With Friends like eBPF, who needs enemies?

Guillaume Fournier
Sylvain Baubeau

August 2021
About us

- Cloud Workload Security Team
- Leverage eBPF to detect attacks at runtime
- Integrated in the Datadog Agent

Sylvain Afchain
Staff Engineer
sylvain.afchain@datadoghq.com

Sylvain Baubeau
Staff Engineer & Team lead
sylvain.baubeau@datadoghq.com

Guillaume Fournier
Security Engineer
guillaume.fournier@datadoghq.com
Agenda

- Introduction to eBPF
- Abusing eBPF to build a rootkit
  - Obfuscation
  - Persistent access
  - Command and Control
  - Data exfiltration
  - Container breakout
- Detection and mitigation strategies
Introduction to eBPF
Introduction to eBPF

What is eBPF?

- Extended Berkeley Packet Filter
- Sandboxed programs in the Linux kernel
- Initially designed for fast packet processing

Use cases:
- Kernel performance tracing
- Network security and observability
- Runtime security
- etc
Introduction to eBPF

Step 1: Loading eBPF programs
Introduction to eBPF
Step 2: Attaching eBPF programs

● Defines how a program should be triggered
● ~ 30 program types (Kernel 5.13+)

● Depends on the program type
  ○ BPF_PROG_TYPE_KPROBE
  ○ BPF_PROG_TYPE_TRACEPOINT
  ○ BPF_PROG_TYPE_SCHED_CLS
  ○ BPF_PROG_TYPE_XDP
  ○ etc

● Programs of different types can share the same eBPF maps

“perf_event_open” syscall
Dedicated Netlink command
The eBPF verifier ensures that eBPF programs will finish and won’t crash.

- Directed Acyclic Graph
- No unchecked dereferences
- No unreachable code
- Limited stack size (512 bytes)
- Program size limit (1 million on 5.2+ kernels)
- Bounded loops (5.2+ kernels)
- … and cryptic output …
Introduction to eBPF

eBPF internals: eBPF helpers

● Context helpers
  ○ bpf_get_current_task
  ○ bpf_get_current_pid_tgid
  ○ bpf_ktime_get_ns
  ○ etc

● Map helpers
  ○ bpf_map_lookup_elem
  ○ bpf_map_delete_elem
  ○ etc

● Program type specific helpers
  ○ bpf_xdp_adjust_tail
  ○ bpf_csum_diff
  ○ bpf_l3_csum_replace
  ○ etc

● Memory related helpers
  ○ bpf_probe_read
  ○ bpf_probe_write_user
  ○ etc

… ~160 helpers (kernel 5.13+)
Abusing eBPF to build a rootkit
Abusing eBPF to build a rootkit

Why?

- Cannot crash the host
- Minimal performance impact
- Fun technical challenge
- A growing number of vendors use eBPF
- eBPF “safety” should not blind Security Administrators
Abusing eBPF to build a rootkit

Goals

- Trade off between latest BPF features / availability
  => Latest Ubuntu LTS, RHEL/CentOS

- KRSI and helpers such bpf_dpath may help
Abusing eBPF to build a rootkit

Obfuscation

- Hide the rootkit process
  - eBPF programs are attached to a running process
    - Our userspace rootkit has to stay resident
  - Detection through syscalls that accept pids as arguments: `kill`, `waitpid`, `pidfd_open`, ...

- Hide our BPF components:
  - programs
  - maps
Demo
Abusing eBPF to build a rootkit

Program obfuscation - Techniques

- `bpf_probe_write_user`
  - Corrupt syscall output
  - Minor and major page faults

- `bpf_override_return`
  - Block syscall
  - Alter syscall return value
    - But syscall was really executed by the kernel!
Abusing eBPF to build a rootkit

File obfuscation - stat /proc/<rootkit-pid>/cmdline (1)
Abusing eBPF to build a rootkit

Program obfuscation - stat /proc/<rootkit-pid>/exe (2)
Abusing eBPF to build a rootkit

Program obfuscation

- Block signals
  - Hook on the kill syscall entry
  - Override the return value with ESRCH
- Block kernel modules
Abusing eBPF to build a rootkit

BPF program obfuscation

Demo
Abusing eBPF to build a rootkit
BPF program obfuscation

- bpf syscall
  - Programs:
    - BPF_PROG_GET_NEXT_ID
    - BPF_PROG_GET_FD_BY_ID
  - Maps:
    - BPF_MAP_GET_NEXT_ID
    - BPF_MAP_GET_FD_BY_ID
  - Hook on new prog / map to get the allocated ID

- Hook on read syscall and override the content
Abusing eBPF to build a rootkit

BPF program obfuscation

- bpf_probe_write_user
  - message in kernel ring buffer
  - dmesg
  - journalctl -f
  - syscall syslog
Abusing eBPF to build a rootkit

BPF program obfuscation

Diagram:

- **t0**: journalctl /dev/kmsg
- **t1**: read
- **t2**: read
- **t3**: read overridden content
- **t4**: (Not shown)

Flow:

1. **Journalctl**: 
   - **Read**: `read` (spin loop)
   - **Rootkit**: Override return `read` > 0
   - **Write legit looking content**

2. **Install override content**
   - **Override content + zero**
Abusing eBPF to build a rootkit

Persistent access

- **Self copy**
  - Generate random name
  - Copy into `/etc/rcS.d`
  - Hide file

- **Override content of sensitive files**
  - SSH authorized_keys
  - `passwd`
  - `crontab`

- **Persistent access to an application database**
Abusing eBPF to build a rootkit
Persistent access - uprobes

- eBPF on exported user space functions
- Alter a userspace daemon to introduce a backdoor
- Compared to ptrace
  - Works on all instances of the program
  - Safer
  - Easier to write
Abusing eBPF to build a rootkit
Persistent access - postgresql
Abusing eBPF to build a rootkit
Persistent access - postgresql

```c
int md5_crypt_verify( const char *role, const char *shadow_pass, const char *client_pass,
const char *md5_salt, int md5_salt_len, char **logdetail )

● md5_salt
  challenge sent when user connects

shadow_pass  MD5(role + password)
  stored in database

client_pass  MD5(shadow_pass + md5_salt)
  sent by the client

● new_md5_hash = bpf_map_lookup_elem(&postgres_roles, &creds.role);
  if (new_md5_hash == NULL) return 0;

  // copy db password onto the user input
  bpf_probe_write_user(shadow_pass, &new_md5_hash->md5, MD5_LEN);
```
Abusing eBPF to build a rootkit
Command and control: introduction

- **Requirements**
  - Send commands to the rootkit
  - Exfiltrate data
  - Get remote access to infected hosts

- **eBPF related challenges**
  - Can't initiate a connection
  - Can't open a port

- … but we can hijack an existing connection!
Abusing eBPF to build a rootkit

Command and control: introduction

- Setup
  - Simple webapp with AWS Classic Load Balancer
  - TLS resolution at the Load Balancer level

- Goal: Implement C&C by hijacking the network traffic to the webapp
### Abusing eBPF to build a rootkit

#### Command and control: choosing a program type

<table>
<thead>
<tr>
<th>BPF_PROG_TYPE_XDP</th>
<th>BPF_PROG_TYPE_SCHED_CLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>🟩 Deep Packet Inspection</td>
<td>🟩 Deep Packet Inspection</td>
</tr>
<tr>
<td>🟩 Ingress only</td>
<td>🟩 Egress and Ingress</td>
</tr>
<tr>
<td>🟩 Can be offloaded to the NIC / driver</td>
<td>🟩 Attached to a network interface</td>
</tr>
<tr>
<td>🟩 Can drop, allow, modify and retransmit packets</td>
<td>🟩 Can drop, allow and modify packets</td>
</tr>
<tr>
<td>🟩 Usually used for DDOS mitigation</td>
<td>🟩 Often used to monitor &amp; secure network access at the container / pod level on k8s</td>
</tr>
</tbody>
</table>
Abusing eBPF to build a rootkit

Command and control: choosing a program type

- BPF_PROG_TYPE_XDP
  - Deep Packet Inspection
  - Ingress only
  - Can be offloaded to the NIC / driver
  - Can drop, allow, modify and retransmit packets
  - Usually used for DDOS mitigation

- BPF_PROG_TYPE_SCHED_CLS
  - Deep Packet Inspection
  - Egress and Ingress
  - Attached to a network interface
  - Can drop, allow and modify packets
  - Usually used to monitor & secure network access at the container / pod level on k8s

Network packets can be hidden from the Kernel entirely!
### Abusing eBPF to build a rootkit

#### Command and control: choosing a program type

<table>
<thead>
<tr>
<th>BPF_PROG_TYPE_XDP</th>
<th>BPF_PROG_TYPE_SCHED_CLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>❑ Deep Packet Inspection</td>
<td>❑ Deep Packet Inspection</td>
</tr>
<tr>
<td>❑ Ingress only</td>
<td>❑ <strong>Egress</strong> and Ingress</td>
</tr>
<tr>
<td>❑ <strong>Can be offloaded to the NIC / driver</strong></td>
<td>❑ Attached to a network interface</td>
</tr>
<tr>
<td>❑ Can drop, allow, modify and retransmit packets</td>
<td>❑ Can drop, allow and <strong>modify packets</strong></td>
</tr>
<tr>
<td>❑ Usually used for DDOS mitigation</td>
<td>❑ Usually used to monitor &amp; secure network access at the container / pod level on k8s</td>
</tr>
</tbody>
</table>

**Network packets can be hidden from the Kernel entirely!**

**Data can be exfiltrated with an eBPF TC classifier!**
Abusing eBPF to build a rootkit
Command and control: hijacking HTTP requests
Abusing eBPF to build a rootkit

Command and control: hijacking HTTP requests

Demo

Sending Postgres credentials over C&C
Abusing eBPF to build a rootkit

Data exfiltration
Abusing eBPF to build a rootkit

Data exfiltration

- Multiple program types can share data through eBPF maps
- Anything accessible to an eBPF program can be exfiltrated:
  - File content
  - Environment variables
  - Database dumps
  - In-memory data
  - etc
Abusing eBPF to build a rootkit

Data exfiltration

Demo

Exfiltration over HTTPS

Postgres credentials & /etc/passwd
Abusing eBPF to build a rootkit

DNS spoofing

The same technique applies to any unencrypted network protocol ...
Abusing eBPF to build a rootkit
Container breakout 1: escaping through a pipe

The rootkit can detect and take over pipes between 2 processes

- Kprobes and Tracepoint programs are not constrained to cgroups or namespaces
- Required access:
  - CAP_SYS_ADMIN (or CAP_BPF + CAP_PERFMON depending on the kernel version)
  - CAP_SYS_RESOURCE & CAP_NET_ADMIN & shared net namespace (optional)
  - Default Seccomp profile activated
  - Default AppArmor profile:
    - Activated for the raw_tracepoint variant
    - Deactivated for the kprobe variant
Abusing eBPF to build a rootkit

Container breakout 1: escaping through a pipe

- bash
- Kretprobe on the read syscall
  - Match FD & pid to the active pipe
  - Append the piped data to the buffer of data to pipe
  - Write the data to pipe in the user space buffer with bpf_probe_write_user
  - Change the return value to match the injected data size with bpf_override_return
- ping -c 1 google.fr ...

User space

Kernel space
Abusing eBPF to build a rootkit
Container breakout 1: escaping through a pipe

Demo

curl https://.../my_script.sh | bash

Disclaimer: for the demo, we added CAP_NET_ADMIN, CAP_SYSRESOURCE and shared the host network namespace to enable C&C
Abusing eBPF to build a rootkit
Container breakout 2: Docker shenanigans

The rootkit switches Docker images at runtime

- We use a Uprobe on the Docker Daemon on “ParseNormalizedNamed” to switch the Pause container image with a rogue image.
- Uprobe programs are not constrained to cgroups or namespaces
- Required access:
  - CAP_SYS_ADMIN (or CAP_BPF + CAP_PERFMON depending on the kernel version)
  - CAP_SYS_RESOURCE & CAP_NET_ADMIN & shared net namespace (optional)
  - Default Seccomp profile activated
  - Default AppArmor deactivated
  - The host root directory has to be shared with the container
Abusing eBPF to build a rootkit

Container breakout 2: Docker shenanigans

Diagram showing the flow of actions:
- Kubelet
- Docker Daemon
- Kprobe on the stat syscall
- Uprobe on (dockerd) ParseNormalizedNamed
- Internet
- Serves gui774ume/pause:3.2
- gui774ume/pause:3.2
- Pull gui774ume/pause:3.2
- Run gui774ume/pause:3.2
- Run k8s.io/pause:3.2
- Replace k8s.io/pause with gui774ume/pause
- Internet
- gui774ume/pause:3.2
- Execute
- Ping
- Return the configured action
- User space
- Kernel space
- Dockerhub
Demo

Let’s hit Pause for a minute
Detection and mitigation
Detection and mitigation

Step 1: assessing an eBPF based third party vendor

- **Audit & assessment**
  - Ask to see the code! (GPL)
  - Look for sensitive eBPF patterns:
    - program types
    - eBPF helpers
    - cross program types communication

- **Useful tool: “ebpfkit-monitor”**
  - parses ELF files and extract eBPF related information
  - https://github.com/Gui774ume/ebpfkit-monitor
Detection and mitigation
Step 1: assessing an eBPF based third party vendor

```
vagrant@ubuntu-focal:~$ ebpfkit-monitor -a ~/go/src/github.com/Gui774ume/ebpfkit/ebpf/bin/probe.o prog --helper FnProbeWriteUser
trace_md5_crypt_verify
  SectionName: uprobe/md5_crypt_verify
  Type: Kprobe
  InstructionsCount: 1454
  AttachType: 0
  License: GPL
  KernelVersion: 328823
  ByteOrder: LittleEndian
  Helpers:
    - FnGetPrandomU32: 4
    - FnProbeRead: 1
    - FnProbeWriteUser: 1
    - FnProbeReadStr: 2
    - FnMapLookupElem: 9
    - FnMapUpdateElem: 2
  Maps:
    - postgres_roles: 1
    - postgres_cache: 1
    - postgres_list_cursor: 1
    - dedicated_watch_keys: 1
    - fs_watches: 5
    - fs_watch_gen: 2
```

“ebpfkit-monitor” can list eBPF programs with sensitive eBPF helpers
Detection and mitigation
Step 1: assessing an eBPF based third party vendor

"ebpfkit-monitor" shows suspicious cross program types communications
Detection and mitigation
Step 2: runtime mitigation

- Monitor accesses to the “bpf” syscall
  - Keep an audit trail
  - “ebpfkit-monitor” can help!

- Protect accesses to the “bpf” syscall:
  - Block bpf syscalls from unknown processes
  - Reject programs with sensitive eBPF helpers or patterns
  - Sign your eBPF programs (https://lwn.net/Articles/853489)
  - “ebpfkit-monitor” can help!

- Prevent unencrypted network communications even within your internal network
Detection and mitigation
Step 3: Detection & Investigation

● It is technically possible to write a perfect eBPF rootkit *
● But:
  ○ look for actions that a rootkit would have to block / lie about to protect itself
  ○ (if you can) load a kernel module to list eBPF programs
  ○ (if you can) load eBPF programs to detect abnormal kernel behaviors
  ○ monitor network traffic anomalies at the infrastructure level
● Disclaimer: our rootkit is far from perfect !

* with enough time, motivation, insanity, and absolute hatred for life.
Thanks!

“ebpfkit” source code: https://github.com/Gui774ume/ebpfkit

“ebpfkit-monitor” source code: https://github.com/Gui774ume/ebpfkit-monitor