Fixing a Memory Forensics Blind Spot: Linux Kernel Tracing

Andrew Case and Golden G. Richard III

andrew@dfir.org, golden@cct.lsu.edu

July 16, 2021

1 Introduction

Servers running Linux power the majority of important websites [1] and cloud instances [2], and are routinely used for internal services across government, military, and industry. The data and access provided by these servers make them a prime target for attackers, and, as such, makes research into detection of threats against these platforms essential. Unfortunately, defensive research has not kept pace with advances in Linux kernel development, leaving blind spots for attackers to remain undetected. In this paper, we document our research effort to close a significant blind spot - the Linux kernel's tracing infrastructure. As shown later, these features are installed and enabled by default on essentially all Linux distributions. These features, particularly eBPF, are also heavily utilized across a significant number of cloud-centric organizations, such as Facebook, Netflix, Google, GitLab, and Adobe [3, 4, 5].

These tracing features provide developers and systems administrators significant insight into the performance and behaviour of applications and system components, as the tracing APIs allow programs in userland as well as modules in the kernel to observe and modify key portions of the operating system. Unfortunately, these APIs are also ripe for abuse by malware as they provide direct support for hooking kernel subsystems and hardware features, such as the networking stack, system call table, and file system drivers. They also allow placing traditional API hooks at arbitrary places within the kernel, as well as disrupting the control flow of kernel functions.

Current memory forensics techniques provide no means to analyze these tracing features, leaving a significant number of malware capabilities to potentially go undetected. In this paper, we document the internal data structures and algorithms of these tracing features along with new memory forensic techniques that can analyze the various tracing subsystems. These new analysis techniques are embodied in Volatility plugins, as Volatility is the mostly commonly used analysis framework in the field. To provide capabilities that are useful both now and well into the future, we implemented each technique as a plugin for both Volatility 2 and Volatility 3.

Volatility 2 is currently the stable version used in investigations throughout the world, whereas Volatility 3 is the next major version of the framework and will eventually replace Volatility 2. Beyond providing long term value, developing for both versions allowed us to create modern examples of what the API and plugin-level differences are between the major versions. It is our hope that these will be useful examples for researchers learning to develop Volatility plugins. Our team plans to contribute all the new plugins to the public Volatility repositories upon publication of this paper.

2 Kernel Tracing Infrastructure

The tracing infrastructure in the kernel is comprised of a number of different components, many of which are interconnected. At the lowest levels of are the *ftrace*, *kprobes*, and *tracepoints* facilities. These allow direct hooking of kernel functions, on both entry and exit, as well as modification of arbitrary kernel data and code. Until kernel version 4.19, the *jprobes* facility was also available, but given its removal 3 years ago, we do not discuss it further in this paper.

The highly popular *perf_events* and *eBPF* subsystems leverage these low-level facilities to provide the insight needed to monitor system operations. *XDP* is built on top of *eBPF* and allows both monitoring and manipulation of high performance networking operations. XDP allows programs to inspect, modify, redirect, and drop network packets before other software on the system examines them. Example programs that accomplish this can be found at [6, 7].

Section 4 fully explores these tracers, including a discussion of how they are implemented, how userland and the kernel interface with them, and how memory forensics algorithms can be developed to detect when they are used. Readers who are completely new to the kernel tracing infrastructure may wish to read the introduction written by Julia Evans [8]. There is also substantial kernel documentation for many of the features [9]. The Awesome eBPF page also tracks a large number of presentations, code projects, and examples for eBPF [10].

2.1 Prevalence of the Tracing Facilities

To begin our research of the tracing infrastructure, we first measured the percentage of kernels which have the required kernel options present and enabled. This has an obvious and significant effect on the actual, real-world threat posed by these features. To perform the calculations, we analyzed all Linux kernels that have been shipped by major distributions within the last few years. Engineers at Volexity maintain this database and provide an API that allows querying the configuration options present in each kernel.

The kernels covered by this database include those from:

- Debian 6+
- Ubuntu 14.04+
- Red Hat 6+
- CentOS 6+
- Suse (SLES) 11, 12, and 15
- Amazon (AWS) Linux

At the time of writing and calculation of the statistics listed in this section, the database held a total of 14,762 distribution-released kernels.

The list of kernel requirements necessary to support all the tracing features was gathered from the install documentation of *bpftrace* [11]. As discussed in Section 4.6, this suite of tools uses eBPF to monitor a wide range of system activity. The documentation stated that only kernel versions 4.9 and higher should be used as this is when all the needed features were added. This series of kernels was released in 2016, so limiting our study to these versions and newer is not unrealistic when aiming to match versions of production systems.

The kernel configuration options listed as required in the install documentation are:

- CONFIG_BPF
- CONFIG_BPF_SYSCALL
- CONFIG_BPF_JIT
- CONFIG_HAVE_EBPF_JIT
- CONFIG_ARCH_SUPPORTS_UPROBES
- CONFIG_FTRACE_SYSCALLS
- CONFIG_FUNCTION_TRACER
- CONFIG_HAVE_DYNAMIC_FTRACE
- CONFIG_DYNAMIC_FTRACE
- CONFIG_HAVE_KPROBES
- CONFIG_KPROBES
- CONFIG_KPROBE_EVENTS
- CONFIG_BPF_EVENTS
- CONFIG_UPROBES
- CONFIG_UPROBE_EVENTS
- CONFIG_DEBUG_FS

After filtering to only kernel versions 4.9 and greater, we were left with 5,386 kernels. We then calculated the percentage of these that had all of the previously listed options present. We discovered that 4,468 kernels (82.9%) had all options present. We then investigated the kernels that were missing at least one option to determine which option(s) were missing and if we could learn why they were missing. This revealed that kernels versioned 4.9.x and 4.10.x were missing CONFIG_UPROBE_EVENTS and CONFIG_KPROBE_EVENTS. Consulting the Linux Kernel Driver Database, we learned that these configuration options were not actually added until kernel version 4.11 [12, 13]. We then modified our statistics script to filter out checking for these two options in kernel versions 4.9.x and 4.10.x. This new calculation method showed that 5,191 of the kernels (96.3%) supported all configuration options needed.

Of the 195 kernels that were still missing at least one option, all were Ubuntu-packaged kernels and all of them were *kvm* variants. These variants are stripped down kernels meant to provide only what is needed to run as a KVM virtual machine guest [14]. This is obviously not a configuration found in most enterprises or cloud providers.

As our calculations show, the kernel tracing infrastructure is enabled on essentially all modern Linux distributions kernels. This means the majority of production systems have the tracing features enabled, making them a necessary target of defensive research.

2.2 Malicious Uses

The power provided by the tracing infrastructure has led to several projects that leverage these features for malicious purposes. The potential for abuse of kprobes is documented in Phrack issue 67 from 2010 [15]. The ebpf project from NCC Group contains a number of utilities showing how a variety of abuses can be achieved [16]. The *conjob* utility intercepts and modifies reads to */etc/crontab*. *obie-trice-conjob* achieves the same goal, but uses raw tracepoints as a method to bypass AppArmor. *glibcpwn* injects a shared library into systemd using eBPF-based kprobes. *unixdump* saves all data sent through UNIX domain sockets to disk. *uprobe-ulose* allows setting arbitrary uprobes in running processes. Besides the source code for these utilities, the referenced GitHub project also has several PDFs from presentations on the tool suite.

These examples showcase just some of the immense power that abuse of the tracing facilities gives to malware authors. In Section 4, we discuss even more malicious examples as we detect them with our newly developed Volatility plugins.

3 Current Memory Analysis Capabilities

Unfortunately, existing memory forensic techniques do not detect the artifacts generated by the tracing facilities. Current techniques are able to find modifications to the system call table, but only under two conditions. The first is when indexes in the table are changed to point to malicious handlers, and the second is when the reference to the system call table address is changed in the system call handling function. The tracing facilities we discuss in this paper use neither of these approaches to hook functions.

Similarly, existing techniques attempt to detect when function prologues (beginning instructions) are hooked, but these techniques are not useful when the kernel's tracing facilities place a hook. These facilities use one of two methods when placing API hooks. The first writes a debug trap instruction (int3) at the beginning of the target function to trigger a debug trap upon future calls. Existing analysis techniques do not check for int3 instructions, but even if they were updated to do so, the results would offer nothing in terms of knowledge about which module is handling the hook and why the hook is present. All that would be available to be reported is that a particular function has been hooked by a tracing facility. The second hooking method places a CALL instruction to a special kernel code buffer that then determines which handler to call. This again means that traditional API hook techniques would only be able to report that a function is hooked, but nothing about the module handling the hook.

As shown in Section 4, by performing structured analysis of the tracing facilities, we can determine not only which functions are hooked, but also provide information about the hook's handler. This allows for deeper analysis and extraction of the malicious code region(s) and hosting kernel module, assuming the module is still present in kernel memory.

4 Detecting Tracers

In this section, we document our research effort against each active tracer in mainline kernels. For each tracer, we document how it works, describe the Volatility plugins we created to detect abuse of the tracer, and show output of our plugins after they are executed against memory samples from computers in which malicious modules were using the tracing facilities.

4.1 Test Environment

For our test environment we used several Linux virtual machine guests. These included a Debian 10 system running kernel version 4.19, an Ubuntu 18.04 system running kernel version 4.15, an Ubuntu 20.04 system running kernel version 5.4, and a Kali Linux system running kernel version 5.10.

To acquire memory, Surge Collect Pro from Volexity as well as virtual machine snapshots were used. Both of these acquire memory in a stable and quick manner and allowed us to rapidly gather samples with particular tracers active.

4.2 ftrace

4.2.1 Kernel Implementation

ftrace is nicely documented in the kernel documentation tree [17]. To register a callback with ftrace, a kernel module has to first define a *ftrace_ops* structure whose *func* member points to the callback handler. The *ftrace_set_filter* function can be then be used to associate the ftrace_ops structure with one or more functions based on their name. The *ftrace_set_filter_ip* function can also be used to set the filter on a specific address. Finally, *register_ftrace_function* is used to activate the filter within the kernel.

The registration function adds the operations structure to the global *ftrace_ops_list* data structure. This list holds all of the system's active operations structures. To associate a callback with a particular hooked function, a set of embedded data structures and hash tables are used. To start, the *func_hash* member of type *ftrace_ops_hash* references the hash table of filters for the particular operations structure. The elements of this hash table are of type *ftrace_func_entry* and store the address being hooked in the *ip* member. The use of these data structures allows the kernel to efficiently map a hooked address (function start) to its registered ftrace callback(s).

4.2.2 Volatility Plugin Implementation

To enumerate the ftrace callbacks present in a memory sample, we developed the *linux_ftrace* plugin. This plugin begins by locating and enumerating *ftrace_ops_list*. This is accomplished using existing Volatility APIs. For each operation element, the plugin then walks the hash table of registered callbacks, if any. For each callback, the plugin gathers the data structure address, callback handler address, name of the module that registered the callback, the symbol name of the callback, and the name of the function hooked.

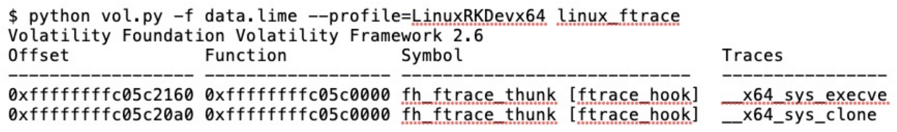
The address of each callback handler is discovered via its *func* member. The module hosting the callback is found through first enumerating the kernel code ranges and the ranges of each kernel module in the active module list, and, then determining which, if any, the callback fits within. If a module is found, then the symbol address is searched within the module's symbol table using standard ELF parsing techniques. If a module is not found, then it means the callback is within a non-file backed region or within a hidden kernel module - both of which standout in the plugin output. The name of the hooked function is gathered by mapping the *ip* member of each *ftrace_func_entry* to its symbol name using the same method as for *func*.

The end result of *linux_ftrace* is a complete listing of all registered ftrace callbacks, including the address and symbol name of all symbols involved. This can be used to immediately inform the investigator of any malicious callbacks.

4.2.3 Detection Example

To illustrate the usefulness of our plugin, we analyzed a POC-rootkit that abuses ftrace [18]. The full source code to the rootkit can be found on GitHub at the link in the bibliography. This rootkit operates by using ftrace to hook the clone and execve system calls. In its implementation, both hooked system calls are redirected to the same callback, named *fh_ftrace_thunk*.

Figure 1 shows the output of *linux_ftrace* against a memory sample with the POC rootkit loaded. In the output of the plugin, the address of the callback function is listed in the second column (*Function*) along with the symbol name (*fh_ftrace_thunk*) and kernel module name (*ftrace_hook*) in the Symbol column. The Traces values correspond to the functions that are hooked, which are the handlers for the execve and clone system calls, as expected.



```
$ python vol.py -f data.lime --profile=LinuxRKDevx64 linux_ftrace
Volatility Foundation Volatility Framework 2.6
Offset             Function           Symbol                          Traces
------------------ ------------------ ------------------------------  ----------------
0xffffffffc05c2160 0xffffffffc05c0000 fh_ftrace_thunk [ftrace_hook]   __x64_sys_execve
0xffffffffc05c20a0 0xffffffffc05c0000 fh_ftrace_thunk [ftrace_hook]   __x64_sys_clone
```

Figure 1: Output of linux_ftrace

4.3 tracepoints

4.3.1 Kernel Implementation

The tracepoints feature allows hooking functions that define tracepoint entries at compile time, and, in modern kernels, a significant number of functions define these. Illustrating this is that our test systems had thousands of functions available as tracepoint targets. Each tracepoint is tracked by a *tracepoint* structure, and the set of tracepoints are referenced in a compile-time built array of pointers to these structures. This array is placed in a boundary defined by the *__start___tracepoints_ptrs* and __stop___tracepoints_ptrs global variables.

By default, each tracepoint has no callbacks, referred to as *probes*, attached. To register a probe for a particular tracepoint, a kernel module must call *tracepoint_probe_register*. This registration API takes two parameters, the first being a pointer to the tracepoint to attach the probe and the second being the function pointer to the probe handler. The kernel provides the *for_each_kernel_tracepoint* macro that enumerates all of the tracepoint instances. This macro simply walks the array stored between *__start___tracepoints_ptrs* and *__stop___tracepoints_ptrs*.

To associate the probe with a particular tracepoint, the registration function builds a dynamically sized array of *tracepoint_func* structures, which are referenced from the tracepoint's *funcs* member. Each probe corresponds to one entry in this array, and the *func* member points to the probe handler.

After this registration process is finished, all calls to the probed function will first be redirected to the registered probe handler.

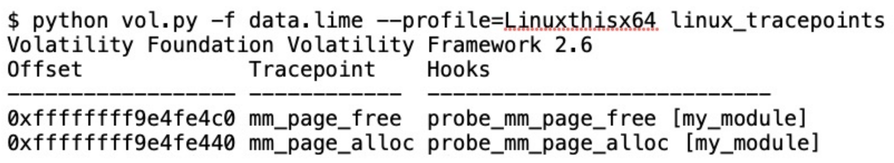
4.3.2 Volatility Plugin Implementation

We developed a new Volatility plugin, *linux_tracepoints*, that is able to enumerate all registered tracepoint probe handlers. These handlers are then mapped back to their owning module and symbol, if present. The plugin begins by enumerating each tracepoint in the same manner as the *for_each_kernel_tracepoint* macro. The plugin then walks each array of probe handlers referenced by a tracepoint's *funcs* member. The *func* member is then used to extract the probe handler's address as well as map it back to a kernel module and symbol. We note that the plugin only lists tracepoints that have at least one probe attached, otherwise thousands of lines with no forensic use would be displayed on each plugin run.

4.3.3 Detection Example

To demonstrate this plugin, we used an open source example [19]. This example registers probe handlers for the *sched_switch* and *sched_wakeup* functions. These were not present in our tested kernels though (the POC code is from 2016), so we changed the code to instead probe *mm_page_alloc* and *mm_page_free*.

Figure 2 shows the output of *linux_tracepoints* against the memory sample with the modified POC active. As can be seen, the plugin correctly determines the two functions that are being probed and successfully maps them back to their handler symbol inside of the *my_module* POC kernel module.



```
$ python vol.py -f data.lime --profile=Linuxthisx64 linux_tracepoints
Volatility Foundation Volatility Framework 2.6
Offset             Tracepoint    Hooks
------------------ ------------  ----------------------------
0xffffffff9e4fe4c0 mm_page_free  probe_mm_page_free [my_module]
0xffffffff9e4fe440 mm_page_alloc probe_mm_page_alloc [my_module]
```

Figure 2: Output of linux_tracepoints

4.4 kprobes/kretprobes

4.4.1 Kernel Implementation

kprobes allows function hooking by kernel modules. The API for kernel modules to register a kprobe is *register_kprobe*. This function takes a *kprobe* structure as a parameter and activates the kprobe on the system. On modern kernels, this structure provides two fields that can be used to specify the target of the kprobe: an *addr* field and a *symbol_name* field. Only one of these may be used and specifying values for both *symbol_name* and *addr* will result in an error. The *symbol_name* field was introduced in 2.6.13; prior to this, only the *addr* field was available. If the *addr* field is used to specify the function that should be probed, it should be populated with the kernel function's address. The *symbol_name* provides a more streamlined approach, since the kprobe author can specify a string identifying the function to probe and this string is automatically resolved to a kernel address via *kallsyms_lookup_name*. The *kprobes* structure also contains function pointers for a pre-handler, post-handler, and fault handler. The pre-handler runs before the first instruction in the probed function is executed and the post-handler runs after the first instruction has executed. The fault handler is used to handle faults within the other two handlers or when kprobes single steps inside a handler.

To ensure a function is hooked, the kprobes subsystem places a software breakpoint instruction (e.g., an int3 on x86/x86-64) at the function's first byte. This forces a debug trap whenever the function is called. kprobes will then activate any registered hook handlers, as appropriate.

The set of active kprobes is stored in the *kprobe_table* hash table. Each element of the table represents one active kprobe and is of type *kprobe*. Inside each kprobe are references to two lists. The first stores all of the kprobes that have hooked the same function, and the second is the current structure's position within the hash table bucket. The structure also contains the name of the symbol it has hooked as well as the offset into the function. The function pointers for the pre-, post- and fault handlers are also stored in this structure.

Importantly, the post-handler for registered kprobes does *not* execute after the function has completed, but rather after the first instruction in the function has executed. To access or modify the return address requires a kprobe variant called a kretprobe, which are registered using the function *register_kretprobe()*, which takes a *kretprobe* structure that contains an embedded *kprobe* structure. Two handlers are specified in the *kretprobe* structure, a handler that is invoked when the function completes (which offers a chance to read or modify the return value of the probed function) and a kprobes pre-handler, invoked on function invocation. The kretprobes mechanism is implemented using kprobes, with a pre-handler saving and then modifying the return address of the function so that the exit handler can gain control as the function returns.

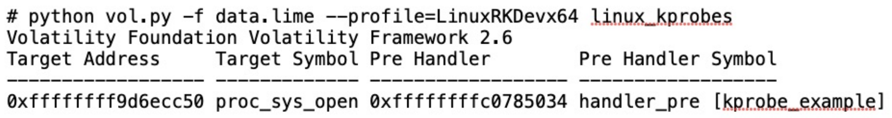
4.4.2 Volatility Plugin Implementation

Our new Volatility plugins, *linux_kprobes* and *linux_kretprobes*, are capable of enumerating all active kprobes on a system. They do so by locating and parsing all of the elements of *kprobe_table*. For each kprobe found, the plugin reports the name of the hooked symbol as well as the address and name, if present, for each of the hook handlers. The address to symbol mappings uses the same techniques as described for *linux_ftrace*.

4.4.3 Detection Examples

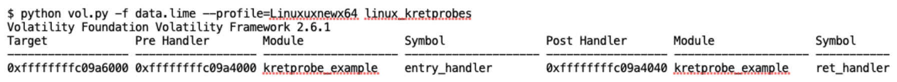
For our open source POC that abuses kprobes, we chose the example kprobe module from Spotify's GitHub repository [20]. This module hooks *do_fork* by default, but we modified it to hook *proc_sys_open* instead as the *do_fork* hook did not register correctly. We then loaded the module and acquired memory.

Figure 3 shows the output of the plugin run against the memory sample from the infected virtual machine. As can be seen, the target symbol, *proc_sys_open*, was correctly identified as well as the pre-operation handler pointing to the *handler_pre* function inside of the *kprobe_example* module.



```
# python vol.py -f data.lime --profile=LinuxRKDevx64 linux_kprobes
Volatility Foundation Volatility Framework 2.6
Target Address     Target Symbol Pre Handler        Pre Handler Symbol
------------------ ------------- ------------------ ------------------
0xffffffff9d6ecc50 proc_sys_open 0xffffffffc0785034 handler_pre [kprobe_example]
```

Figure 3: Output of linux_kprobes



```
$ python vol.py -f data.lime --profile=Linuxuxnewx64 linux_kretprobes
Volatility Foundation Volatility Framework 2.6.1
Target             Pre Handler        Module               Symbol               Post Handler       Module               Symbol
------------------ ------------------ -------------------- -------------------- ------------------ -------------------- -----------
0xffffffffc09a6000 0xffffffffc09a4000 kretprobe_example    entry_handler        0xffffffffc09a4040 kretprobe_example    ret_handler
```

Figure 4: Output of linux_kretprobes

Figure 4 illustrates the new plugin that detects kretprobes running against a memory sample in which both a *ret_handler* and a *entry_handler* were registered, to monitor both function invocation and completion. Both of the handlers are correctly identified.

4.5 Trace Events

4.5.1 Kernel Implementation

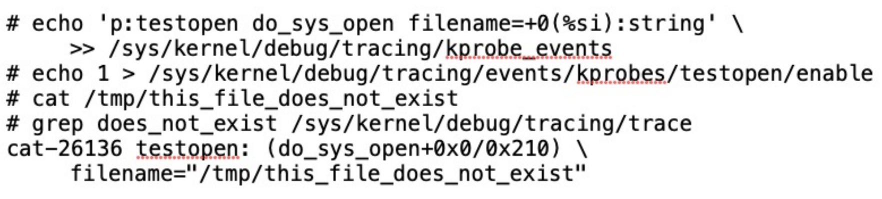
Trace events provide users the ability to write to files under */sys/kernel/debug/ tracing/* and hook functions both in userland applications (CONFIG_UPROBE_EVENTS [21]) as well as the kernel address space (CONFIG_KPROBE_EVENTS [22]). These in turn become uprobe (userland) or kprobe (kernel) events in the kernel. The set of active trace events is stored within the *ftrace_events* list. Each element of this list is of type *trace_event_call*, and has members that store the name of the function being hooked as well as the format of the trace string. These trace strings specify which parameters to a function should be logged and how they should be inspected (as a number, string, etc.). In Section 4.5.3, we show an example of this by hooking the kernel function *do_sys_open* to then record the paths of all files accessed on the system.

4.5.2 Volatility Plugin Implementation

We developed a new Volatility plugin, *linux_trace_events*, that enumerates each element of *ftrace_events*. It then prints the the function hooked along with the format for each event handler. We note that there is no kernel module associated with these userland-created hooks, so there is nothing we can key in on from that perspective. Instead, by alerting to the precense of these hooks, investigators can then use existing memory analysis techniques to recover commands and executables run on systems in an attempt uncover back the related activity.

4.5.3 Detection Example

Figure 5 shows the creation and enabling of a kprobe named *testopen* by writing to files under */sys/kernel/debug/tracing*. In particular, this kprobe hooks *do_sys_open* and extracts the filename parameter to the function. This will effectively gather the file paths of all files opened on the system. As shown, after the kprobe was activated, there was an attempt made to read a non-existent file. Searching this filename across the *trace* file afterwards showed that the testopen kprobe recorded that the cat command was used to access the file. As mentioned previously, kprobes registered through this userland interface can be used to record the parameters sent to any exported kernel function.

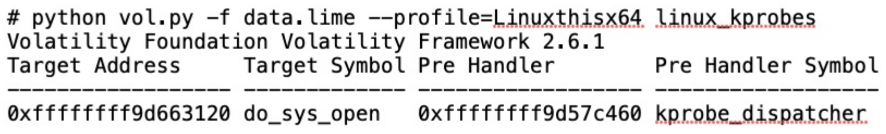


```
# echo 'p:testopen do_sys_open filename=+0(%si):string' \
     >> /sys/kernel/debug/tracing/kprobe_events
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/testopen/enable
# cat /tmp/this_file_does_not_exist
# grep does_not_exist /sys/kernel/debug/tracing/trace
cat-26136 testopen: (do_sys_open+0x0/0x210) \
     filename="/tmp/this_file_does_not_exist"
```

Figure 5: kprobe Hooking from Userland

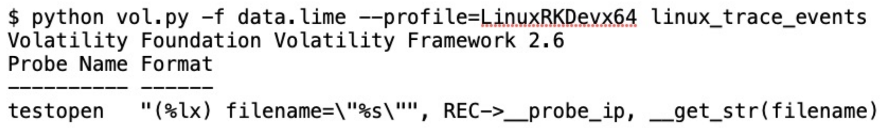
Figure 6 shows the output of *linux_kprobes* against a sample taken after testopen was activated. The *kprobe_dispatcher* function in the kernel is used to handle these tracing-based probes, and, as such, the only information gained from examining the kprobes table is that *do_sys_open* is hooked.

To gain complete information, *linux_trace_events* can be used as shown in Figure 7. This information includes the name of the probe as well as the argument printing format. These output strings would make for ideal candidates to search across all of a memory sample in an attempt to gather related activity.



```
# python vol.py -f data.lime --profile=Linuxthisx64 linux_kprobes
Volatility Foundation Volatility Framework 2.6.1
Target Address     Target Symbol Pre Handler        Pre Handler Symbol
------------------ ------------- ------------------ ------------------
0xffffffff9d663120 do_sys_open   0xffffffff9d57c460 kprobe_dispatcher
```

Figure 6: Output of linux_kprobes



```
$ python vol.py -f data.lime --profile=LinuxRKDevx64 linux_trace_events
Volatility Foundation Volatility Framework 2.6
Probe Name Format
---------- ------
testopen   "(%lx) filename=\"%s\"", REC->__probe_ip, __get_str(filename)
```

Figure 7: Output of linux_trace_events

4.6 eBPF

4.6.1 Kernel Implementation

eBPF is a significant subsystem of Linux that provides an in-kernel JIT engine to execute a variety of eBPF-program types. These program types allow observing and changing significant portions of the kernel's control flow and operation. To avoid re-inventing the wheel, eBPF re-uses previously discussed facilities when possible to implement features.

The usual method to use eBPF is through programs. These programs can be written in a language similar to C and there are also a number of convenient wrappers and APIs. Before being allowed to execute, programs are verified to ensure they do not perform illegal operations [23]. The *bpftrace* suite provides a very simple-to-use API and a number of eBPF programs that implement a wide range of system monitoring features [24], including monitoring of network connections, program execution, and file system access. This project provides a well-documented introduction to the (ab)use of eBPF programs and features.

Internally, the kernel stores the set of active ePBF programs in the *prog_idr* radix tree and each instance is of *bpf_prog*. Each program tracks its type, attach type, name, and set of instructions. Since the programs can be JIT'd, the instructions can be either in the native eBPF format or already converted to raw assembly instructions.

eBPF is also tightly coupled with the *perf_event* subsystem of Linux [25]. This subsystem allows monitoring of essentially all operations of the kernel and userland, and has built-in support for tracepoints, software events, and hardware events. The main userland interface to these features is through the *perf* utility maintained as part of the kernel itself [26], but *bpftrace* and other eBPF programs also leverage these features.

Performance events are tracked on a per-process or per-CPU basis depending on the event type and target. The per-process events are tracked in a doubly linked list of *perf_event* structures that are referenced from a process' *perf_event_list* member. The *perf_event* structure tracks the name of the event as well as associated information, such as the tracepoint reference (*tp_event*) or the eBPF program reference (*prog*). This structure ties a performance event to its owning context as well as the subsystem and handlers used to implement it.

4.6.2 New Volatility Plugins

While studying the kernel internals related to eBPF we learned that the crash utility from RedHat has a fairly robust parser for the associated data structures [27]. Like WinDbg for Windows, this utility is meant to help developers analyze kernel dump files after a system crash, but it is also certainly usable in many memory analysis tasks. The crash extension is able to list loaded eBPF programs as well as other metadata not relevant to our work. This existing extension helped guide our kernel source study and plugin implementation.

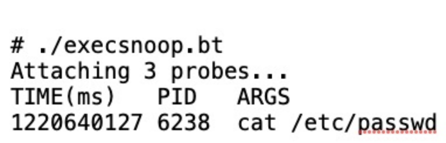
Our newly developed *linux_ebpf* plugin analyzes all eBPF programs active in a memory sample. It begins by enumerating *prog_idr* and then lists the metadata for each active program. It also has the ability to optionally extract the instructions of a program. These can then be analyzed with the llvm suite of tools.

We also developed the *linux_perf_events_ebpf* plugin that lists information related to eBPF-related performance events. This plugin operates by walking the list of active processes and then determining which have executed eBPF programs. These processes are found by attempting to enumerate the list referenced from the *perf_event_list* member. If any elements are found on this list, then the metadata of each program is reported.

4.6.3 Detection Example

To showcase these plugins, we executed the *execsnoop* utility of *bpftrace* [28]. This utility registers a tracepoint through eBPF that monitors the execve system call. This system call is used for program execution and includes the command line arguments passed to the program.

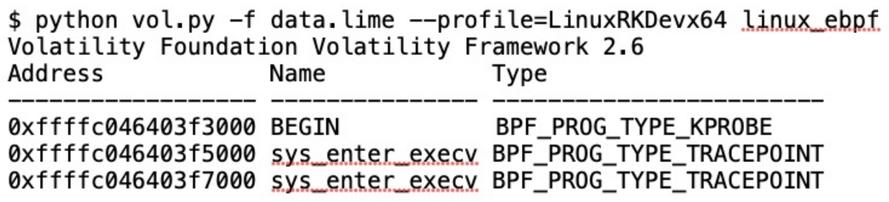
Figure 8 shows us executing execsnoop in one terminal, and then after it loaded we ran *cat /etc/passwd* in another. As can be seen, our cat command is reported by execsnoop along with the process ID.



```
# ./execsnoop.bt
Attaching 3 probes...
TIME(ms)   PID   ARGS
1220640127 6238  cat /etc/passwd
```

Figure 8: Loading execsnoop and running cat

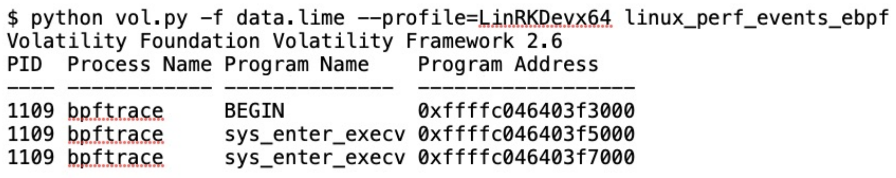
Figure 9 shows the output of *linux_ebpf* against the memory sample with execsnoop active. As can be seen, it correctly reports that the system call handler for execve, *sys_enter_execve*, is being targeted with a tracepoint by an eBPF program.



```
$ python vol.py -f data.lime --profile=LinuxRKDevx64 linux_ebpf
Volatility Foundation Volatility Framework 2.6
Address            Name            Type
------------------ --------------- ------------------------
0xffffc046403f3000 BEGIN            BPF_PROG_TYPE_KPROBE
0xffffc046403f5000 sys_enter_execv BPF_PROG_TYPE_TRACEPOINT
0xffffc046403f7000 sys_enter_execv BPF_PROG_TYPE_TRACEPOINT
```

Figure 9: Output of linux_ebpf

To determine the specific process that executed the eBPF program, we use the *linux_perf_events_ebpf* plugin as show in Figure 10.



```
$ python vol.py -f data.lime --profile=LinRKDevx64 linux_perf_events_ebpf
Volatility Foundation Volatility Framework 2.6
PID  Process Name Program Name    Program Address
---- ------------ --------------  ------------------
1109 bpftrace     BEGIN           0xffffc046403f3000
1109 bpftrace     sys_enter_execv 0xffffc046403f5000
1109 bpftrace     sys_enter_execv 0xffffc046403f7000
```

Figure 10: Output of linux_perf_events_ebpf

As the output illustrates, the bpftrace program that we used to launch execsnoop is correctly tied to the *sys_enter_execve* hooks. We also note that the BEGIN program is an artifact of how the bpftrace tool works, as described in the documentation [29]. A careful look at the output also shows that it correctly extracts the program addresses which match the previous *linux_ebpf* output.

5 Conclusion

The prevalence and power of the Linux kernel tracing infrastructure necessitates that defenders have proper toolsets to detect abuse of these features. In this paper, we have presented our research effort to deeply examine the core components of this infrastructure. The results of this effort are a number of new Volatility plugins that investigators can immediately use to detect abuse of these features within analyzed memory samples. These plugins were developed for both the Volatility 2 and Volatility 3 frameworks to allow for immediate use in the field with Volatility 2 and to provide fully-commented and documented plugins that showcase a number of Volatility 3 features and APIs. These Volatility 3 plugins can now serve as a basis for future researchers to understand how to parse and present a wide variety of in-memory artifacts. Our paper has also documented a number of open-source utilities and proof-of-concept tools that will allow defenders to examine the behaviour of these tracing features within their own environments. This is the most direct way for defenders to discover what these behaviours look like and how the most effective detections be can built in specific environments.

References

[1] B. With, “Web Server Usage Distribution in the Top 1 Million Sites,” https://trends.builtwith.com/web-server, 2021.

[2] ZDNet, “Microsoft Developer Reveals Linux is Now More Used on Azure than Windows Server,” https://www.zdnet.com/article/microsoft-developer-reveals-linux-is-now-more-used-on-azure-than-windows-server/, 2019.

[3] Cilium, “eBPF - The Future of Networking and Security,” https://cilium.io/blog/2020/11/10/ebpf-future-of-networking, 2020.

[4] Facebook, “Open-sourcing Katran, a Scalable Network Load Balancer,” https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2019.

[5] Cilium, “BPF: A New Type of Software,” http://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html, 2019.

[6] Jesper Brouer and Andy Gospodarek, “A Practical Introduction to XDP,” https://www.linuxplumbersconf.org/event/2/contributions/71/attachments/17/9/presentation-lpc2018-xdp-tutorial.pdf, 2018.

[7] XDP-project, “Tutorial: Packet02 - Packet Rewriting,” https://github.com/xdp-project/xdp-tutorial/tree/master/packet02-rewriting, 2019.

[8] Julia Evans, “Linux Tracing Systems and How They Fit Together,” https://jvns.ca/blog/2017/07/05/linux-tracing-systems/, 2017.

[9] Kernel Development Community, “Linux Tracing Technologies,” https://www.kernel.org/doc/html/latest/trace/index.html, 2021.

[10] awesomeebf, “Awesome ebpf,” https://github.com/zoidbergwill/awesome-ebpf, 2021.

[11] BPF Trace Team, “bpftrace Install,” https://github.com/iovisor/bpftrace/blob/master/INSTALL.md, 2021.

[12] Linux Kernel Driver Database, “CONFIG_KPROBE_EVENTS: Enable kprobes-based Dynamic Events,” https://cateee.net/lkddb/web-lkddb/KPROBE_EVENTS.html, 2021.

[13] ——, “CONFIG_UPROBE_EVENTS: Enable uprobes-based Dynamic Events,” https://cateee.net/lkddb/web-lkddb/UPROBE_EVENTS.html, 2021.

[14] Ubuntu, “Ubuntu Kernel Variants from Canonical,” https://ubuntu.com/kernel/variants, 2021.

[15] ElfMaster, “Kernel Instrumentation Using kprobes,” vol. 14, 2010.

[16] epbf Project, “Miscellaneous eBPF Tooling,” https://github.com/nccgroup/ebpf, 2019.

[17] Kernel Development Community, “Using ftrace to Hook to Functions,” https://www.kernel.org/doc/html/latest/trace/index.html, 2021.

[18] ilammy, “ftrace-hook,” https://github.com/ilammy/ftrace-hook, 2019.

[19] Hugo Guiroux, “Hooking into the Kernel: Real-time Code Execution at Kernel Level,” https://hugoguiroux.blogspot.com/2016/01/hooking-into-kernel-real-time-code.html, 2016.

[20] Spotify, “kprobes Example,” https://github.com/spotify/linux/blob/master/samples/kprobes/kprobe_example.c, 2008.

[21] Kernel Development Community, “Uprobe-tracer: Uprobe-based Event Tracing,” https://www.kernel.org/doc/html/latest/trace/uprobetracer.html, 2021.

[22] ——, “kprobe-based Event Tracing,” https://www.kernel.org/doc/html/latest/trace/kprobetrace.html, 2021.

[23] Alan Maguire, “BPF In Depth: The BPF Bytecode and the BPF Verifier,” https://blogs.oracle.com/linux/notes-on-bpf-5, 2019.

[24] IO Visor Project, “bpftrace,” https://github.com/iovisor/bpftrace, 2017.

[25] ——, “perf Examples,” http://www.brendangregg.com/perf.html, 2020.

[26] Kernel Development Community, “perf: Linux Profiling with Performance Counters,” https://perf.wiki.kernel.org/index.php/Main_Page, 2020.

[27] David Anderson, “bpf.c in the Core Analysis Suite,” https://github.com/crash-utility/crash/blob/master/bpf.c, 2018.

[28] IO Visor Project, “Trace new processes via exec() syscalls,” https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md#13-beginend-built-in-events, 2021.

[29] ——, “bpftrace Reference Guide: BEGIN / END events,” https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md#13-beginend-built-in-events, 2021.