Evil eBPF

Practical Abuses of an In-Kernel Bytecode Runtime

Jeff Dileo
@chaosdatumz

DEF CON 27

nccgroup

```
call_usermodehelper("/bin/sh",(char*[]){"/bin/sh","-c","whoami",NULL},NULL,5)
```

- @chaosdatumz
- Agent of chaos
- Unix aficionado
- Principal Consultant / Research Director @ NCC Group
- I like to do terrible things to/with/in:
  - programs
  - languages
  - runtimes
  - memory
  - kernels
  - packets
  - bytes
  - ...

Outline

- Introduction to eBPF
- On Using eBPF for Malign Purposes
- Tooling Up to Build a Birdfeeder (of Doooom!)
- The IPC You Don’t See
- Reliable Corruption
- On Fighting Wizards and Dragons
- Q&A

eBPF — Background

- “extended” BPF
- What is “BPF”?

BPF

- Berkeley Packet Filter
- Limited instruction set for a bytecode virtual machine
- Originally created to implement *FAST* programmatic network filtering in kernel
- has a few (2) 32-bit registers (and a hidden frame pointer)
- load/store, conditional jump (forward), add/sub/mul/div/mod, neg/and/or/xor, bitshift

BPF

- `tcpdump -i any -n 'tcp[tcpflags] & (tcp-syn|tcp-ack) != 0'`

```
(000) ldh      [14]
(001) jeq      #0x800           jt 2   jf 10
(002) ldb      [25]
(003) jeq      #0x6             jt 4   jf 10
(004) ldh      [22]
(005) jset     #0x1fff          jt 10   jf 6
(006) ldxb     4*([16]&0xf)
(007) ldb      [x + 29]
(008) jset     #0x12            jt 9   jf 10
(009) ret      #262144
(010) ret      #0
```

eBPF

- “extended” Berkeley Packet Filter
- *“designed to be JITed with one to one mapping”*
- *“originally designed with the possible goal in mind to write programs in ‘restricted C’”*
- socket filters, packet processing, tracing, internal backend for “classic” BPF, and more...
- Many different APIs exposed through the `bpf(2)` syscall
  - The main ones are for loading/interacting with eBPF programs and ”maps”
    - Programs can be one of several types
    - Maps are in-kernel structures shared between kernel space eBPF code and userspace program code

eBPF — High Level Overview

- eBPF's virtual ISA is featureful enough to support C
- The kernel places restrictions on eBPF programs to prevent it from breaking the kernel
- eBPF programs are created through the `bpf(2)` syscall
  - Pass in an array of eBPF instructions and an eBPF program type
  - The type dictates the set of out-of-sandbox APIs the eBPF code can call
- eBPF maps are also created through the `bpf(2)` syscall
  - Generally loaded first so that loaded eBPF programs can reference them by their FD
- eBPF program FDs are then attached to kernel structures using type specific kernel APIs
- The programs are then invoked to process type- and attachment-specific events

eBPF — Things to Keep in Mind

- The interesting eBPF features require `CAP_SYS_ADMIN`
  - Without that, the only program types that can be loaded are `BPF_PROG_TYPE_SOCKET_FILTER` and `BPF_PROG_TYPE_CGROUP_SKB`
    - And the latter requires `CAP_NET_ADMIN` to attach
- The BPF helper functions do all of the heavy lifting and interesting work
  - If you want to read/write data outside of the eBPF non-Turing-tarpit, you need them
- eBPF's validator ("verifier") can be very pedantic about what eBPF programs can and can't do
  - This talk will not be covering the validator in depth
  - For information on living with it, see our 35C3 talk

Why eBPF?

- eBPF offers a lot of new features to play around with
- Originally created for (performant) packet processing, now applied to everything in the kernel
- While the interesting capabilities require high privileges, eBPF only has two modes
  - Unprivileged (basic socket filters, not very useful on their own)
  - ALL THE PRIVILEGES (everything else)
- Everything that uses eBPF for wholesome endeavors runs fully privileged
  - And are hard to sandbox

Why (Evil) eBPF?

- eBPF offers a lot of new features to play around with
- Originally created for (performant) packet processing, now applied to everything in the kernel
- While the interesting capabilities require high privileges, eBPF only has two modes
  - Unprivileged (basic socket filters, not very useful on their own)
  - ALL THE PRIVILEGES (everything else)
- Everything that uses eBPF for wholesome endeavors runs fully privileged
  - And are hard to sandbox

Why (Evil) eBPF?

- eBPF offers a lot of new features to play around with
- Originally created for (performant) packet processing, now applied to everything in the kernel
- While the interesting capabilities require high privileges, eBPF only has two modes
  - Unprivileged (basic socket filters, not very useful on their own)
  - ALL THE PRIVILEGES (everything else)
- Everything that uses eBPF for wholesome endeavors runs fully privileged
  - And are hard to sandbox

So, what is this talk about?

SHENANIGANS

An Evil Agenda

- A Treatise on Evil eBPF Tooling
- Abusing eBPF for IPC
  - Unprivileged API abuses
  - Privileged API shenanigans
- “Post-exploitation” with eBPF
  - Privileged API shenanigans

On Developing eBPF-Based Things

- Several hurdles with developing eBPF-based programs
  - Compiling eBPF code into something loadable by the kernel
  - Interacting with the kernel and userspace from eBPF code
  - Interacting with the eBPF code from userspace
  - Lack of portability/deployability due to runtime dependencies (headers, shared libs, etc.)
- In the typical Linux fashion, there are are many choices
  - And most are painful or have complicated tradeoffs

Choosing Your Level of eBPF Abstraction

There are three main choices for eBPF development toolchains:

- Raw eBPF instructions written by hand using a C macro DSL
  - Often used for very simple examples
- Direct use of LLVM/Clang to compile C files into eBPF ELF files
  - Linux kernel build infrastructure (pulls in headers, but slow build times)
    - e.g. `samples/bpf/` and `tools/bpf/`
  - Out-of-tree development (need to manage headers, but fast build times)
- High-level APIs that compile and load strings of a custom DSL C dialect
  - `iovisor/bcc` (Python)
  - `iovisor/gobpf`

There are several (overlapping) ways to invoke eBPF APIs:

- Raw syscalls (libcs do not ship syscall wrappers for eBPF)
- `libbpf` (provides syscall wrappers and more)
- `bpf_load.c` (actual deep magic!)

nccgroup

Choosing Your Level of eBPF Abstraction — Raw eBPF

- Like raw water, will give you cholera
- Very portable (not potable), but basically useless for anything worth writing

```
struct bpf_insn prog[] = {
  BPF_LD_MAP_FD(BPF_REG_2, map_fd),
  BPF_MOV64_IMM(BPF_REG_3, 3),
  BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_tail_call),
  BPF_MOV64_IMM(BPF_REG_0, -1),
  BPF_EXIT_INSN(),
};
size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
char bpf_log_buf[2048];
int prog_fd = bpf_load_program(BPF_PROG_TYPE_SOCKET_FILTER,
  prog, insns_cnt, "GPL", 0, bpf_log_buf, 2048
);
```

Choosing Your Level of eBPF Abstraction — Direct LLVM/Clang

- Clean water, but with a few hurdles:
  - Correctly exposing the right kernel headers
    - I like the (`xdp-project/xdp-tutorial`) toolchain (mimics in-tree dev, hackable build system)
  - Preprocessing/instrumentation (e.g. auto-wiring of maps into eBPF instructions and userland)
    - Both `libbpf` and `bpf_load.c` do this to different degrees

```
int filter(struct __sk_buff *skb) {
  payload_t p;
  switch (filter_type) {
    case (RAW_SOCKET_FILTER):
      p = parse_packet_from_to(skb, ETHERNET_LAYER, APPLICATION_LAYER); break;
...
  uint32_t index = 0;
  size_t* v = bpf_map_lookup_elem(&my_map, &index);
...
        size_t l = p.len; char* c = (char*)&v[4];
        #pragma unroll
        for (size_t i=0; i < 32; i++) {
          if (l > 0) {
            bpf_skb_load_bytes(skb, p.offset + i, &c[i], 1); l--;
          }
        }
```

Choosing Your Level of eBPF Abstraction — High-Level APIs

- Branwdo, the thirst mutilator; it's got what eBPF programs crave
- Make certain (specialization aligned) tasks much easier
- Add a lot of magic that can make it hard to reason about how code actually runs
  - And can make it hard to directly interface with lower-level/unsupported APIs when needed
  - Requires non-trivial toolchain to exist on the system running the code

```
from bcc import BPF
program = """
#include <asm/ptrace.h>  // for struct pt_regs
#include <linux/types.h> // for mode_t

int kprobe__sys_openat(struct pt_regs *ctx,
                       int dirfd, char __user* pathname, int flags, mode_t mode) {
  bpf_trace_printk("sys_openat called.\\n");
  return 0;
}
"""

b = BPF(text=program)
b.trace_print()
```

Choosing Your Level of eBPF Abstraction — Evil Edition

- In general, it's probably best to go with the direct LLVM/Clang approach
- We need maximum portability with limited support from our operating environment
  - Ideally, we would statically link everything into a single binary without runtime dependencies
    - So we can drop a binary that "just works"
    - This is easy to implement with simple modifications to the `xdp-project/xdp-tutorial` Makefiles
  - BCC/gobpf cannot reasonably do this
- Additionally, while BCC is quick to pick up, dealing with its abstractions takes its toll over time
  - But it's still very useful for kernel tracing

Evil IPC

eBPF Map Primer

- Generally, eBPF maps are used to interface eBPF programs with userland processes

eBPF Map Primer

- Generally, eBPF maps are used to interface eBPF programs with userland processes
- But eBPF maps do not actually need to be attached to an eBPF program
- Userland processes can use them as a way to store data off-process

eBPF Map Primer

- Generally, eBPF maps are used to interface eBPF programs with userland processes
- But eBPF maps do not actually need to be attached to an eBPF program
- Userland processes can use them as a way to store data off-process
- Additionally, eBPF maps are interacted with through their FDs

eBPF Map Primer

- Generally, eBPF maps are used to interface eBPF programs with userland processes
- But eBPF maps do not actually need to be attached to an eBPF program
- Userland processes can use them as a way to store data off-process
- Additionally, eBPF maps are interacted with through their FDs
- As a result they can be passed between processes using system APIs that transfer FDs

Map Transit

IPC via passing eBPF maps between processes and reading/writing them to send messages

1. In a userspace C program, create a BPF_MAP_TYPE_ARRAY map

```
int fd = bpf_create_map_node(
  BPF_MAP_TYPE_ARRAY, "mymap", sizeof(uint32_t), 256, 2, 0, 0
);
```

2. Use Unix domain sockets, or a similar API, to pass the map FD to a cooperating process
3. Assign index 0 for messages sent by the map creator process
4. Assign index 1 for messages sent by the cooperating process

Map Transit (2)

5. To send messages, use `bpf_map_update_elem`

```
char buf[256] = "hello world";
uint32_t key = 0;
bpf_map_update_elem(fd, &key, buf, BPF_ANY);
```

6. To receive messages, use `bpf_map_lookup_elem`

```
char buf[256];
uint32_t key = 0;
while (bpf_map_lookup_elem(fd, &key, &buf)) {
  sleep(1);
}
```

Map Transit — Warning!

- All facets of eBPF maps are managed by the kernel

Map Transit — Warning!

- All facets of eBPF maps are managed by the kernel
  - Including the sizes of their values

Map Transit — Warning!

- All facets of eBPF maps are managed by the kernel
  - Including the sizes of their values
- Due to this, blindly receiving and operating on eBPF map FDs is extremely dangerous
  - Reads from an eBPF map can overflow the target
  - Writes to an eBPF map can overread past the source

nccgroup

Map Transit — Warning!

- All facets of eBPF maps are managed by the kernel
  - Including the sizes of their values
- Due to this, blindly receiving and operating on eBPF map FDs is extremely dangerous
  - Reads from an eBPF map can overflow the target
  - Writes to an eBPF map can overread past the source
- Make sure to get and validate the type and size metadata from any received eBPF map
  - Through `bpf(BPF_OBJ_GET_INFO_BY_FD,...)`/`bpf_obj_get_info_by_fd()`

```
struct bpf_map_info info = {};
uint32_t info_len = sizeof(info);
bpf_obj_get_info_by_fd(shady_map_fd, &info, &info_len);
char* buf = (char*)malloc(info.value_size);
...
```

eBPF Program Primer

- Normally, an eBPF program is a single function with all others inlined into it

eBPF Program Primer

- Normally, an eBPF program is a single function with all others inlined into it
- But eBPF supports loading multiple eBPF programs/functions into a single execution context
  - eBPF has a map type for storing eBPF program file descriptors, `BPF_MAP_TYPE_PROG_ARRAY`
  - And eBPF's `bpf_tail_call` helper function, performs no-return calls into another program
    - By their index into a given `BPF_MAP_TYPE_PROG_ARRAY` map

eBPF Program Primer

- Normally, an eBPF program is a single function with all others inlined into it
- But eBPF supports loading multiple eBPF programs/functions into a single execution context
  - eBPF has a map type for storing eBPF program file descriptors, `BPF_MAP_TYPE_PROG_ARRAY`
  - And eBPF's `bpf_tail_call` helper function, performs no-return calls into another program
    - By their index into a given `BPF_MAP_TYPE_PROG_ARRAY` map
- Additionally, `BPF_MAP_TYPE_PROG_ARRAY` maps can be updated at runtime

eBPF Program Primer

- Normally, an eBPF program is a single function with all others inlined into it
- But eBPF supports loading multiple eBPF programs/functions into a single execution context
  - eBPF has a map type for storing eBPF program file descriptors, `BPF_MAP_TYPE_PROG_ARRAY`
  - And eBPF's `bpf_tail_call` helper function, performs no-return calls into another program
    - By their index into a given `BPF_MAP_TYPE_PROG_ARRAY` map
- Additionally, `BPF_MAP_TYPE_PROG_ARRAY` maps can be updated at runtime
- Such that `bpf_tail_call` invocations will call the new eBPF program swapped into the map

Interprocess Call-Based Messaging

IPC via the swapping in and out of eBPF programs that deliver messages to userspace

1. In the eBPF-C program, declare two maps:
   - A BPF_MAP_TYPE_PROG_ARRAY map to hold 2 program FDs, including the main entry point program
   - A BPF_MAP_TYPE_ARRAY (or similar) map to send messages to userspace
2. The body of the main entry point eBPF program should be as follows:

```
SEC("socket/0")
int main_prog(struct __sk_buff *skb) {
  bpf_tail_call(skb, &prog_map, 1);
  return -1;
}
```

3. In the ”reader” userspace program, load the above eBPF program as a BPF_PROG_TYPE_SOCKET_FILTER along with its maps
4. Use Unix domain sockets, or a similar API, to pass both map FDs to a ”writer” process

Interprocess Call-Based Messaging (2)

5. In the reader, set up a TCP socket server
6. Attach the eBPF program to it using `setsockopt(2)` with `SO_ATTACH_BPF`
7. Have the reader connect to its own server and send data to it at a regular interval
   - After sending the data, check the `BPF_MAP_TYPE_ARRAY` map for a message
8. In the writer, load a `BPF_PROG_TYPE_SOCKET_FILTER` program
   - It should declare a `BPF_MAP_TYPE_ARRAY` map identical to the one from step 1
9. Extract its instructions and iterate through them to inject the `BPF_MAP_TYPE_ARRAY` map FD in place of the one they declared

```
for (size_t i=0; i < insns_cnt; i++) {
  if (prog[i].code != (BPF_LD | BPF_IMM | BPF_DW)) {
    continue;
  }
  if (prog[i].src_reg == BPF_PSEUDO_MAP_FD) {
    prog[i].imm = recvd_array_map;
  }
}
```

nccgroup

Interprocess Call-Based Messaging (3)

10. Re-load the modified instructions to get a new eBPF program FD
11. Place the writer's eBPF program FD into index 1 of the BPF_MAP_TYPE_PROG_ARRAY map

```
uint32_t key = 1;
bpf_map_update_elem(recvd_prog_map, &key, &writer_prog_fd, BPF_ANY);
```

DEMO

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes
  - It is also poorly documented

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes
    - It is also poorly documented
- With privileges, a process can create raw IP sockets
- Without privileges, a process may only create normal sockets (TCP, UDP, Unix)
- Attaching a `BPF_PROG_TYPE_SOCKET_FILTER` program to a socket means completely different things depending on the socket type

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes
  - It is also poorly documented
- With privileges, a process can create raw IP sockets
- Without privileges, a process may only create normal sockets (TCP, UDP, Unix)
- Attaching a `BPF_PROG_TYPE_SOCKET_FILTER` program to a socket means completely different things depending on the socket type
  - For raw sockets, the program will see packets from the beginning of its ethernet frame
  - For IP-based TCP/UDP sockets, it will see packets from the start of the transport header
  - For Unix sockets, it will see packets from the start of the data payload

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes
  - It is also poorly documented
- With privileges, a process can create raw IP sockets
- Without privileges, a process may only create normal sockets (TCP, UDP, Unix)
- Attaching a `BPF_PROG_TYPE_SOCKET_FILTER` program to a socket means completely different things depending on the socket type
  - For raw sockets, the program will see packets from the beginning of its ethernet frame
  - For IP-based TCP/UDP sockets, it will see packets from the start of the transport header
  - For Unix sockets, it will see packets from the start of the data payload
- Additionally, while these programs cannot modify the packets, they can drop them, which breaks `SOCK_STREAM`, but is not `SOCK_DGRAM`

eBPF Socket Filter Primer

- The `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program type is special
- It is the only one that may be used *freely* freely by unprivileged processes
  - It is also poorly documented
- With privileges, a process can create raw IP sockets
- Without privileges, a process may only create normal sockets (TCP, UDP, Unix)
- Attaching a `BPF_PROG_TYPE_SOCKET_FILTER` program to a socket means completely different things depending on the socket type
  - For raw sockets, the program will see packets from the beginning of its ethernet frame
  - For IP-based TCP/UDP sockets, it will see packets from the start of the transport header
  - For Unix sockets, it will see packets from the start of the data payload
- Additionally, while these programs cannot modify the packets, they can drop them, which breaks `SOCK_STREAM`, but is not `SOCK_DGRAM`
- But, more interestingly, these programs run with the packet is received, regardless of if the process has performed a `recv(2)/read(2)` on the socket FD

Packet Reduce

IPC via the peeking and/or dropping of packets that will never be read

1. Set up a TCP/UDP socket server or a Unix socket server
   - If the socket is stream-based, the userland process should never read from it
   - If it is datagram-based, the userland process may `sendto(2)/recvfrom(2)` the socket
2. Use `setsockopt/SO_ATTACH_BPF` to attach a `BPF_PROG_TYPE_SOCKET_FILTER` program
   - If attached to a stream socket, read all incoming data into an eBPF map shared with userspace
   - If attached to a datagram socket, read all incoming data into an eBPF map shared with userspace and drop all incoming packets
3. At regular intervals, have the userspace program poll the shared eBPF map for messages

Packet Reduce

IPC via the peeking and/or dropping of packets that will never be read

1. Set up a TCP/UDP socket server or a Unix socket server
   - If the socket is stream-based, the userland process should never read from it
   - If it is datagram-based, the userland process may `sendto(2)/recvfrom(2)` the socket
2. Use `setsockopt/SO_ATTACH_BPF` to attach a `BPF_PROG_TYPE_SOCKET_FILTER` program
   - If attached to a stream socket, read all incoming data into an eBPF map shared with userspace
   - If attached to a datagram socket, read all incoming data into an eBPF map shared with userspace and drop all incoming packets
3. At regular intervals, have the userspace program poll the shared eBPF map for messages

- This technique can be used in the reverse direction to prevent packets from being sent while reading the data they contain
- It may also be used by various other privileged eBPF programs that operate on (and can write to!) packets (e.g. XDP, LWT, etc.)

DEMO

eBPF Kernel Tracing Primer

- eBPF supports kernel tracing based on Linux’s kprobe and tracepoint kernel APIs
  - This is restricted to privileged processes and can be abused to compromise entire systems

eBPF Kernel Tracing Primer

- eBPF supports kernel tracing based on Linux's kprobe and tracepoint kernel APIs
    - This is restricted to privileged processes and can be abused to compromise entire systems
- These programs can read arbitrary kernel and userspace memory

eBPF Kernel Tracing Primer

- eBPF supports kernel tracing based on Linux's kprobe and tracepoint kernel APIs
  - This is restricted to privileged processes and can be abused to compromise entire systems
- These programs can read arbitrary kernel and userspace memory
- This presents an interesting opportunity for IPC on a system

eBPF Kernel Tracing Primer

- eBPF supports kernel tracing based on Linux's kprobe and tracepoint kernel APIs
  - This is restricted to privileged processes and can be abused to compromise entire systems
- These programs can read arbitrary kernel and userspace memory
- This presents an interesting opportunity for IPC on a system
- By using the ability to read arbitrary process and kernel memory, tracing eBPF programs can access data that was never explicitly sent to the kernel, and data that will be rejected in-kernel

Whole System Legilimency

IPC via the peeking of data that was never fully sent nor received across system interfaces

- Closed Reading
  - This technique uses an eBPF Kprobe trace on `close(2)`
  - It takes advantage of the fact that `close(2)` will automatically fail on any negative FD
  - By establishing a handshake that fits in file descriptors (`int`) such that the highest bit is 1, an eBPF tracer and colluding processes may agree on locations within memory to read into maps
    - This may involve several calls to `close(2)` as part of a port knocking-like protocol

```
int kprobe__sys_close(struct pt_regs *ctx, int fd) {
  size_t pid_tgid = bpf_get_current_pid_tgid();
  size_t pid = (u32)(pid_tgid >> 32);

  ... // if connection established

  if (fd != handshake_fd) { return 0; }

  ... // save connection state
```

Absolute

(Reliable)

Corruption

eBPF Kernel Tracing Primer (2)

- In addition to reading userland and kernel memory, kernel tracing eBPF programs can also write userland memory
  - Through the `bpf_probe_write_user()` helper function
    Note: Using this helper raises an event in the kernel

eBPF Kernel Tracing Primer (2)

- In addition to reading userland and kernel memory, kernel tracing eBPF programs can also write userland memory
  - Through the `bpf_probe_write_user()` helper function
    Note: Using this helper raises an event in the kernel
  - They can also abort syscalls at entry through `bpf_override_return()`

eBPF Kernel Tracing Primer (2)

- In addition to reading userland and kernel memory, kernel tracing eBPF programs can also write userland memory
  - Through the `bpf_probe_write_user()` helper function
    Note: Using this helper raises an event in the kernel
  - They can also abort syscalls at entry through `bpf_override_return()`
- Most of the interesting data sent in syscalls are pointers to userland memory

nccgroup

eBPF Kernel Tracing Primer (2)

- In addition to reading userland and kernel memory, kernel tracing eBPF programs can also write userland memory
  - Through the `bpf_probe_write_user()` helper function
    Note: Using this helper raises an event in the kernel
  - They can also abort syscalls at entry through `bpf_override_return()`
- Most of the interesting data sent in syscalls are pointers to userland memory
- Therefore, tracing eBPF programs can overwrite string and struct syscall inputs and outputs
  - And prevent syscalls from reaching the kernel

Interdisciplinary Syscall Interdiction

Precise corruption of data transiting a syscall for nefarious purposes

Three main variants:

- Syscall Redirection/Forgery
  - Directing a target process' syscalls "elsewhere"
  - Hijacking a target process' execution context to perform syscalls
  - Useful Targets: `open(2)`, `connect(2)`, `write(2)`, `send*(2)`, `bpf(2)`
- Lying Kernel
  - Providing false data to a process
  - Useful Targets: `*stat(2)`, `read(2)`, `revc*(2)`, `bpf(2)`
- Black Hole
  - Preventing a process from communicating with the outside world
  - Useful Targets: `open(2)`, `connect(2)`, `socket(2)`, `write(2)`, `send*(2)`, `bpf(2)`

nccgroup

Interdisciplinary Syscall Interdiction — Syscall Forgery/Redirection

1. Attach a kernel tracing eBPF program to target syscall entries and exits
   - This program should be configured with target processes and/or inputs to match/replace
     - This can be done through code generation of the program or by passing data in eBPF maps
2. On hooked syscalls, the eBPF program's kprobe will determine if its inputs should be modified
   - If they should not, return
3. Set contextual state (including original inputs) in an eBPF map indexed by PID/TGID
4. Apply the configured modifications
5. On hooked syscall returns, the eBPF program's kretprobe should identify if there is saved contextual state based on the process's PID/TGID/FD
   - If not, return
6. Restore the relevant data to user memory and clear the contextual state for the process from the relevant eBPF map

Interdisciplinary Syscall Interdiction — Lying Kernel

1. Attach a kernel tracing eBPF program to target syscall entries and exits
   - This program should use kprobes as necessary to track state and descriptors for matching
   - It should also be configured with target processes, FDs, and/or outputs to match/replace
     - This can be done through code generation of the program or by passing data in eBPF maps
2. On hooked syscalls, the eBPF program's kprobe will determine if the syscall's results should be modified unconditionally
   - If they should, write the relevant memory and abort the syscall with an appropriate value
3. Set contextual state (including original inputs) in an eBPF map indexed by PID/TGID/FD
4. On hooked syscall returns, the eBPF program's kretprobe should identify if there is saved contextual state based on the process's PID/TGID/FD
   - If not, return
5. Determine if the legitimate syscall output should be modified
   - If not, return
6. Apply the configured modifications
7. Clear the contextual state for the process from the relevant eBPF map

Interdisciplinary Syscall Interdiction — Black Hole

1. Attach a kernel tracing eBPF program to target syscall entries
   - This program should be configured with target processes
     - This can be done through code generation of the program or by passing data in eBPF maps
2. On hooked syscalls, the eBPF program's kprobe will abort the syscall
   - As applicable, it will abort with an appropriate return value
   - It may also write to userspace memory to spoof a successful result before aborting

eBPF Kernel Tracing Primer (3)

- The `bpf_probe_write_user()` helper function has one main limitation

eBPF Kernel Tracing Primer (3)

- The `bpf_probe_write_user()` helper function has one main limitation
  - It cannot write non-writable pages

eBPF Kernel Tracing Primer (3)

- The `bpf_probe_write_user()` helper function has one main limitation
  - It cannot write non-writable pages
- This means that it cannot write to the text or rodata sections
  - At least for properly compiled programs
- This also means that it can only generally write to the stack, heap, and static data sections, which may contain useful targets:
  - Function pointers
  - Saved file descriptors
  - Scripting language textual content
  - Dynamically-generated shell commands

eBPF Kernel Tracing Primer (3)

- The `bpf_probe_write_user()` helper function has one main limitation
  - It cannot write non-writable pages
- This means that it cannot write to the text or rodata sections
  - At least for properly compiled programs
- This also means that it can only generally write to the stack, heap, and static data sections, which may contain useful targets:
  - Function pointers
  - Saved file descriptors
  - Scripting language textual content
  - Dynamically-generated shell commands
- But there is no guarantee that at least one such abusable target will exist across all processes

eBPF Kernel Tracing Primer (3)

- The `bpf_probe_write_user()` helper function has one main limitation
  - It cannot write non-writable pages
- This means that it cannot write to the text or rodata sections
  - At least for properly compiled programs
- This also means that it can only generally write to the stack, heap, and static data sections, which may contain useful targets:
  - Function pointers
  - Saved file descriptors
  - Scripting language textual content
  - Dynamically-generated shell commands
- But there is no guarantee that at least one such abusable target will exist across all processes
- However, all processes have return addresses

At the Stack with Ebert and ROPer

Precise corruption of the stack to inject generic or dynamic ROP chains

- There are several phases to this technique
  0. Payload Pre-Generation (generic ROP chain)
  1. Syscall Selection
  2. Process Filtration
  3. Text Section Identification
     - Attaining Address Spacial Awareness (generic ROP chain)
     - Stack Skimming (dynamically generated ROP chain)
  4. Text Extraction (dynamically generated ROP chain)
  5. Payload Generation (dynamically generated ROP chain)
  6. Stack Skimming Redux
  7. Backup Memory
  8. Payload Injection and Execution
  9. Coordinated Cleanup
- While they do not necessarily need to be followed serially, it is often simpler to do so

At the Stack with Ebert and ROPer — Setup and Target Acquisition

0. Payload Pre-Generation
   1. Find a commonly loaded shared library with useful gadgets
      - For example, glibc has an internal `dlopen(3)` implementation
      - `dlopen(3)` takes only a `char* path` and `int`
      - On success it loads a shared library from the path and will automatically execute its constructors
   2. Scan for gadgets
   3. Assemble a ROP chain
      - By hand or with an automatic ROP chain generator
1. Syscall Selection
   - Register a generic eBPF kprobe on syscalls regularly invoked by target processes
      - If using a generic ROP chain, select only syscalls made by or on behalf of the selected library
2. Process Filtration
   - Within the eBPF kprobe program, profile the processes and their syscalls to prevent further manipulation of unintended targets

At the Stack with Ebert and ROPer — Text Section Identification

3. Attaining Address Spacial Awareness
   1. Within the registered eBPF kprobe program, extract the original instruction pointer register value from the kprobe context
   2. Verify that the memory it references is a valid syscall instruction
   3. Using a pre-computed offset from the syscall instruction, compute the base address of the library
3. Stack Skimming
   1. Within the registered eBPF kprobe program, extract the original stack register value from the kprobe context
   2. Scan the stack for the return address
      1. For each valid offset, determine if the value on the stack is an address into the text section
      2. If so, shift it backwards and attempt to determine if the previous instruction was a call
      3. If so, determine if the call was direct or through a PLT entry
      4. If direct, compute and save the call target and caller addresses
      5. If PLT-based, parse the PLT jump instruction to compute the target, saving it and the caller address
   3. Scan backwards from these text section address to identify the start of their mapped regions

At the Stack with Ebert and ROPer — Text Extraction and Payload Generation

4. Text Extraction
   - Using the base addresses identified in the Stack Skimming step, extract their entire mapped ranges page by page until a page fault it encountered
5. Payload Generation
   - Use a ROP chain generator to create a payload that can load and execute arbitrary code, and finally perform the userland half of the cleanup routine

At the Stack with Ebert and ROPer — Code Execution

6. Stack Skimming Redux
   1. Perform the same steps as the original Stack Skimming operation to obtain the address containing the return address of the syscall stub
   2. Using an eBPF map, store the context of the syscall and return from the kprobe
7. Backup Memory
   1. In the eBPF kretprobe program of the same syscall, validate that the syscall's return is to be overwritten with ROP chain
   2. Back up all memory that will be clobbered by the ROP chain's execution
8. Payload Injection and Execution
   1. Write the ROP chain into the stack starting at the location of the syscall stub's return address
   2. Return from the eBPF kretprobe program
   3. The syscall stub will eventually return to the beginning of the ROP chain
      - This payload should perform desired functionality and then perform the first phase of its cleanup

At the Stack with Ebert and ROPer — Finale

9. Coordinated Cleanup
   1. In the first part of the ROP chain's cleanup routine, the payload should issue a Closed Reading syscall to an eBPF kprobe program
   2. This program, when accessed with a magic value, writes *most of* original stack back
      - However, it should not write over the remaining gadgets in original chain
   3. It should also write the final ROP chain cleanup gadgets past the end of original stack
   4. The Closed Reader eBPF kprobe program should then return
   5. The second part of the ROP chain's cleanup routine will execute, shifting the stack pointer to newly written ROP gadgets implementing the last part of cleanup routine
   6. The final part of the ROP chain's cleanup routine will execute, writing back the original stack values over the last parts of the originally written gadgets
   7. The last remaining gadget will set the return value for the original syscall
   8. Control flow should return back to the caller of the syscall stub

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`
- However, eBPF has another type of tracing program that does not interact with sysfs at all

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`
- However, eBPF has another type of tracing program that does not interact with sysfs at all
- `BPF_PROG_TYPE_RAW_TRACEPOINT`

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`
- However, eBPF has another type of tracing program that does not interact with sysfs at all
- `BPF_PROG_TYPE_RAW_TRACEPOINT`
- To register these programs:

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`
- However, eBPF has another type of tracing program that does not interact with sysfs at all
- `BPF_PROG_TYPE_RAW_TRACEPOINT`
- To register these programs:
  1. Call `bpf(BPF_RAW_TRACEPOINT_OPEN)` with the name of a registered tracepoint event
     - (They are registered in the kernel source with `TRACE_EVENT(...)`)

eBPF Kernel Tracing Primer (4)

- eBPF's k(ret)probe and tracepoint tracing APIs have an additional limitation
  - They use the ftrace sysfs (e.g. `/sys/kernel/debug/tracing`) mount
- This means that they won't work out of the box in Docker containers
  - Because Docker applies an AppArmor profile that restricts access to `/sys/**`
- However, eBPF has another type of tracing program that does not interact with sysfs at all
- `BPF_PROG_TYPE_RAW_TRACEPOINT`
- To register these programs:
  1. Call `bpf(BPF_RAW_TRACEPOINT_OPEN)` with the name of a registered tracepoint event
     - (They are registered in the kernel source with `TRACE_EVENT(...)`)
  2. That's it.

Obie Trice, Raw Tracepoint, No Gimmicks

Use of Raw Tracepoints to Reimplement eBPF Tracing (Post-)Exploitation

- This technique is a slight modification on the previous ones discussed
    - This is necessary as raw tracepoints cannot trace arbitrary syscall events

Obie Trice, Raw Tracepoint, No Gimmicks

Use of Raw Tracepoints to Reimplement eBPF Tracing (Post-)Exploitation

- This technique is a slight modification on the previous ones discussed
  - This is necessary as raw tracepoints cannot trace arbitrary syscall events
- However, they can still trace the two primordial syscall events
  - `raw_syscalls:sys_enter` (receives: syscall ID, userspace registers)
  - `raw_syscalls:sys_exit` (receives: syscall ID, return value)

Obie Trice, Raw Tracepoint, No Gimmicks

Use of Raw Tracepoints to Reimplement eBPF Tracing (Post-)Exploitation

- This technique is a slight modification on the previous ones discussed
  - This is necessary as raw tracepoints cannot trace arbitrary syscall events
- However, they can still trace the two primordial syscall events
  - `raw_syscalls:sys_enter` (receives: syscall ID, userspace registers)
  - `raw_syscalls:sys_exit` (receives: syscall ID, return value)
- Using these, we can re-implement the same hooks
  - By branching on the syscall ID and yanking out syscall arguments from the native registers

Obie Trice, Raw Tracepoint, No Gimmicks

Let's implement a Lying kernel payload!

1. Start with a tracepoint on the `sys_enter` event
2. This function should yank out the syscall ID and use it to determine the course of action
3. For syscalls to hook, persist the syscall context

```
SEC("raw_tracepoint/sys_enter")
int sys_enter_hook(struct bpf_raw_tracepoint_args *ctx) {
  unsigned long syscall_id = ctx->args[1];
  switch (syscall_id) {
    case (0): // read
      save_state(ctx);
      ...
      break;
    case (4): // stat
      save_state(ctx);
      ...
      break;
    default: break;
  }
  return 0;
}
```

Obie Trice, Raw Tracepoint, No Gimmicks

4. Persisting it is essentially the same as described in previous examples
   - Use the PID/TGID as the key to an eBPF map of type BPF_MAP_TYPE_HASH

```
typedef struct serialize {
  uint64_t syscall_id;
  struct pt_regs regs;
} serialize_t;

static inline void save_state(struct bpf_raw_tracepoint_args *ctx) {
  size_t pid_tgid = bpf_get_current_pid_tgid();
  unsigned long syscall_id = ctx->args[1];
  struct pt_regs *regs = (struct pt_regs *)ctx->args[0];
  serialize_t s;
  #pragma unroll
  for (size_t i=0; i < sizeof(s); i++) {
    ((char*)&s)[i] = 0;
  }
  s.syscall_id = syscall_id;
  bpf_probe_read(&s.regs, sizeof(struct pt_regs), regs);
  bpf_map_update_elem(&state_map, &pid_tgid, &s, BPF_ANY);
}
```

Obie Trice, Raw Tracepoint, No Gimmicks

5. In a program with a tracepoint on the `sys_enter` event, we can then pull the state out
6. And check that it matches a syscall we still need to perform operations on
7. If so, we can handle the exit of the syscall that was serialized

```
SEC("raw_tracepoint/sys_exit")
int sys_exit_hook(struct bpf_raw_tracepoint_args *ctx) {
  size_t pid_tgid = bpf_get_current_pid_tgid();
  serialize_t* s = bpf_map_lookup_elem(&state_map, &pid_tgid);
  if (s == NULL) return 0;
  switch (s->syscall_id) {
    case (0): // read
      ...
      break;
    case (4): // stat
      ...
      break;
    default: break;
  }
  bpf_map_delete_elem(&state_map, &pid_tgid);
  return 0;
}
```

DEMO

Defense Against The Dark Arts

Defense Against The Dark Arts

- Remove/Blacklist the `bpf(2)` syscall entirely

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
    - Modern Linux increasingly requires eBPF support, so it will be compiled in

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
  - Modern Linux increasingly requires eBPF support, so it will be compiled in
  - What do you do when someone eventually accesses it?

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
  - Modern Linux increasingly requires eBPF support, so it will be compiled in
  - What do you do when someone eventually accesses it?
- Log all loaded eBPF programs
  - eBPF resources (programs, maps, etc) can be enumerated and dumped by privileged users
    - bpftool is a (really cool) userland utility for accessing the relevant APIs

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
  - Modern Linux increasingly requires eBPF support, so it will be compiled in
  - What do you do when someone eventually accesses it?
- Log all loaded eBPF programs
  - eBPF resources (programs, maps, etc) can be enumerated and dumped by privileged users
    - bpftool is a (really cool) userland utility for accessing the relevant APIs
  - However, this is still susceptible to Lying Kernel attacks on the `bpf(2)` syscall

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
  - Modern Linux increasingly requires eBPF support, so it will be compiled in
  - What do you do when someone eventually accesses it?
- Log all loaded eBPF programs
  - eBPF resources (programs, maps, etc) can be enumerated and dumped by privileged users
    - bpftool is a (really cool) userland utility for accessing the relevant APIs
  - However, this is still susceptible to Lying Kernel attacks on the `bpf(2)` syscall
- Trace syscalls (securely) to detect warning signs
  - Any time eBPF maps are being transferred between processes
  - Any time eBPF maps are not being used with eBPF programs
  - Any time eBPF programs are being transferred between processes
  - Any time an unexpected eBPF program is being attached to TCP/UDP/Unix socket
  - Any time an unrecognized eBPF tracer program is created

Defense Against The Dark Arts

- ~~Remove/Blacklist the `bpf(2)` syscall entirely~~
  - Modern Linux increasingly requires eBPF support, so it will be compiled in
  - What do you do when someone eventually accesses it?
- Log all loaded eBPF programs
  - eBPF resources (programs, maps, etc) can be enumerated and dumped by privileged users
    - bpftool is a (really cool) userland utility for accessing the relevant APIs
  - However, this is still susceptible to Lying Kernel attacks on the `bpf(2)` syscall
- Trace syscalls (securely) to detect warning signs
  - Any time eBPF maps are being transferred between processes
  - Any time eBPF maps are not being used with eBPF programs
  - Any time eBPF programs are being transferred between processes
  - Any time an unexpected eBPF program is being attached to TCP/UDP/Unix socket
  - Any time an unrecognized eBPF tracer program is created
- It's still unclear how much more common these operations will get

Conclusion

- Mo APIs Mo Problems

Conclusion

- Mo APIs Mo Problems
- Even unprivileged eBPF can enable screwy behaviors
- Privileged eBPF is nigh-impossible to stop

Conclusion

- Mo APIs Mo Problems
- Even unprivileged eBPF can enable screwy behaviors
- Privileged eBPF is nigh-impossible to stop
- A good number of eBPF APIs probably don’t need to require crazy privileges
  - If they become unprivileged, they will probably enable some more shenanigans

nccgroup

Conclusion

- Mo APIs Mo Problems
- Even unprivileged eBPF can enable screwy behaviors
- Privileged eBPF is nigh-impossible to stop
- A good number of eBPF APIs probably don't need to require crazy privileges
  - If they become unprivileged, they will probably enable some more shenanigans
- I'm waiting for an eBPF map that can pass arbitrary file descriptors between processes

Greetz

- Andy O
- jkf

You can't hide ~~secrets~~ from the future ~~using math~~

Questions?

jeff.dileo@nccgroup.com
@chaosdatumz

nccgroup

Evil eBPF

Practical Abuses of an In-Kernel Bytecode Runtime

Jeff Dileo
@chaosdatumz

DEF CON 27

nccgroup