

# BPF Signing: What's next

KP Singh

## Why sign programs?

### **Provenance (The "Who")**

Confirm exactly who built the binary and protect the integrity of the binary

### **Policy Enforcement**

Based on provenance and signing entity

## Where are we today?

Programs with stable instruction buffers can be signed directly and pass the signature and verification keyring in the bpf syscall.

Programs with relocations use a signed loader program , `BPF_PROG_TYPE_SYSCALL`, with stable instruction buffer and a metadata map.

The hash of the metadata map is embedded in the loader program and the signed loader then verifies the payload of the map.

Exclusive maps were introduced to prevent modifications to the metadata map by other programs.

## The Ultimate Policy

All programs must be signed

```
if (!attr->signature)  
    return -EPERM;
```

## The Ultimate Policy: Limitations

If simply applied, breaks key use-cases with dynamic program generation e.g. cilium and bpftrace.

Dynamic relocations need light skeletons which need to catch up with skeletons. Not everyone likes skeletons

**Policy needs to be more nuanced**

# Trusted Loaders: dm-verity

An LSM program, loaded and pinned at early boot checks if the loader is loaded from a trusted dm-verity partition by fetching the digest (`bpf_get_dm_verity_digest`, downstream kfunc)

Comparing just the digest is fine as kernel ensures dm-verity signature verification at boot. Can potentially be extended to verifying the signature against the digest with the BPF signing key (eliminating the need for a digest manifest)

Suitable for systems with minimal / locked down root filesystems, However, If not done carefully, can lead to trusting swiss army knife loaders like `bpf tool`



## Trusted Loaders: fs-verity / IMA

A package manager signs the loader off-host and stores the signature in an extended attribute (e.g. `user.bpf_prog_sig`)

The trusted is deployed on an fs-verity enabled partition with the extended attribute and fs-verity is enabled on the target system

A BPF LSM program allows unsigned loading if the signature in the `xattr` verifies against the digest (provided by IMA or fs-verity).

## Trusted Loaders: Delegated Offline Signing

A delegated entity certificate pre-signed by the root key is created and baked into the image.

The delegated certificate is loaded into a keychain. Access to the keychain is restricted to trusted loaders.

Loaders sign the programs with this key, and request verification with the appropriate keyring ID passed in the syscall.

# Trusted Loaders: Delegated Online Signing

A delegated entity certificate is created on the the client host and the verification at runtime.

The client sends a request to a signing service which authenticates and authorizes request based on an application TLS certificate (e.g. Kubernetes credential)

The client (e.g. Cilium) signs the programs with the delegated certificate, loads the key into a keychain for verification and destroys the private key ensuring no further programs can be signed with the certificate

## Safe Programs: bypass CAP\_\* checks

Don't require capability checks for certain signed programs. However, requires restructuring of capability checks into a single place that can be hooked into for additional policy enforcement.

```
bool bpf_capable(union bpf_attr *attr, int cap) {  
    return security_bpf_capable(attr, cap) || capable(cap);  
}
```

Re-triggers discussions around capabilities, maybe worth it?

## Safe Programs: A path to unprivileged eBPF

Ever since side channel attacks were discovered and continue to be, the unprivileged eBPF goal fizzled away.

Unprivileged users should be able to load certain signed programs without requiring elevated privileges.

Better user-experience, helps with principle of least privilege and even more relevant as use-cases for BPF grow (e.g. sched\_ext)

## Safe Programs: BPF Token

Tokens are powerful and allow finer grained control with the gatekeeping policy being encoded in the filesystem delegation

Currently only built for user namespaces and no way to use them without running in a user-namespace.

Systemd service with a socket to grant tokens, library code to load programs wrapped in a user namespace.

# What am I doing?

Helping with a reference implementation to understand the gaps

<https://github.com/sinkap/bpf-trust.git>

This an early prototype hackspace for verifying ideas and brainstorming