Two-Phase eBPF Program Signing

Linux Kernel Maintainer
Cong Wang, xiyou.wang@gmail.com
https://wangcong.org/about/

LSF/MM/BPF 2025, March 25, 2025

The Evolution of eBPF Security

- The Linux kernel has implemented various eBPF safeguards:
  - eBPF verifier performs static analysis to ensure kernel safety
  - Privilege restrictions requiring `CAP_BPF` capability
  - BPF tokens for fine-grained control of capabilities
  - LSM hooks for additional security controls

Why Traditional Signing Doesn't Work

- eBPF programs undergo necessary modifications during loading
- These modifications invalidate traditional signatures
- Major modifications include:
  - Updating BPF map file descriptors
  - Patching relocations
  - Runtime JIT adjustments (program size, offsets, etc.)

The Catch-22 Situation

- If we sign the original binary:
  - Signature becomes invalid after `libbpf`'s modifications
- If we sign after modifications:
  - We lose the ability to verify the program's original authenticity

In-kernel eBPF Program Loader

- Multiple proposals to move the eBPF program loader into the kernel
- Critical issues with this approach:
  - Adds complex code to privileged kernel space
  - Compatibility challenges with different kernel versions
  - Reduced flexibility compared to user-space loading
  - Increased verification complexity
- Essentially user-space vs kernel-space debate

Two-Phase eBPF Program Signing

- Mirrors the eBPF program preparation and loading process
- Like a legal document requiring both:
  - Initial notarization
  - Subsequent verification of modifications
- Leverages existing eBPF infrastructure

Phase 1: The Baseline Signature

- Generated when the eBPF program is initially compiled
- PKCS#7 signature for the original, unmodified program
- Serves as proof that the original program came from a trusted source
- Analogous to getting a document notarized before filling in details

Phase 2: The Modified Program Signature

- Created after `libbpf` has made necessary modifications
- New signature covering both:
  - The modified program
  - Its original signature
- Establishes a chain of trust
- Proves modifications were authorized and applied to legitimate code

Verification Process

1. Kernel verifies original program against baseline signature
2. Then verifies secondary signature covering both:
    - Modified program
    - Original signature

Guarantees

- Program originated from a trusted source
- Modifications were authorized
- Chain of trust remains unbroken

Advantages

- No kernel modifications required
  - Built on existing eBPF infrastructure
  - Uses standard BPF LSM hooks and kfuncs
- Strong auditability
  - Precise tracing of failures
  - Clear audit trails for security investigations
- Highly customizable
  - No need to upstream
  - Finer granularity of control
    - e.g. if you want to skip signing for `bpftrace`

Disadvantages

- The signing eBPF program must be trusted
  - Hard to integrate with secure boot?
- It also must be loaded as early as possible
  - But we can place it in initramfs
  - Or built into the kernel (not yet supported)
- Distribution of the private key is a challenge
  - Not specific to this proposal
  - Same for DKMS signed modules

Proof of Concept

- PKCS#7 signatures for both phases
- BPF LSM hooks to intercept program loading
- Standard cryptographic primitives from OpenSSL
- Leverages existing Linux kernel keyring infrastructure
- Available as open source (proof of concept)
  github.com/congwang/ebpf-2-phase-signing

Building eBPF Programs into the Kernel

- The signing eBPF program is part of the trusted kernel image
- Eliminates the security concerns of dynamic loading
- Challenges:
  - No libbpf loader to use?
  - How to verify the program? What if it fails to verify?
  - Less flexibility for rapid deployment
  - Limited to programs that don't need runtime configuration

Possible Solution

- Compile the signing eBPF program into the kernel image
- Parse the ELF section which contains the program
- Call `call_usermodehelper()` to load and attach the program from user-space
- Panic if the program fails to verify
- Use `memfd` to load the program from memory

Pseudo Code

```
static int __init bpf_signing_init(void)
{
    int ret;
    char *argv[] = {
        "/sbin/bpftool",
        "prog",
        "load",
        "/boot/signing_prog.o",
        "pinned",
        "/sys/fs/bpf/signing_prog",
        "type",
        "lsm",
        NULL
    };
    char *envp[] = {
        "HOME=/",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
        NULL
    };

    ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
    if (ret < 0) {
        panic("Failed to load eBPF signing program: %d\n", ret);
    }

    ret = call_usermodehelper("/sbin/bpftool",
        (char *[]){ "/sbin/bpftool", "prog", "attach", "pinned",
                    "/sys/fs/bpf/signing_prog", "lsm", NULL },
        envp, UMH_WAIT_PROC);
    if (ret < 0) {
        panic("Failed to attach eBPF signing program: %d\n", ret);
    }

    pr_info("eBPF signing program loaded successfully\n");
    return 0;
}

early_initcall(bpf_signing_init); /* After bpf subsystem init */
```

Thank You!

Questions?

Contact: Cong Wang xiyou.wangcong@gmail.com

https://wangcong.org/about/