Security Monitoring with eBPF
Extended Berkley Packet Filter (eBPF) is a new Linux feature which allows safe and efficient monitoring of kernel functions. This has dramatic implications for security monitoring, especially at Netflix scale. We are encouraging the security community to leverage this new technology to all of our benefit.
There are many security monitoring solutions available today that meet a wide range of requirements. Our design goals were: push vs poll, lightweight, with kernel-level inspection. Our environment is composed of micro-services running on ephemeral and immutable instances built and deployed from source control into a public cloud.

Existing Solutions.

- auditd
- osquery
- ossec
- sysdig
- auditd
A new Option.
<table>
<thead>
<tr>
<th>TIME</th>
<th>UID</th>
<th>PID</th>
<th>COMM</th>
<th>CAP</th>
<th>NAME</th>
<th>AUDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>22:11:23</td>
<td>114</td>
<td>2676</td>
<td>snmpd</td>
<td>12</td>
<td>CAP_NET_ADMIN</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>6990</td>
<td>run</td>
<td>24</td>
<td>CAP_SYS_RESOURCE</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7003</td>
<td>chmod</td>
<td>3</td>
<td>CAP_FOWNER</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7003</td>
<td>chmod</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7005</td>
<td>chmod</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7005</td>
<td>chmod</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7006</td>
<td>chown</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>7006</td>
<td>chown</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>6990</td>
<td>setuidgid</td>
<td>6</td>
<td>CAP_SETGID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>6990</td>
<td>setuidgid</td>
<td>6</td>
<td>CAP_SETGID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:23</td>
<td>0</td>
<td>6990</td>
<td>setuidgid</td>
<td>7</td>
<td>CAP_SETUID</td>
<td>1</td>
</tr>
<tr>
<td>22:11:24</td>
<td>0</td>
<td>7013</td>
<td>run</td>
<td>24</td>
<td>CAP_SYS_RESOURCE</td>
<td>1</td>
</tr>
<tr>
<td>22:11:24</td>
<td>0</td>
<td>7026</td>
<td>chmod</td>
<td>3</td>
<td>CAP_FOWNER</td>
<td>1</td>
</tr>
<tr>
<td>22:11:24</td>
<td>0</td>
<td>7026</td>
<td>chmod</td>
<td>4</td>
<td>CAP_FSETID</td>
<td>1</td>
</tr>
</tbody>
</table>

Snooping on Linux cap_capable() calls using bcc/eBPF
# argdist -i 5 -C 'p::cap_capable():int:ctx->dx'

[06:32:08]

```
p::cap_capable():int:ctx->dx
COUNT   EVENT
  2     ctx->dx = 35
  5     ctx->dx = 21
 83     ctx->dx = 12
```

[06:32:13]

```
p::cap_capable():int:ctx->dx
COUNT   EVENT
  1     ctx->dx = 1
  7     ctx->dx = 21
 82     ctx->dx = 12
```

[...]  

Now frequency counting in-kernel and only sending the summary to user. eBPF is much more than just a per-event tracer (this is a bcc/eBPF hack; I should make this into a real tool like the previous one)
LINUX TRACING
TIMELINE

● 2004: kprobes (2.6.9)
● 2005: DTrace (not Linux); SystemTap (out-of-tree)
● 2008: ftrace (2.6.27)
● 2009: perf_events (2.6.31)
● 2009: tracepoints (2.6.32)
● 2010-2016: ftrace & perf_events enhancements
● 2012: uprobes (3.5)
● 2014-2016: Enhanced BPF patches

+ other out of tree tracers
  LTTng, ktap, sysdig, ...
KERNEL INSTRUMENTATION USING KPROBES
PHRACK ZINE #67/6 2010-11-17

1 - Introduction
   1.1 - Why write it?
   1.2 - About kprobes
   1.3 - Jprobe example
   1.4 - Kretprobe example & Return probe patching technique

2 - Kprobes implementation
   2.1 - Kprobe implementation
   2.2 - Jprobe implementation
   2.3 - File hiding with jprobes
   2.4 - Kretprobe implementation
   2.5 - A quick stop into modifying read-only kernel segments
   2.6 - An idea for a kretprobe implementation for hackers

3 - Patch to unpatch W^X (mprotect/mmap restrictions)

4 - Notes on rootkit detection for kprobes

5 - Summing it all up.

6 - Greetz

7 - References and citations

8 - Code

"So why write this? Because... we are hackers. Hackers should be aware of any and all resources available to them -- some more auspicious than others -- Nonetheless, kprobes are a sweet deal when you consider that they are a native kernel API..."

http://phrack.org/issues/67/6.html
(also see http://phrack.org/issues/63/3.html)
BERKELEY PACKET FILTER

# tcpdump host 127.0.0.1 and port 22 -d
(000) ldh [12]
(001) jeq #0x800    jt 2    jf 18
(002) ld [26]
(003) jeq #0x7f000001   jt 6    jf 4
(004) ld [30]
(005) jeq #0x7f000001   jt 6    jf 18
(006) ldb [23]
(007) jeq #0x84    jt 10   jf 8
(008) jeq #0x6      jt 10   jf 9
(009) jeq #0x11     jt 10   jf 18
(010) ldh [20]
(011) jset #0x1fff    jt 18   jf 12
(012) ldxb 4*[14]&0xf
[...]
There are front-ends (eg, bcc) so we never have to write such raw eBPF

```c
struct bpf_insn prog[] = {
    BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
    BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto*/),
    BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
    BPF_ALUG4_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
    BPF_LD_MAP_FD(BPF_REG_1, map_fd),
    BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
    BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
    BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_dw, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
    BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
    BPF_EXIT_INSN(),
};
```

10 x 64-bit registers maps (hashes) actions

Alexei Starovoitov, 2015+
eBPF USE CASES

User-Defined BPF Programs

- SDN Configuration
- DDoS Mitigation
- Intrusion Detection
- Container Security
- Observability

Kernel

Runtime

BPF

BPF actions

Event Targets

- verifier
- sockets
- kprobes
- uprobes
- tracepoints
- perf_events

...
WHAT TO MONITOR

Trace low-frequency events wherever possible to lower overhead.

Eg, TCP connection init; not TCP send/receive.
These BCC/BPF observability tools show what is possible.
# ./execsnoop -x

<table>
<thead>
<tr>
<th>PCOMM</th>
<th>PID</th>
<th>RET</th>
<th>ARGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>supervise</td>
<td>9661</td>
<td>0</td>
<td>./run</td>
</tr>
<tr>
<td>mkdir</td>
<td>9662</td>
<td>0</td>
<td>/bin/mkdir -p ./main</td>
</tr>
<tr>
<td>run</td>
<td>9663</td>
<td>0</td>
<td>./run</td>
</tr>
<tr>
<td>chown</td>
<td>9664</td>
<td>0</td>
<td>/bin/chown nobody:nobody ./main</td>
</tr>
<tr>
<td>run</td>
<td>9665</td>
<td>0</td>
<td>/bin/mkdir -p ./main</td>
</tr>
<tr>
<td>run</td>
<td>9660</td>
<td>-2</td>
<td>/usr/local/bin/setuidgid nobody</td>
</tr>
</tbody>
</table>

[...]

# ./tcpconnect -t

<table>
<thead>
<tr>
<th>TIME(s)</th>
<th>PID</th>
<th>COMM</th>
<th>IP</th>
<th>SADDR</th>
<th>DADDR</th>
<th>DPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.871</td>
<td>2482</td>
<td>local_agent</td>
<td>4</td>
<td>10.103.219.236</td>
<td>10.251.148.38</td>
<td>7001</td>
</tr>
<tr>
<td>31.874</td>
<td>2482</td>
<td>local_agent</td>
<td>4</td>
<td>10.103.219.236</td>
<td>10.101.3.132</td>
<td>7001</td>
</tr>
<tr>
<td>31.878</td>
<td>2482</td>
<td>local_agent</td>
<td>4</td>
<td>10.103.219.236</td>
<td>10.171.133.98</td>
<td>7101</td>
</tr>
<tr>
<td>90.917</td>
<td>2482</td>
<td>local_agent</td>
<td>4</td>
<td>10.103.219.236</td>
<td>10.251.148.38</td>
<td>7001</td>
</tr>
<tr>
<td>90.928</td>
<td>2482</td>
<td>local_agent</td>
<td>4</td>
<td>10.103.219.236</td>
<td>10.102.64.230</td>
<td>7001</td>
</tr>
</tbody>
</table>

[...]
Use the stable-ist API possible

In order of preference:

**Kernel events**

a. Tracepoints: stable API, if available.
b. Kprobes: dynamic tracing of security hooks
c. Kprobes: dynamic tracing of kernel functions

**User events**

d. User Statically Defined Tracing (USDT) probes: stable API, if available
e. Uprobes: dynamic tracing of API interface functions
f. Uprobes: dynamic tracing of internal functions
WHY eBPF ROCKS

Safe
- Kernel verifies eBPF code (DAG and null reference check)
- Kernel memory access controlled through helper functions
- Part of the mainline kernel, no 3rd party kernel modules

Flexible
- Add new instrumentation to production servers anytime
- Any event, any data

Performant
- JIT’d instrumentation
- Data from kernel to user via async maps or per-events on a ring buffer
- Custom filters and summaries in kernel
- Can choose lower-frequency events to trace

Preliminary results of logging TCP accept() to the file system, with a certain workload, and comparing overheads. Active benchmarking was performed. Each of these can likely be tuned further: results are not final.
Eg, tracing TCP retransmits

**Old way:** packet capture

- tcpdump
  - 1. read
  - 2. dump

- Analyzer
  - 1. read
  - 2. process
  - 3. print

**New way:** dynamic tracing

- Tracer
  - 1. configure
  - 2. read
WRITING A bcc/eBPF PROGRAM

What is in a bcc eBPF Python file:

- Python code for userland reporting
- eBPF C code for event handling, in a variable (or file)
- BCC calls to initialize BPF and probes

```python
# load BPF program
b = BPF(text=""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
BPF_HISTOGRAM(dist);
int kprobe__blk_account_io_completion(struct pt_regs *ctx,
    struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""")

# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print

# output
b["dist"].print_log2_hist("kbytes")
```

bitehist.py example
// pull in details
u16 family = 0, lport = 0;
bpf_probe_read(&family, sizeof(family), &newsk->__sk_common.skc_family);
bpf_probe_read(&lport, sizeof(lport), &newsk->__sk_common.skc_num);

if (family == AF_INET) {
    struct ipv4_data_t data4 = {.pid = pid, .ip = 4};
data4.ts_us = bpf_ktime_get_ns() / 1000;
bpf_probe_read(&data4.saddr, sizeof(u32),
    &newsk->__sk_common.skc_rcv_saddr);
bpf_probe_read(&data4.daddr, sizeof(u32),
    &newsk->__sk_common.skc_daddr);
data4.lport = lport;
bpf_get_current_comm(&data4.task, sizeof(data4.task));
ipv4_events.perf_submit(ctx, &data4, sizeof(data4));
} else if (family == AF_INET6) {

from tcpaccept.py
Summary.
Thank you.
Bonus round.
WHAT’S YOUR SIGN (SYMBOL)

- Example: I want to detect unusual listening ports and what process has bound them.
- Let’s look at the socket lifecycle...
  - socket() is too early, no port yet
  - bind() and listen() are good candidates
  - if access is the only concern, accept()
- We can find kernel symbols a number of ways
  - List them: `sudo cat /proc/kallsyms`
  - Use perf-tools to trace ex. `nc -l 12345`

```
$ sudo cat /proc/kallsyms | grep -E 'inet.*listen'
inet_lookup_listener
inet_hash_nolisten
inet_csk_listen_start
inet_csk_listen_stop
inet_csk_listen
inet_addr
inet_addrinfo
inet_addrinfo6
inet_addrinfo6
```

- `inet_` is the subsystem hooked in BCC examples and seems to have the context we need... but is not guaranteed stable across Linux builds.
Most of the relevant functions we care about are already passing through the LSM (with good context), let’s Kprobe there (if we can’t find a tracepoint) as it will be more stable:

```c
#include/linux/security.h
```

```c
void unix_stream_connect(struct sock *sock, struct sockaddr *addr);

void unix_may_send(struct sock *sock, struct kbytes *kbytes);

int socket_create(int family, int type, int protocol, int kern);

int socket_post_create(struct sock *sock, int family,
                        int type, int protocol, int kern);

int socket_bind(struct sock *sock, struct sockaddr *addr, int len);

int socket_connect(struct sock *sock, struct sockaddr *addr, int len);

int socket_listen(struct sock *sock, int backlog);

int socket_accept(struct sock *sock, struct sock *newsock);

int socket_sendmsg(struct sock *sock, struct kmsg *msg, int size);

int socket_recvmsg(struct sock *sock, struct kmsg *msg, int size, int flags);
```
The end end.