

shirudo: BPF live-patching infra for the agentic era

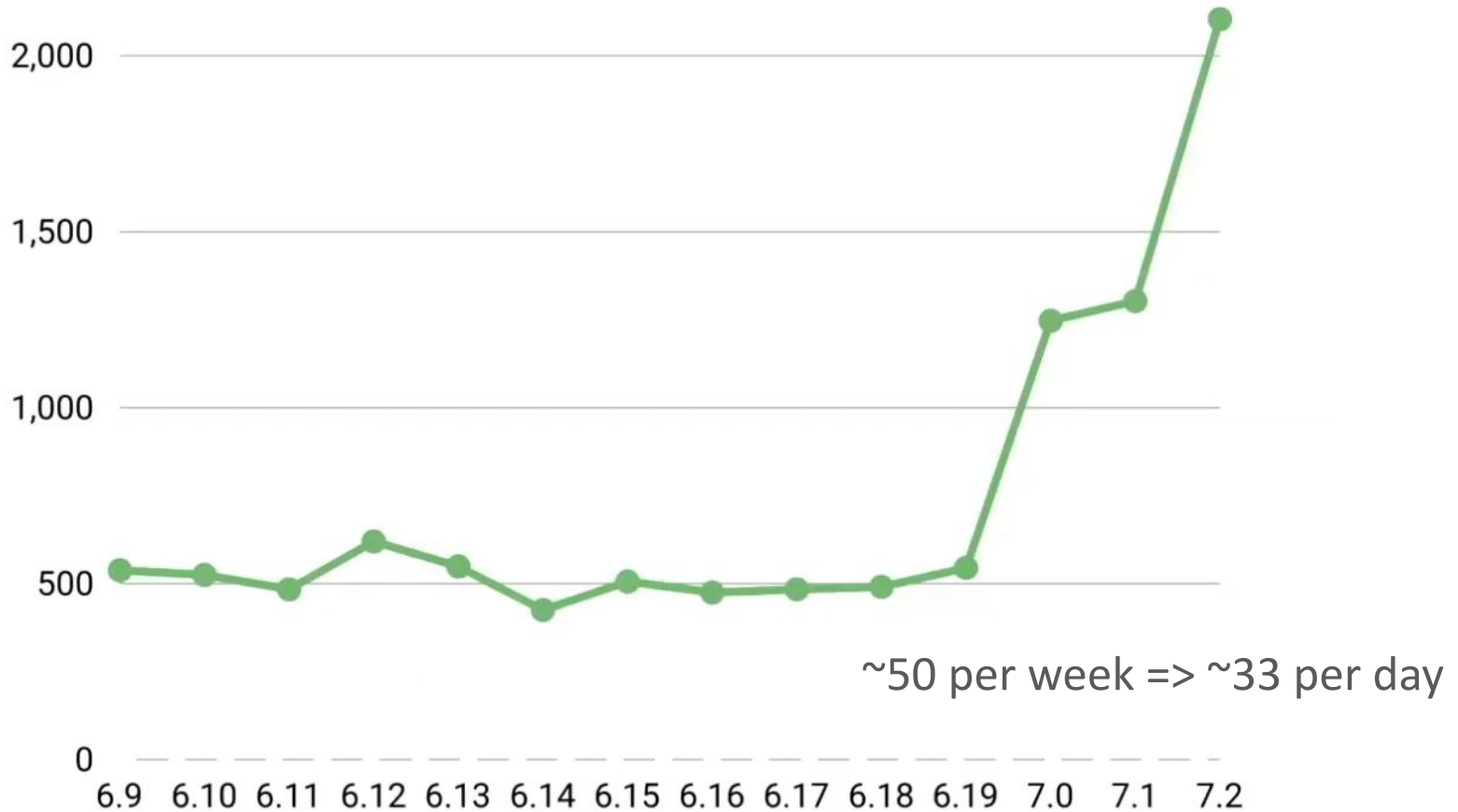
John Fastabend

Daniel Borkmann

Michi Mutsuzaki

CVE fixes per release

(GKH talk @ kernel-recipes '26)





Name

- patched kernel
- bpf-lsm mitigated
- unmitigated (not found)

May'26

(CopyFail CVE)

Public
Disclosure

Patch
Developed

Servers
Patched

04/29 20:00

04/30 00:00

04/30 04:00

04/30 08:00

04/30 12:00

04/30 16:00

04/30 20:00

05/01 00:00

05/01 04:00

05/01 08:00

05/01 12:00

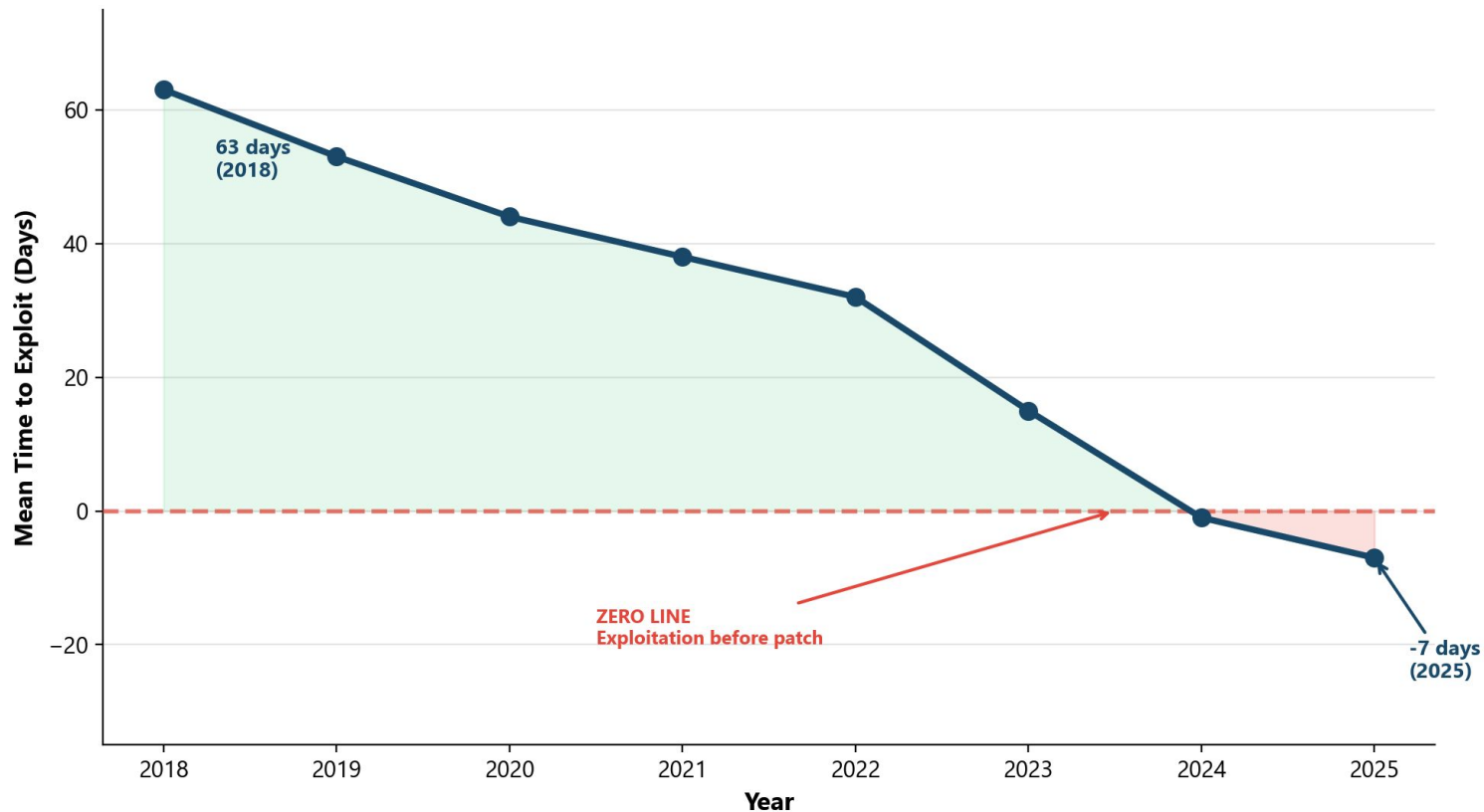
05/01 16:00

05/01 20:00

CF blog #1: Following a successful release, the Edge Reboot Release (ERR) pipeline manages a systematic update and reboot of the edge infrastructure on a four-week cycle.

CF blog #2: The mitigation via BPF LSM took 26 to 30 hours to fully mitigate CopyFail across the fleet.

Mean Time to Exploit Has Gone Negative



Source: Mandiant M-Trends Reports (2019-2026)

Deploy-cycle was built for human (slow) adversary, not LLMs.

Hugging Face: [...] Like the earlier events, IM1 agents found new ways to chain together several novel security flaws to gain greater access to our infrastructure and reach the broader Internet. [...]

Some thoughts...

- The sheer amount of fix churn is not going to end any time soon
- Distro model of “enabling every possible Kconfig” for max compatibility is broken
- Distro model of deviating from upstream stable kernels is broken
- Just rolling out new kernel patch releases across fleet not feasible anymore
- Time-to-mitigate critical bug of 24hrs is too long, ideally asap / in order of <<hrs

Idea back at LSF/MM/BPF

#1 Massive hardening needed to remove surface

#2 Mitigations need to be created at the speed of agents

#3 Consider BPF security part of the TCB

Properties for a new BPF security platform

Agentless

- BPF security is part of the TCB “always on” launched from early boot
- BPF security is signed via BPF keyring and sealed resilient to malicious root users
- BPF control plane lives in kernel so any operations are signature-verified
- TPM signature chain maintains history of all operations

Properties for a new BPF security platform

Packaging

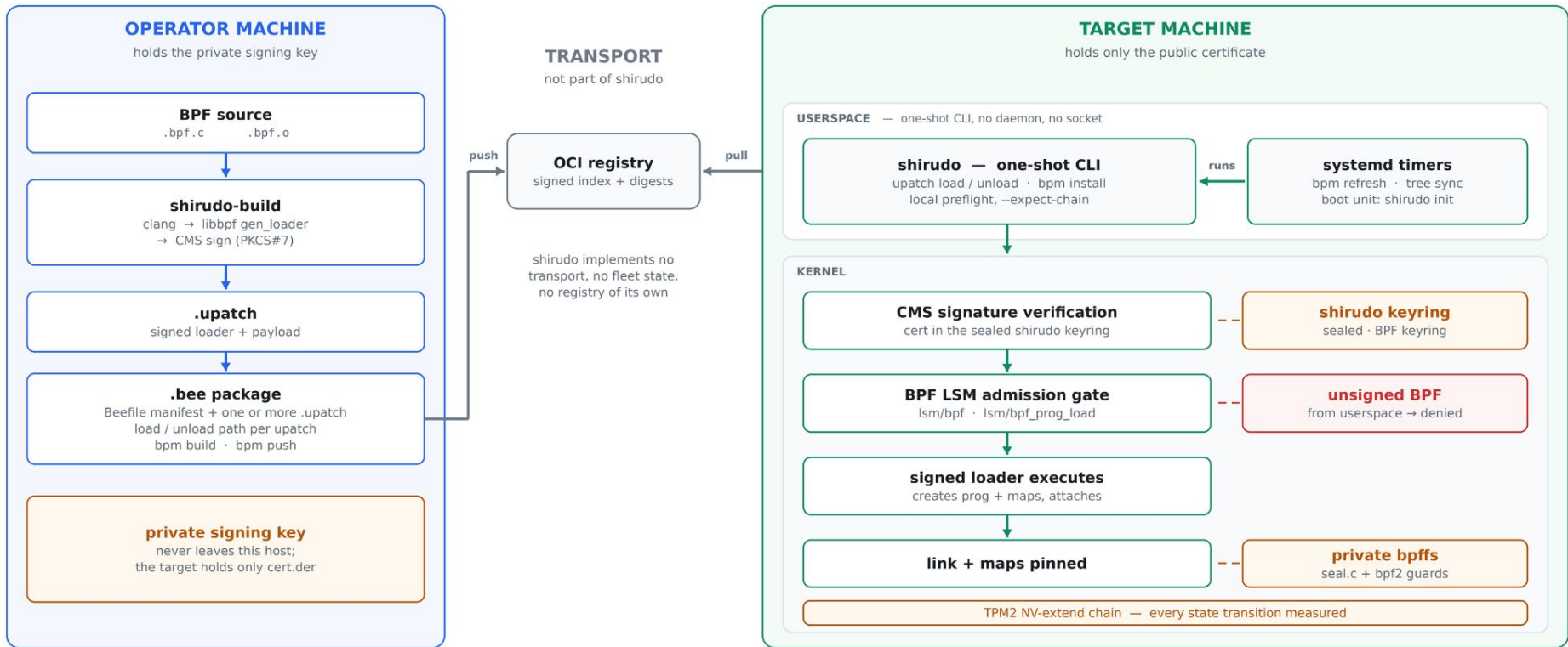
- BPF security platform itself only protects its assets: functionality shipped as BPF upatch
- BPF upatches are blobs containing signed BPF lightweight skeletons (loader + assets)
- In an agentic age we *ship code* not DSLs/APIs to reduce contextual barriers
- Agentless platform itself *pulls* BPF upatches to remove any open port dependency
- Agentless platform itself *pushes* telemetry data to external collector

Properties for a new BPF security platform

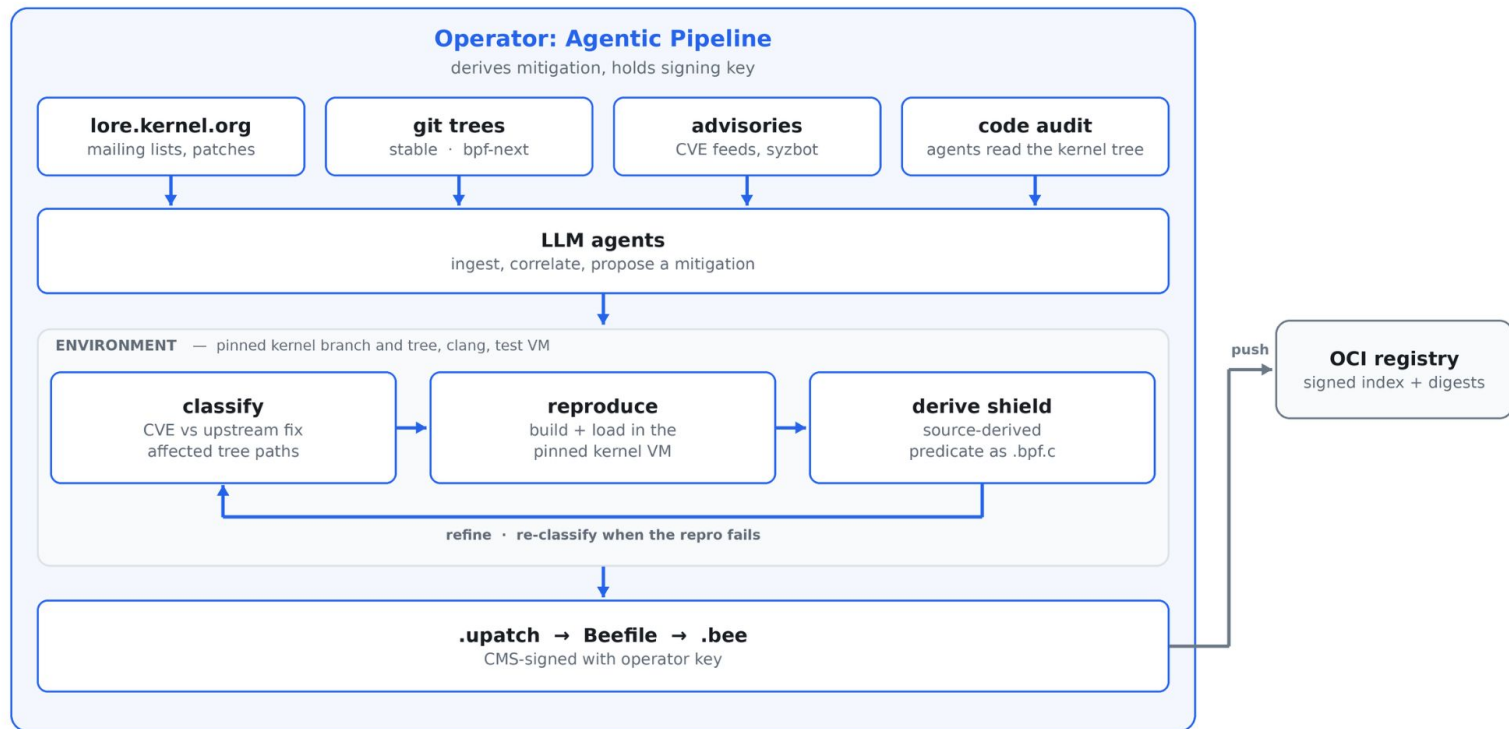
Agentic

- Structured input/output to allow agents reading and interacting with data
- Agentic loop: Ingest data (e.g. lore git), generate test cases, craft BPF mitigations
- Source code set of small gadgets so agents can maintain context
- Outcome: Operator-signed BPF upatches which can then be pushed to OCI and deployed (pulled) into the fleet

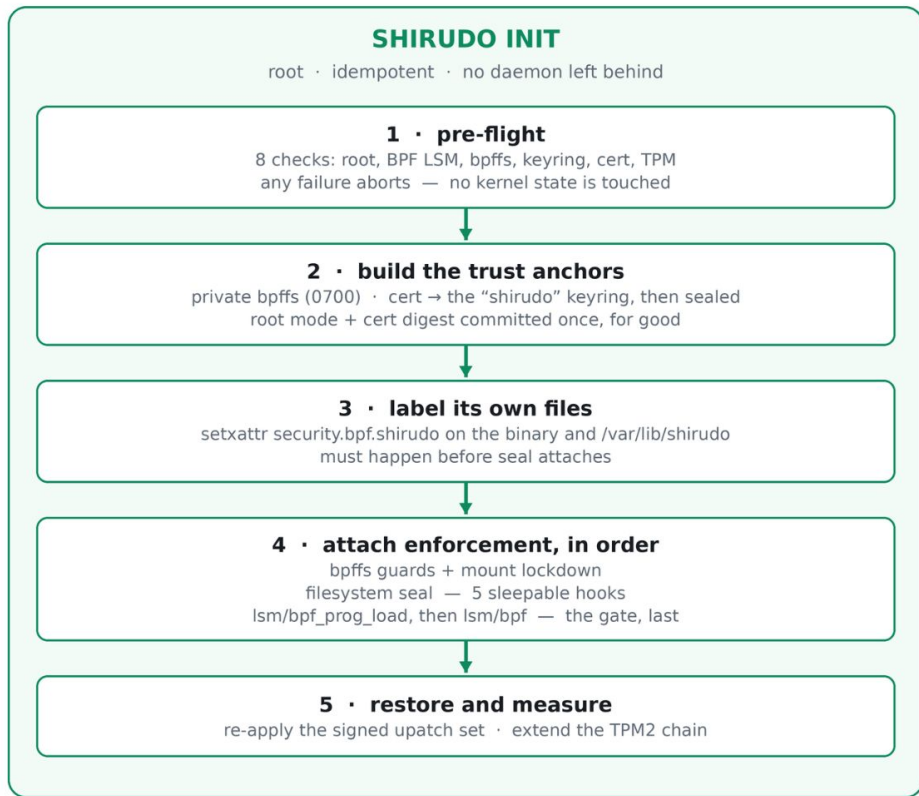
shirudo: Operator/target architecture



shirudo: Pipeline for window between deploy-cycles



shirudo: Bootstrap process



Bootstrap must bring up **BPF LSM** which then gates all bpf(2) interaction from root/non-root. The shirudo BPF LSM program also must protect all its assets from untrusted root via security.bpf.shirudo **xattrs**.

Once done, all upatches get applied.

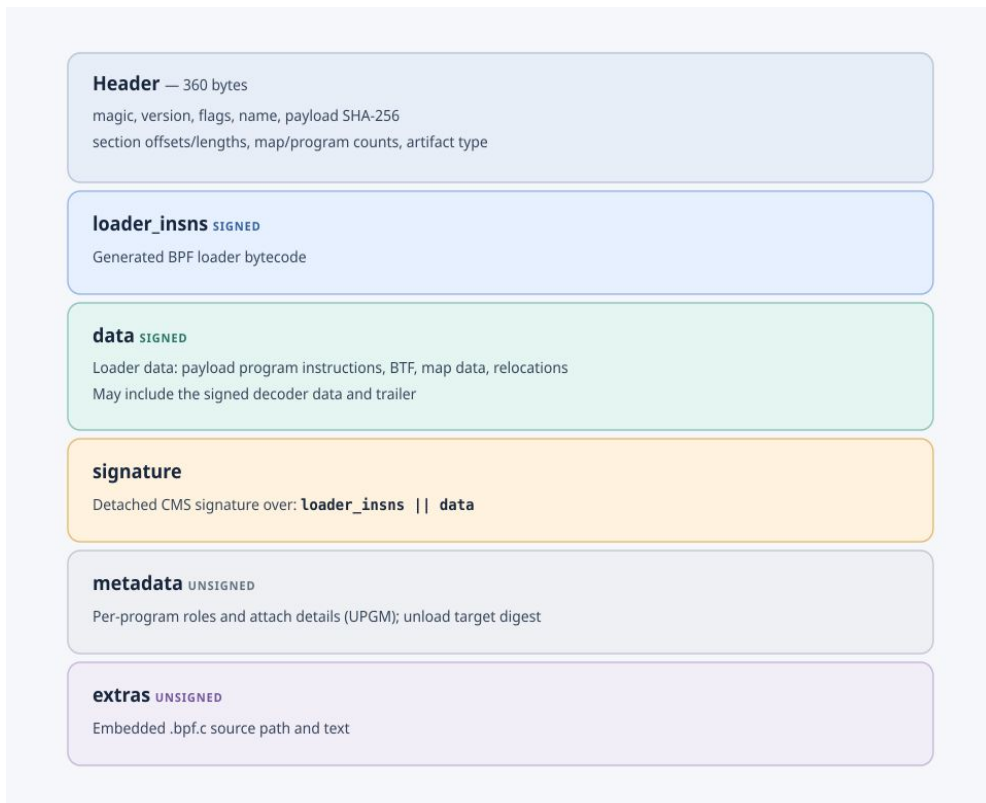
shirudo: Asset sealing

security.bpf.shirudo xattr

- Set on shirudo binary itself
- Set on all assets (including upatches) under /var/lib/shirudo
- Set on all active BPF links under /var/lib/shirudo
- BPF LSM to only allow security.bpf.shirudo -> security.bpf.shirudo access
- BPF LSM denies (re-)labeling of security.bpf.shirudo from non-shirudo apps
- Thanks to xattrs, there is no need to specifically deal with paths, symlinks, hardlinks, ...

shirudo: .upatch

A **signed** set of related **BPF** programs, **maps**, and **metadata**. Typically represents a signal functionality, such as network observability or CVE “shield”.



shirudo: Beefile Manifest

Beefile: Describe files needed to build a upatch pkg. The pkg includes manifest, upatch and any collateral files needed. For testing/debug we often pkg reproducer files with the BPF.

Gitops workflow: Tag version triggers Beefile build and publish to OCI/S3/artifactory. Target nodes can automatically pull updates using **Tree Manifest**.

```
[package]
name = "bpf-observability"
description = "Shirudo observability BPF upatches"
license = "GPL-2.0 OR BSD-2-Clause"
authors = ["Isovalent"]

[[files]]
source = "linux-observability.bpf.c"

[[files]]
source = "exec_uid_arena.bpf.c"

[[files]]
source = "exec_uid_common.h"

[[files]]
source = "dns_arena.bpf.c"

[[files]]
source = "decoders/exec.lua"

[[upatches]]
source = "linux-observability.bpf.c"
```

shirudo: Tree Manifest

Given a set of upatches the target will need to know order to load them, rollback, update. For this we create an append only tree with ops list with selectors.

```
```json
{
 "schema": 2,
 "tree_id": "production",
 "generation": 8,
 "parent_manifest_digest": "sha256:...",
 "base": {"seq": 0},
 "operations": [
 {
 "seq": 1,
 "op": "add",
 "id": "block-afalg",
 "selector": {
 "kernel_release_prefix": ["6.16", "6.18"],
 "kernel_build_id": ["<GNU-vmlinux-build-id>"],
 "architecture": ["x86_64"],
 "kconfig": {"CONFIG_BPF": ["y"]},
 "labels": {"env": ["prod"]}
 },
 "artifact": {
 "repository": "oci://quay.io/example/shirudo-upatch-tree",
 "manifest_digest": "sha256:...",
 "layer_digest": "sha256:...",
 "payload_sha256": "sha256:..."
 },
 "operation_digest": "sha256:..."
 }
],
 "head": {"seq": 1}
}
```
```

shirudo: Tree Manifest

Operations either add/delete upatch

Selector block tells node if upatch applies to system or not. <build-id> is critical for kprobe+offset upatch.

Artifact tells us where to pull the upatch from along with digest.

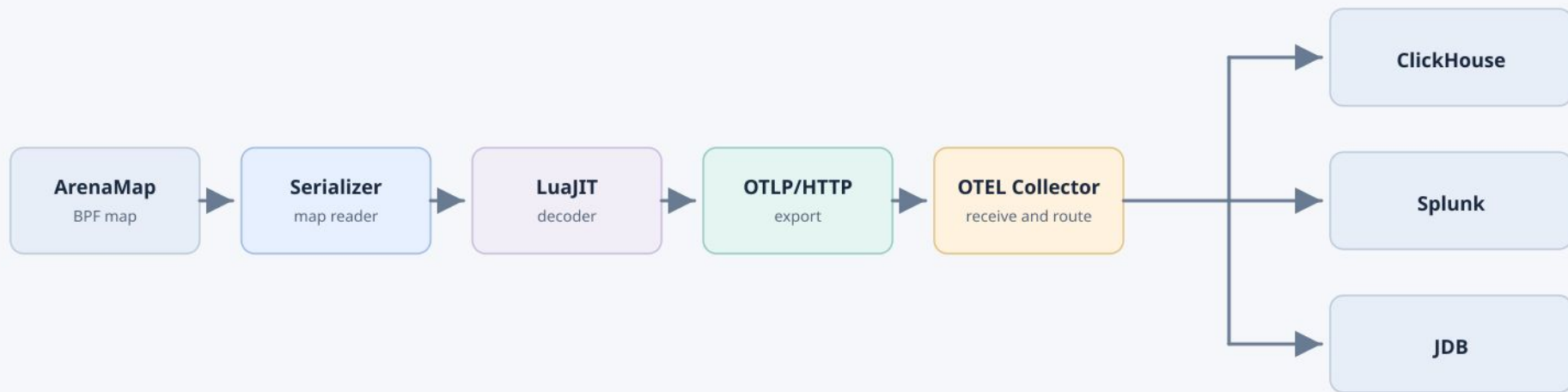
Every new op has an increasing seq number. Every target node has its current “seq” number and checks tree to ensure it is up to date.

```
```json
{
 "schema": 2,
 "tree_id": "production",
 "generation": 8,
 "parent_manifest_digest": "sha256:...",
 "base": {"seq": 0},
 "operations": [
 {
 "seq": 1,
 "op": "add",
 "id": "block-afalg",
 "selector": {
 "kernel_release_prefix": ["6.16", "6.18"],
 "kernel_build_id": ["<GNU-vmlinux-build-id>"],
 "architecture": ["x86_64"],
 "kconfig": {"CONFIG_BPF": ["y"]},
 "labels": {"env": ["prod"]}
 },
 "artifact": {
 "repository": "oci://quay.io/example/shirudo-upatch-tree",
 "manifest_digest": "sha256:...",
 "layer_digest": "sha256:...",
 "payload_sha256": "sha256:..."
 },
 "operation_digest": "sha256:..."
 }
],
 "head": {"seq": 1}
}
```

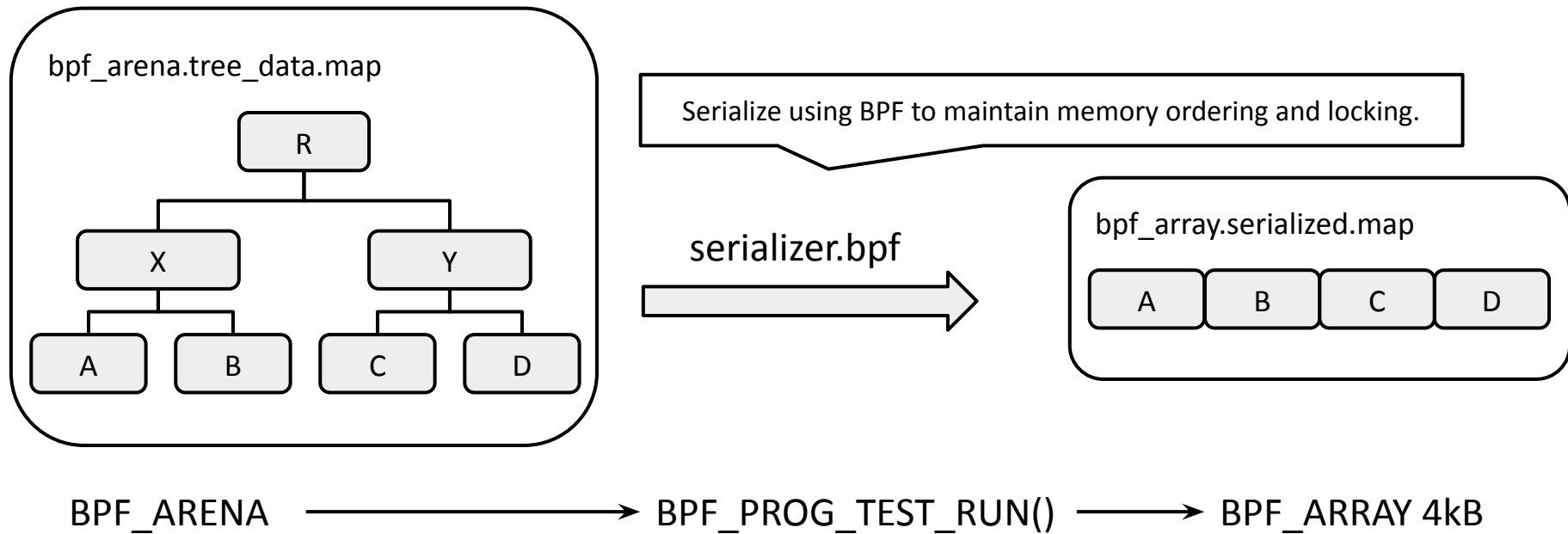
# shirudo: TPMs

- Everything must be measurable, there are no “silent” operations on upatches
- Every state transition (init, load, unload) extends the measurement chain
- Chain mismatch at pre-flight causes operation to be refused, thus nothing loaded and node considered tainted
- Append-only eventlog alongside the chain for offline replay/audit purpose

# Export: Architecture/Pipeline



# Export: Serializer



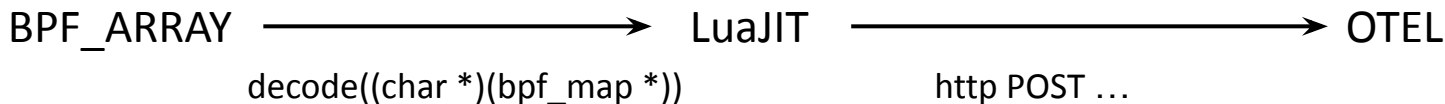
# Export: LuaJIT



LuaJIT provides extra flexibility to encode into OTEL.

```
function decode(input)
 if not input.data.serialized or input.data.serialized_bytes ~= 32 then
 error("unexpected IPv4 shields counter payload")
 end
 local records = shirudo.fixed_records(input.data.pages, 16)
 -- fixed_records reserves Lua slot 1 for the serialized page header.
 local record = records[2]
 if not record then
 error("missing IPv4 shields counter record")
 end
 return {
 {name = "enforcement.ipv4_shields_metric", unit = "1",
 description = "IPv4 TCP connect and UDP send decisions",
 sum = {
 aggregationTemporality = 2,
 isMonotonic = true,
 dataPoints = {
 metric_point(input, record:u64(0), "allowed"),
 metric_point(input, record:u64(8), "denied"),
 },
 }},
 }
end

~
"shields/linux_obs/decoders/ipv4_shields.lua" 34L, 965B
```



# shirudo: First experimental results

Initial experiment focused on LLM generated detectors for **489 CVEs** that met requirements (high severity, reproducible) scan from June '26 - Sep '26.

Outcome	Count
Loaded, passed health check, and reproducers	317
Load Failed: Hook function inlined or otherwise missing	22
Load Failed: Verifier rejected, unresolved	34
Load Failed: Attach rejected	7
Load Failed: Missing module	89
BPF gen failed: Source reference unknown	20

## Future Work:

Of the **317** it is an open question how many can be translated into mitigations. Currently, **30** verified mitigations, but often requires widening scope resulting in increasingly more aggressive side effects. Requires human judgment to decide value.

**Example:** CopyFail bug can be mitigated by blocking socket create. But observed much deeper in the stack.

# Recent kernel additions in this context

## **Signed BPF loader rework**

- In-kernel BPF program acting as loader, loading actual program from data map
- Light skeleton program (bpftool gen skeleton -L)
- Signature over loader and assets `Sig(l_loader || D_meta)` or `Sig(l_loader)`
- ML-DSA support (2029 Google's cryptography timeline for PQC)
- Admission via `lsm/bpf_prog_load` BPF LSM hook
- Gate based on keyring, key serial, verdict

# Recent kernel additions in this context

## BPF example

```
SEC("lsm/bpf_prog_load")
int BPF_PROG(admit, struct bpf_prog *prog, union bpf_attr *attr,
 struct bpf_token *token, bool kernel)
{
 __u32 verdict = prog->aux->sig.verdict;
 __u32 ktype = prog->aux->sig.keyring_type;
 __s32 serial = prog->aux->sig.keyring_serial;
 struct decision *d;
 int ret = 0;

 if (kernel)
 return 0; /* trust in-kernel loads */

 if (verdict != BPF_SIG_VERIFIED) {
 ret = -EPERM; /* must be validly signed */
 } else switch (ktype) {
 case BPF_SIG_KEYRING_BPF:
 break;
 case BPF_SIG_KEYRING_USER:
 if (!bpf_map_lookup_elem(&staging_keys, &serial))
 ret = -EPERM;
 break;
 default:
 ret = -EPERM; /* keyring not in policy */
 }

 d = bpf_ringbuf_reserve(&audit, sizeof(*d), 0);
 if (d) {
 d->prog_type = attr->prog_type;
 d->verdict = verdict;
 d->ktype = ktype;
 d->serial = serial;
 d->ret = ret;
 bpf_ringbuf_submit(d, 0); /* record allow *and* deny */
 }

 return ret;
}
```

BPF keyring (for normal operation mode)

User/session key (for staging mode)

# Recent kernel additions in this context

## BPF keyring

- Existing keyrings: builtin/secondary trusted keyring, platform keyring, user/session key
- New BPF keyring specifically for signed BPF loaders (for one or multiple keys)
- Sealed empty upon boot by default meaning no enrollment possible similar to dm-verity
- `bpf.keyring_unsealed=1` boot option for enrollment, systemd enrolls BPF key and seals it

```
for key in /etc/bpf/keys/*.der; do
 keyctl padd asymmetric "" -9 < $key
done
```

```
keyctl restrict_keyring -9
keyctl show -9
```

 Enrollment example at boot

# Recent kernel additions in this context

## BPF in-kernel sockets

- Needed for agentless and in-kernel “control plane”, see [Mahe’s talk](#)
- Use cases range from keepalive, shirudo status and event/alert logging

```
SEC("lsm.s/socket_bind")
int BPF_PROG(ksock_socket_bind, struct socket *sock,
 struct sockaddr *address, int addrlen, int ret)
{
 struct bpf_ksock *ks;
 u32 pid = bpf_get_current_pid_tgid() >> 32;

 if (ret || pid != target_pid)
 return ret;

 ks = ksock_ctx_get();
 if (!ks) {
 send_ret = -ENOENT;
 return ret;
 }

 send_ret = bpf_ksock_send(ks, send_data, sizeof(send_data));
 bpf_ksock_release(ks);

 return ret;
}
```

Sending void\* data buffer



**BPF example**

# Recent kernel additions in this context

## **BPF filesystem xattr support**

- Serves trusted.\* and security.\* domains; user.\* not supported
- Store is in-memory and non-persistent
- BPF LSM or user space tooling can then uniformly look at objects (e.g. security.bpf.\*) and enforce policy
- Example: Only allow access/removal of BPF link from BPF filesystem with security.bpf.shirudo if app has security.bpf.shirudo label

# Recent kernel additions in this context

## **BPF xattr inode atomic init support**

- Closes an important gap in BPF LSM to propagate security.bpf.\* xattr labels race-free
- Example: directory has security.bpf.shirudo, within directory we create a new directory (or other file system object). BPF can then specify to propagate security.bpf.shirudo label on the new object before it is published

# Next items: Expanding BPF xattr support

## **BPF xattr getter/setter buildout**

- dentry: bpf\_{set,get,remove}\_dentry\_xattr for security.bpf.\* xattr labels already there
- file: bpf\_{set,get,remove}\_file\_xattr for security.bpf.\* xattr labels to complete
- bpf\_{set,get}\_kernfs\_xattr for security.bpf.\* xattr labels (e.g. for cgroups v2)
- Other misc additions to close corner cases

# Next items: Gaps in BPF syscall program's abilities

## **BPF signed loader next steps**

- Today, the signed loader loads the program and passes fd back to user space
- Signed loader should also create the BPF link, and pin the BPF object in BPF fs, such that all this is covered by the signature
- Same for in-kernel BPF link detach/unload, so that full lifecycle is complete
- Also, support for all BPF programs needed (e.g. struct\_ops)

# Next items: fmod\_ret expansion via kernel build

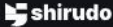
## Use case for building shirudo BPF upatches

- Needed for observability, mitigations and hardening kernel surface, see [Antons's talk](#)

```
bpftool fmodret | head -4321 | tail -7; echo
1:123105 cxl_cper_sec_prot_err_valid (vmlinux)
1:123106 cxl_cper_setup_prot_err_work_data (vmlinux)
1:123121 cxl_reset_bus_function (vmlinux)
1:123143 d3cold_allowed_store (vmlinux)
1:123174 d_mark_tmpfile_name (vmlinux)
1:123186 d_set_mounted (vmlinux)
1:123210 da903x_probe (vmlinux)
```

```
bpftool fmodret | wc -l
18073
```

# Demo

 JBRUST CONTROL ROOM  
Project default

Stack ready Refresh

PANE 01

Inventory

1 VMs

Tracked runners, telemetry health, and loaded upatches.

● cve-6-18

6.18

HEALTHY

running

PANE 02

Shields

4 shields

Build, sign, publish, and roll out protection packages.

✦ bpf-observability

linux\_obs/Beefile

v0.1.0

PUBLISHED

Build

Deploy

✦ java-cve-upatches

java-cve/Beefile

v0.1.0

PUBLISHED

Build

Deploy

✦ java-sql-injection

java\_sql\_inj/Beefile

v0.1.0

PUBLISHED

Build

Deploy

✦ kernel-cve-upatches

kernel-cve/Beefile

v0.1.0

PUBLISHED

Build

Deploy

 **shirudo**

**ISOVALENT**  
now part of **cisco**