Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing

Soo Yee Lim (UBC), Xueyuan Han (WFU), Thomas Pasquier (UBC)
eBPF Improves Kernel Extensibility

Cilium
eBPF-based Networking, Security, and Observability

Falco
Cloud Native Runtime Security

Katran
High Performance Layer-4 Load balancer

Pixie
Scriptable observability for Kubernetes

These tools *cannot* be used by unprivileged users.
Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion
eBPF ensures safety via the eBPF Verifier
But...

Is the safety of eBPF programs always guaranteed at runtime?
Trick the verifier into trusting a malicious eBPF program.
Implication of eBPF Vulnerabilities

• CVE-2021-3490: The eBPF verifier’s ALU32 bounds tracking for bitwise ops (AND, OR and XOR) did not properly update 32-bit bounds.

What the static verifier believes the program is doing:
Implication of eBPF Vulnerabilities

• CVE-2021-3490: The eBPF verifier’s ALU32 bounds tracking for bitwise ops (AND, OR and XOR) did not properly update 32-bit bounds.

What actually happens during runtime:

Arbitrary read/write in kernel

Overwrite cred struct to escalate privilege

Out-of-bounds accesses

Address space of an eBPF program
Research Problem

The eBPF verifier alone does not guarantee runtime safety.
Content Overview

• Problem
• Related Work
• Our Solution
• Evaluation Results
• Future Work
• Conclusion
Disable Unprivileged eBPF

- Many Linux distributions (e.g., Ubuntu, SUSE) *disable unprivileged eBPF by default* to prevent unprivileged users from exploiting eBPF vulnerabilities.

Unprivileged users cannot use eBPF to customize policies for a particular application or container.
Formally Verifying the eBPF Verifier

• Formally verifying that the eBPF verifier can ensure that it *correctly* implements the specification.

The *size* and *complexity* of the eBPF verifier makes it difficult to formally verify the verifier in its entirety.
The Evolution of the Verifier’s Size

The size of the eBPF verifier has more than doubled in the last four years.

No existing work has managed to formally verify the verifier in its entirety.
Rust-based eBPF

• Rust-based eBPF replaces the eBPF verifier with the Rust tool-chain to perform static checks (e.g., memory safety).

eBPF programs can exploit vulnerabilities in the Rust verifier to violate safety at runtime.
Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion
Threat Model

The kernel is assumed benign and side-channel attacks are considered orthogonal.

Unprivileged adversary capable of exploiting eBPF vulnerabilities to achieve **out-of-bounds access within kernel memory**.
SandBPF: Dynamically Sandboxed eBPF

1. Address Masking (Memory Safety)

2. Redirect Calls to Trampoline (Control Flow Integrity)
SandBPF Is Minimally Invasive

We reuse existing eBPF pipeline and extend only what is necessary.
Memory Safety
A region of memory (the sandbox) is pre-allocated to store the data of an eBPF program.
Address Masking

Consider an invalid memory access at address 0xB123

```
and_mask = 0xFFF; or_mask = 0xA000
```

addr = 0xA000
len = 4096 bytes

All memory accesses always fall within the bounds of an eBPF sandbox.
Control Flow Integrity
Call Capabilities

• At load time, we associate each eBPF program type with a set of capabilities corresponding to the helper functions it is allowed to call.

• The capabilities are stored in a hash table to provide $O(1)$ search time.
Redirect Control Transfers to Trampoline

Without Dynamic Sandboxing

With Dynamic Sandboxing

```
- eBPF Program
  call 0xffffffff12345678 <ebpf_helper>
```

```
- eBPF Program
  call 0xffffffff24681357 <trampoline>
```

```
- Check if the eBPF program has capability for the call target at runtime
  yes
  Proceeds execution at the call target
  no
  Return without transferring control to the intended destination
```

```
Recap: SandBPF

1. Address Masking (Memory Safety)

2. Redirect Calls to Trampoline (Control Flow Integrity)
Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion
## Breakdown of SandBPF Overhead

Table 1: The number of checks inserted and executed by SandBPF in our example programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Injected</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Address Masking</td>
<td>Trampoline</td>
</tr>
<tr>
<td>XDP</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Socket Filter</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Katran</td>
<td>641</td>
<td>42</td>
</tr>
</tbody>
</table>
Breakdown of SandBPF Overhead

Sandbox Management
Constant overhead upon each eBPF invocation.

Memory Access & Control Flow Integrity
Overhead scales with the complexity of eBPF programs.

Figure 5: Breakdown of the overhead introduced by SandBPF
Macro-benchmark

Table 3: Macrobenchmark measuring web server performance of 20-1000 concurrent connections.

<table>
<thead>
<tr>
<th>Test</th>
<th>XDP Program</th>
<th>Socket Filter Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla</td>
<td>SandBPF</td>
</tr>
<tr>
<td>Throughput (request/s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache 20</td>
<td>64,591</td>
<td>64,526 (0%)</td>
</tr>
<tr>
<td>Apache 100</td>
<td>86,190</td>
<td>79,638 (8%)</td>
</tr>
<tr>
<td>Apache 200</td>
<td>85,614</td>
<td>81,749 (5%)</td>
</tr>
<tr>
<td>Apache 500</td>
<td>68,329</td>
<td>63,691 (7%)</td>
</tr>
<tr>
<td>Apache 1000</td>
<td>66,472</td>
<td>62,508 (6%)</td>
</tr>
<tr>
<td>Nginx 20</td>
<td>49,170</td>
<td>45,731 (7%)</td>
</tr>
<tr>
<td>Nginx 100</td>
<td>58,613</td>
<td>54,494 (7%)</td>
</tr>
<tr>
<td>Nginx 200</td>
<td>56,581</td>
<td>53,051 (6%)</td>
</tr>
<tr>
<td>Nginx 500</td>
<td>50,495</td>
<td>47,699 (6%)</td>
</tr>
<tr>
<td>Nginx 1000</td>
<td>46,302</td>
<td>44,977 (3%)</td>
</tr>
</tbody>
</table>

SandBPF incurs no more than 10% overhead in terms of network throughput.
Security Evaluation

• We tested SandBPF against CVE-2021-3490 and CVE-2021-4204:
  • Both results in arbitrary read/write in the kernel.
  • Both can be exploited to escalate privileges.

SandBPF successfully prevents both vulnerabilities.
Content Overview

- Problem
- Related Work
- Our Solution
- Evaluation Results
- Future Work
- Conclusion
Future Work

• Optimizing Performance
  • We see $\leq 10\%$ overhead on network throughput.
  • This is without any optimization to SandBPF (e.g., asynchrony).
  • We see this as a reasonable baseline for future work to improve performance.

• Simplify the eBPF verifier
  • Remove some constraints on eBPF program expressiveness.
Content Overview

• Problem
• Related Work
• Our Solution
• Evaluation Results
• Future Work
• Conclusion
Unprivileged eBPF for Better Kernel Extensibility

• Dynamic sandboxing is a viable approach to enforce security properties in eBPF programs, *complementary* to the current static mechanism employed by the eBPF verifier.

• SandBPF enhances runtime safety of the kernel to justify the (currently dismissed) support of unprivileged eBPF programs.
Thank you! Any Questions?

Soo Yee Lim (sooyee@cs.ubc.ca)

Joint work with Xueyuan Han and Thomas Pasquier