red canary

RESOURCES • BLOG

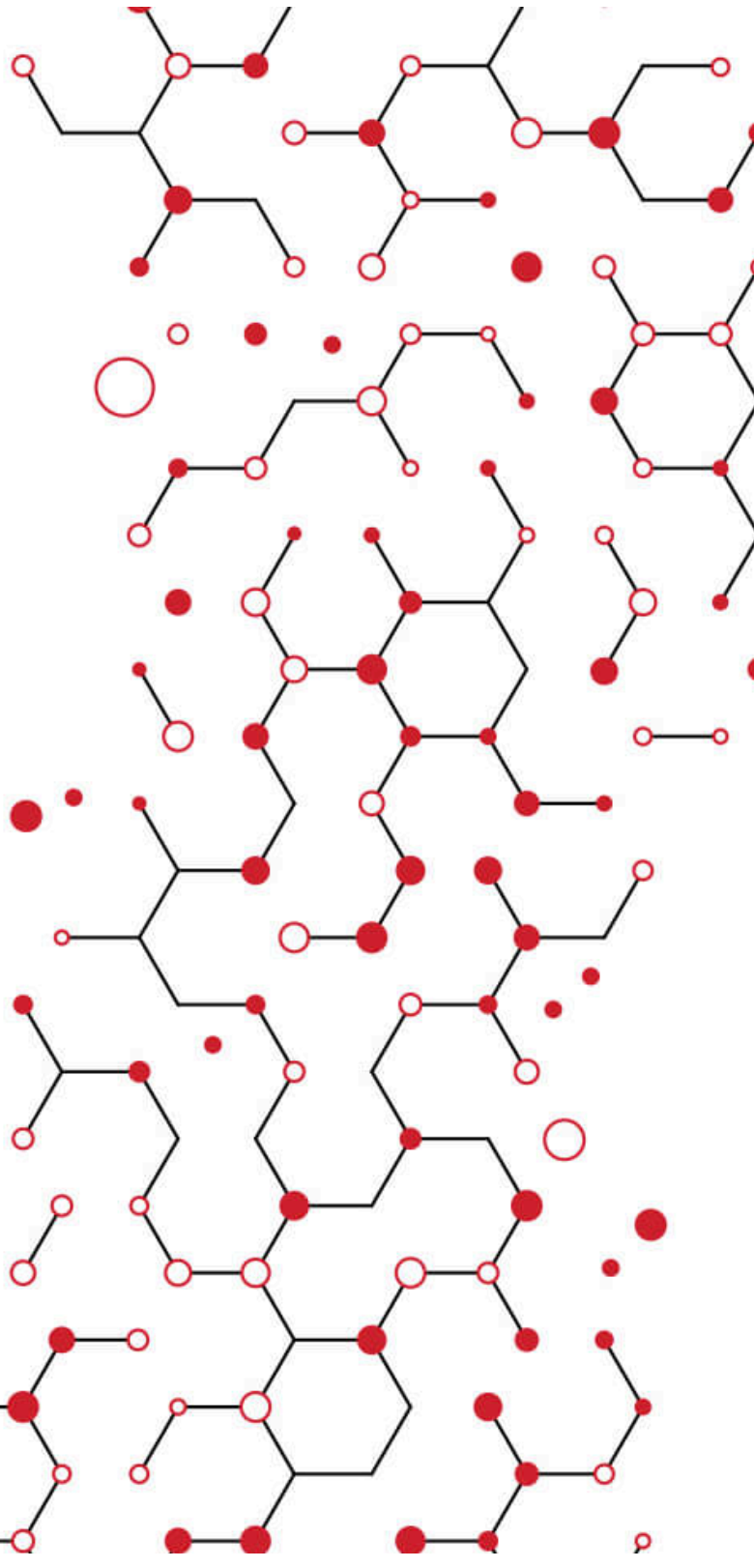
LINUX SECURITY

eBPF: A new frontier for malware

eBPF is beginning to transform the Linux malware landscape—here's what defenders should look out for.

DAVE BOGLE

Originally published January 5, 2023.
Last modified October 1, 2024.



eBPF (**e**xtended **B**erkeley **P**acket **F**ilter) has taken the **Linux world by storm**. First introduced in 2013 to enable programmable networking, it's now used for observability, security, networking, and much more. Many large companies—including Meta, Google, Microsoft, and Netflix—are committed to helping **develop and support it**.

Note: "eBPF" and "BPF" are practically synonymous and the community often uses these terms interchangeably, in part because eBPF has almost entirely supplanted the classic BPF technology.

In this article, we're going to examine a relatively new use case for eBPF. Over the past few years researchers have been investigating eBPF's usefulness in developing malware. More recently, there have been a few reported examples of actual malware leveraging this technology in the wild. See the appendix at the end of this post for a list of numerous talks, research projects, and reports about malware leveraging eBPF.

What is eBPF?

eBPF stands for extended Berkeley Packet Filter. It's an extension of the **original Berkeley Packet Filter** (BPF) design, which was a very simplistic construct used for network filtering. Initially developed to enable programmable networking, use cases for eBPF have since expanded to include security, observability, tracing, and many other applications. It has its **own website**, is part of the **Linux foundation**, and is supported by many companies.

At its core, eBPF is an **instruction set architecture** (ISA) that can be run in a virtual machine-like construct inside the Linux kernel. It has registers, instructions, and a stack, to name a few of its components. In order to use eBPF, users create eBPF programs and attach them to appropriate places around the system, typically in the kernel. When events relating to the attach point occur, the program runs and has the opportunity to read data from the system and return that data to a controlling application in user space. To sum it up, eBPF allows users to dynamically install code that can execute in kernel context but be orchestrated from user space. It's sort of a hybrid between user-space applications and Linux kernel modules.

> eBPF allows users to dynamically install code that can execute in kernel context but be orchestrated from user space.

Lifetime of an eBPF program

To better understand how eBPF works, let's briefly examine the lifetime of an eBPF program:

*The following images are based on resources from the **official eBPF website**.*

1. An eBPF program typically starts out as a "restricted" C program. Restricted means things like stack size, program size, looping, available functions, and more are limited in comparison to a normal C program. The C code gets compiled into eBPF byte code.

3. Before the eBPF code is fully loaded into the kernel, it runs through a verifier. The verifier's job is to determine if the eBPF program is safe to run. By "safe," we mean it will not get stuck in an infinite loop, there are no unsafe memory operations, and it is below the maximum complexity/code size.

4. After a kernel component verifies the program, it then attaches itself to the appropriate place in the kernel. How this happens varies widely depending on the type of eBPF program. For example, an XDP program, which is typically used for high-speed packet management, gets attached to a network interface. A tracepoint program gets attached to one of the many predefined places in the kernel used for tracing. Uprobe programs can be attached to arbitrary locations in user-space applications. So to reiterate, how to attach, where to attach, and what program to attach will vary widely based on the program type.

5. Now that the program is verified and attached, we just wait. eBPF programs are generally event-driven, so the program will only be run when a specific event occurs. At that time, the code executes in the context that it was attached to (usually in kernel space).

This diagram summarizes the entire lifetime of an eBPF program:

eBPF goes super Saiyan

So now that we understand what eBPF is, what makes it so interesting to malware authors? To understand the answer to that question we need to examine some of the powers that eBPF has gained over the years.

As mentioned earlier, it was initially designed with networking in mind. Since then, many new and different types of eBPF programs have greatly expanded its scope and capabilities. Let's look at a few program types that will be key to understanding how eBPF malware typically works.

Socket filters

Socket filters were the original use case for classic BPF. A socket filter is an eBPF program that can be attached to a socket. That program can then filter the incoming traffic for that socket. The name Berkley Packet Filter implies it was a technology designed to filter packet data. This functionality has remained even into modern eBPF.

kprobes, uprobes, and tracepoints

An eBPF program can be attached to a kernel probe (**kprobe**), a user probe (**uprobe**), or a tracepoint. Kprobes and uprobes can be attached to virtually any location in kernel space or user space respectively. Tracepoints, on the other hand, can only be attached to predefined locations in kernel or user space. Any time that function or address runs, the eBPF program will be called and the program will be able to examine information about the function call and the system at that point in time.

eXpress Data Path (XDP)

XDP programs are attached early in the packet-processing pipeline. This allows for quick and efficient packet processing. It's typically used for things like **DDoS mitigation**, packet flow management, and routing, to name a few use-cases. It has the ability to modify or redirect packets. XDP only runs on packet reception.

Traffic Control (TC)

A TC program is similar to XDP. The main difference is that it runs a bit later in the networking stack. It also can run on packet ingress and egress. The benefit to running later in the networking stack is that it offers more context around the packet. This allows more data to be gathered about the packet and where it's headed.

The dark side of eBPF

Based on the rapid adoption of eBPF, it's clearly here to stay—and likely to grow even more powerful. Let's now dive into the dark side of eBPF and see how researchers and malware have begun to leverage some of these powerful features.

`bpf_probe_write_user`

eBPF programs have access to a limited set of helper functions. These functions are built into the kernel. One helper that has been utilized by eBPF-based malware is `bpf_probe_write_user`. This function allows an eBPF program to write to the user-space memory of the process that is currently running. Malware can use this ability to modify the memory of a process during syscalls, for example how **bad-bpf** writes to user-space memory when `sudo` reads `/etc/sudoers`. It injects an extra line allowing a specific user to use the `sudo` command.

function

`bpf_override_return`

Another eBPF helper function, `bpf_override_return`, allows the program to override return values. Malware developers can leverage this to block actions that they consider undesirable. For example, if you wanted to run `kill -9 <pid-of-ebpf-malware>` , the malware could attach a kprobe to the appropriate kernel function for handling the kill signal, return an error, and effectively block the syscall from occurring. **ebpfkit** uses this to block actions that could lead to the discovery of the user-space process controlling the eBPF programs.

Limitations:

- There is a kernel build-time option to enable it: `CONFIG_BPF_KPROBE_OVERRIDE`
- It only works with functions that use the `ALLOW_ERROR_INJECTION` macro
- It is only supported on x86 right now
- It can only be used with kprobes

XDP and TC

ebpfkit leverages XDP and TC to communicate inconspicuously. Below is a slide from a **Blackhat conference talk** where the creators of ebpfkit (Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau) outlined how they used XDP and TC to hide commands being sent to ebpfkit. A request is received and processed by an XDP program running on the host. The program recognizes it as a request to the malware running on the host and modifies the packet to appear as a normal HTTP request to a web app running on the host machine. On egress, ebpfkit captures the response from the webapp using a TC program and modifies its output with the response data from ebpfkit.

Limitations:

- XDP programs run so early the data is not associated with a process or socket so very little context exists around the packet

BPFDoor uses socket filters to enable stealthy communications. It's able to receive commands on any port on the system since the eBPF program it uses sees all incoming traffic. **Symbiote** also leverages socket filters but in a different way. Symbiote hooks the `setsockopt` function call and, when it sees the creation of a socket filter, injects its own code to filter traffic it wants to hide. This allows it to evade packet analysis tools like `tcpdump`.

These are just a few examples of how malware can leverage eBPF. There are likely many more.

Windows beware

As a quick aside, for all the Windows users who were thinking, "This is just another Linux threat…" beware! **It's already in Windows!**

Don't be too worried though. The implementation for Windows is still very new and has a very limited feature set, and, therefore, it may be a while, if ever, until we see eBPF-based malware on Windows.

Prevention

So is there any hope of defeating such a powerful foe? Of course! As with any malware, the first line of defense is to take preventive measures. If implemented, the following steps may help security teams prevent eBPF malware:

- **Make sure unprivileged eBPF is disabled.** Nowadays, to install an eBPF program, you typically need root—or at least CAP_SYS_ADMIN and/or CAP_BPF. This was not always the case. Unprivileged eBPF was introduced around kernel 4.4. Be sure to check this config option by running:
  ```
  # sysctl kernel.unprivileged_bpf_disabled
  ```
- **Disable features you don't need.** Admins can programmatically disable things like kprobes:
  ```
  # echo 0 > /sys/kernel/debug/kprobes/enabled
  ```
- **Create firewall filters on external firewalls to block suspicious packets**
- **Build kernels without support for kprobes, eBPF based TC filters, or eBPF entirely (although that may not be an option for many)**
- **Ensure `CONFIG_BPF_KPROBE_OVERRIDE` is not set unless absolutely necessary**

Detection

monitoring tool was present when eBPF-based malware was loaded, there are still a few things you can do to try and detect eBPF-based malware.

Look for unexpected kprobes loaded:

```
# cat /sys/kernel/debug/kprobes/list
ffffffff8ad687e0 r ip_local_out+0x0 [FTRACE]
ffffffff8ad687e0 k ip_local_out+0x0 [FTRACE]
```

Use `bpftool` to list programs. It is not uncommon to see some eBPF programs such as `cgroup_skb` programs.

```
# bpftool prog
176: cgroup_skb tag 6deef7357e7b4530 gpl
loaded_at 2022-10-31T04:38:09-0700 uid 0
xlated 64B jited 54B memlock 4096B
185: kprobe tag a7ce508aab49e47f gpl
loaded_at 2022-10-31T10:03:16-0700 uid 0
xlated 112B jited 69B memlock 4096B map_ids 40

# bpftool perf
pid 543805 fd 22: prog_id 3610 kprobe func tcp_v4_connect offset 0
pid 543805 fd 23: prog_id 3610 kprobe func tcp_v6_connect offset 0
pid 543805 fd 25: prog_id 3611 kretprobe func tcp_v4_connect offset 0
pid 543805 fd 26: prog_id 3611 kretprobe func tcp_v6_connect offset 0
pid 543805 fd 28: prog_id 3612 kretprobe func inet_csk_accept offset 0
```

Look for loaded XDP programs. You'll see a line similar to this in the output.

```
$ ip link show dev <interface>
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xdpgeneric qdisc noqueue state UNKNOWN mode
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
prog/xdp id 220 tag 3b185187f1855c4c jited
```

Check if there are any pinned objects in the bpffs (BPF filesystem).

```
$ mount | grep bpf
…
bpf on /sys/fs/bpf type bpf (rw,nosuid,nodev,noexec,relatime,mode=700)
…

# ls -la /sys/fs/bpf/
```

Check if any TC programs are loaded.

```
# tc filter show dev <device-name>
```

Monitor system logs for any mention of the BPF helper-generated warning messages.

```
# dmesg -k | grep 'bpf_probe_write_user'
```

In summary, eBPF is a powerful tool for legitimate developers and malware authors alike. Due to this paradigm shift from how malware has been implemented in the past, we need to continue researching eBPF to better understand the threat landscape.

Our threat research team at Red Canary is at the forefront of researching and understanding emerging threats. We will continue to publish our findings as we come to better understand how to identify and detect adversary abuse of eBPF. Stay tuned for more deeper dives into what this type of malware looks like, how it behaves, and how best to detect it.

Appendix

DEFCON talks

- Evil eBPF In-Depth: Practical Abuses of an In-Kernel Bytecode Runtime
- Warping Reality—Creating and countering the next generation of Linux rootkits using eBPF
- eBPF, I thought we were friends!

Blackhat talks

- With friends like eBPF, who needs enemies?
- Fixing a memory forensics blind spot: Linux kernel tracing
- eBPF ELFs JMPing through the Windows

GitHub projects

- Boopkit
- Ebpfkit
- bad-bpf

Known malware

- Bpfdoor
- Symbiote

LINUX SECURITY

Look beyond processes with Linux EDR

LINUX SECURITY

Contain yourself: An intro to Linux EDR

LINUX SECURITY

eBPFmon: A new tool for exploring and interacting with eBPF applications

Subscribe to our blog

You'll receive a weekly email with our new blog posts.

First Name

Last Name

Email Address

SUBSCRIBE >