffc08a20c0 [tracepoint_mod] |                      True
* | 0xffffa7df8760 [mm_page_alloc] | 0xffffc08a0000 [probe_mm_page_alloc] | 0xffffc08a20c0 [tracepoint_mod] |                      True
```

Figure 7: *check_tracepoints* : *mm_page_free* and *mm_page_alloc* probes

The plugin was successfully able to detect, firstly, the format of the "*tracepoint*" pointers array, and secondly the ones with an attached probe. Module detection works the same as for the *check_ftrace* plugin.

Finally, we can also confirm successful detection on a 4.4.0 kernel memory sample :

```
$ python3 $vol3/vol.py -r pretty -f Ubuntu-xenial_4.4.0-210-generic_tracepoints.lime  -v linux.check_tracepoints
Volatility 3 Framework 2.5.2
INFO    volatility3.plugins.linux.check_tracepoints: CONFIG_HAVE_ARCH_PREL32_RELOCATIONS was determined to be False
Formatting...9.80               Iterating over tracepoints...
  |                  tracepoint |                                Probe |                          Module | Probe out of kernel .text
* | 0xffff81f17520 [sched_switch] | 0xffffc0386000 [probe_sched_switch] | 0xffffc0388040 [tracepoint_mod] |                      True
* | 0xffff81f175a0 [sched_wakeup] | 0xffffc0386010 [probe_sched_wakeup] | 0xffffc0388040 [tracepoint_mod] |                      True
```

Figure 8: *check_tracepoints* : *sched_switch* and *sched_wakeup* probes

4 Why should I win the 2023 Volatility contest ?

Memory forensics is an exciting and complex computer science field, which I enjoy studying and working in. It offers an interesting view about the preciousness of volatile data, even the ones seemingly uninteresting. Progresses and discoveries rely on the basis of open source and shared researches, across experts and non-experts around the world. Getting involved in Volatility, and contributing to the framework, is an incredible journey. Outside the technical aspect, which is often highly specialized, I appreciate exchanging and learning from peers the most, especially in this community.

Over a span of five months, I engaged myself in the development of various tools and ressources. Firstly, I was able to construct an automated and robust kernel symbols scraper and builder, which generated files have proven to be very useful for end users. It also spared me time, and provided me flexibility, as I was able to create and analyze memory samples by using the already generated symbols of my collection. Then, I started getting more familiar with the Volatility3 codebase, looking forward to understand it more and interface with it. Once I had an acceptable understanding of the logic, I began to search out for ideas of useful plugins to develop. After a few attempts and ideas, I got my hand on an interesting research paper about kernel tracing and its malicious use. Even if plugins to detect this behaviour were developed, they weren't made publicly available. Thereby, I thought it would be a great project to implement them for the open source community. As I do not wanted my work to be a pale copy (even though the research document only provided the general steps, not the code), I incorporated my own creativity and ideas to it.

To provide a work of quality over quantitiy, I worked only on two of the techniques mentioned : *ftrace* and *tracepoints* Linux kernel hooking detection. Moreover, I created an additional plugin to detect hidden kernel modules in memory. When working on these, I insisted on making them modular and adapted to the most situations possible. At the end of the day, the plugins were successfuly able to detect malicious behaviour on a set of Linux rootkits, as detailed in this document.

Winning the 2023 Volatility contest would be an incredible event for me, as all these hours working to help the community would be directly rewarded by memory forensics experts and pioneers. This would really nicely add up to the "Thank you" I can receive, following the use of resources I created.

References

[1] Apple. *macOS KDK*. URL: https://developer.apple.com/download/all/?q=kernel%20debug%20kit.

[2] bluedragonsecurity. *BDS LKM FTRACE ROOTKIT*. URL: https://github.com/bluedragonsecurity/bds_lkm_ftrace.

[3] carloslack. *KoviD LKM*. URL: https://github.com/carloslack/KoviD.

[4] Andrew Case and Golden G. Richard III. "Fixing a Memory Forensics Blind Spot: Linux Kernel Tracing". In: 2021. URL: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Fixing-A-Memory-Forensics-Blind-Spot-Linux-Kernel-Tracing-wp.pdf.

[5] Volatility Foundation. *dwarf2json*. URL: https://github.com/volatilityfoundation/dwarf2json.

[6] Volatility Foundation. *linux-sample-1.bin.gz*. URL: https://downloads.volatilityfoundation.org/volatility3/images/linux-sample-1.bin.gz.

[7] Volatility Foundation. *Volatility Foundation*. URL: https://www.volatilityfoundation.org.

[8] JPCERTCC. *Symbol tables for Windows*. URL: https://blogs.jpcert.or.jp/en/2021/09/volatility3_offline.html.

[9] kernel.org. *Using ftrace to hook to functions*. URL: https://www.kernel.org/doc/html/latest/trace/ftrace-uses.html.

[10] kernel.org. *Using the Linux Kernel Tracepoints*. URL: https://docs.kernel.org/trace/tracepoints.html.

[11] Michael Hale Ligh, Andrew Case, Jamie Levy, and AAron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014. ISBN: 1118825098.

[12] NinnOgTonic. *Out-of-Sight-Out-of-Mind-Rootkit*. URL: https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit.

[13] Oracle. *Differences Between Kernel Modules and User Programs*. URL: https://docs.oracle.com/cd/E19253-01/817-5789/emjjr/index.html.

[14] Ubuntu. *Debug Symbol Packages*. URL: https://wiki.ubuntu.com/Debug%20Symbol%20Packages.

[15] gentoo wiki. *Linux-headers*. URL: https://wiki.gentoo.org/wiki/Linux-headers.

[16] Wikipedia. *Debug symbol*. URL: https://en.wikipedia.org/wiki/Debug_symbol.

[17] Wikipedia. *Linux distributions*. URL: https://en.wikipedia.org/wiki/List_of_Linux_distributions.

[18] Wikipedia. *vmlinux*. URL: https://en.wikipedia.org/wiki/Vmlinux.

[19] xcellerator. *Linux Kernel Hacking*. URL: https://github.com/xcellerator/linux_kernel_hacking.

List of Figures