Userland-execve

Running Linux executables without `execve`

Rafael Junio da Cruz — Presenter
Shawn Chang (citypw) — Co-author
eCrime 2026

About me

Cryptography and systems-security engineer with 10+ years of experience across offensive research and low-level security engineering.

Applied cryptography and hardware security

- Mathematics and cryptographic engineering for constrained systems
- Side-channel and fault-injection analysis
- Embedded platforms, ARM, and trusted-execution environments

Currently: Critical TechWorks (BMW Group)

Mobile and offensive security

- Security research and penetration testing across the Android stack
- Threat modeling and exploit-mitigation work at Samsung
- Contributor to Samsung Message Guard

HardenedLinux: holistic security research

HardenedLinux is a community for practical, defense-in-depth security across free software, firmware, and hardware.

Community

- Launched by Shawn Chang (citypw) in 2014.
- Covers GNU/Linux, mobile, Internet of Things (IoT), firmware, and hardware.

Intel ME research

- 2016: minimized Intel Management Engine (ME) firmware on Sandy and Ivy Bridge.
- Used external Serial Peripheral Interface (SPI) flash programming for analysis.



The question

Modern Linux defenses commonly assume executable code:

- comes from a **file-backed source**;
- enters through `execve()`; and
- is checked at that boundary.

> **What happens when a process builds a new ELF image in memory instead?**

Why this matters

Control	What it normally protects
SELinux	Memory permissions and executable mappings
`noexec`	Binaries stored on writable file systems
dm-verity	Integrity of protected on-disk content
seccomp	Access to selected system calls

All are valuable controls. This work examines an execution path that does not use the usual file-to-process transition.

Why this matters for drones and robots

In cyber-physical systems, software decisions can directly affect motion, sensing, and safety.

Drones

- Flight limits, altitude limits, and geofencing are frequently software-enforced.
- Compromise of a flight, mission, or companion process can turn an execution-path issue into a safety issue.

Robotics

- Robots similarly enforce motion limits, safety zones, and operator separation in software.
- Monitoring must recognize behavior changes even when the process identity does not change.

This presentation does not claim to demonstrate userland-execve on flight or safety controllers. The relevance is the post-compromise execution and provenance gap.

Drone and robot remote code execution (RCE): safety consequence

DJI research — Schiller *et al.*

- Analyzed DJI drone and remote-control hardware, firmware, and wireless interfaces.
- Reported **16 vulnerabilities** in four of five tested devices; **14** were remotely triggerable through the operator's smartphone.
- Reported arbitrary command execution and crashes during flight.

Unitree G1 proof of concept (PoC) — UniBLEed

- The UniBLEed repository documents RCE chains affecting the Unitree G1 robot.
- It identifies Bluetooth Low Energy (BLE), Web Real-Time Communication (WebRTC), Data Distribution Service (DDS), and service interfaces in the attack surface.
- Its active tests can disrupt services and alter network configuration.

> **In safety-relevant systems, execution provenance is a safety property.**

Userland-execve in one sentence

An existing process can reproduce essential Executable and Linkable Format (ELF) loader work in user space, then transfer control to the loaded program **without calling** `execve()`.

The kernel continues to see

- the original process identifier (PID);
- credentials and SELinux context;
- the original executable identity.

The process begins to run

- a new ELF image;
- a rebuilt stack and auxiliary vector;
- the target program or dynamic linker entry point.

Threat model

The technique **does not provide initial access or privilege escalation**.

It assumes an attacker already controls code inside an unprivileged process, for example through:

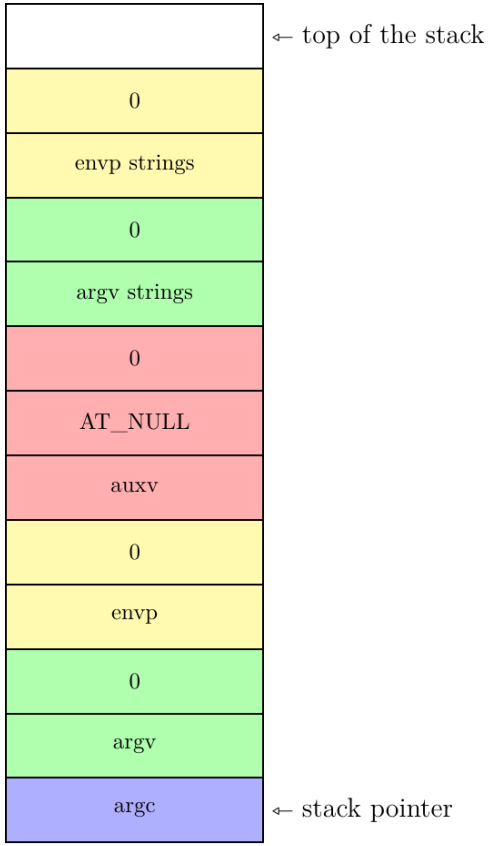
- a memory-safety flaw;
- script or supply-chain injection; or
- a compromised application component.

The attacker cannot write protected file systems or modify the kernel.

What `execve()` normally does

1. Replace the current address space.
2. Map the executable and, when needed, its dynamic linker.
3. Build the initial stack.
4. Populate the auxiliary vector.
5. Jump to the appropriate entry point.

Userland-execve reproduces the relevant steps with ordinary user-space primitives.



The loader: four phases

1. Parse ELF

Read and validate headers and loadable segments.

2. Build an in-memory file

Materialize the image in a `memfd`.

3. Rebuild mappings

Map loadable segments at their intended addresses.

4. Rebuild startup state

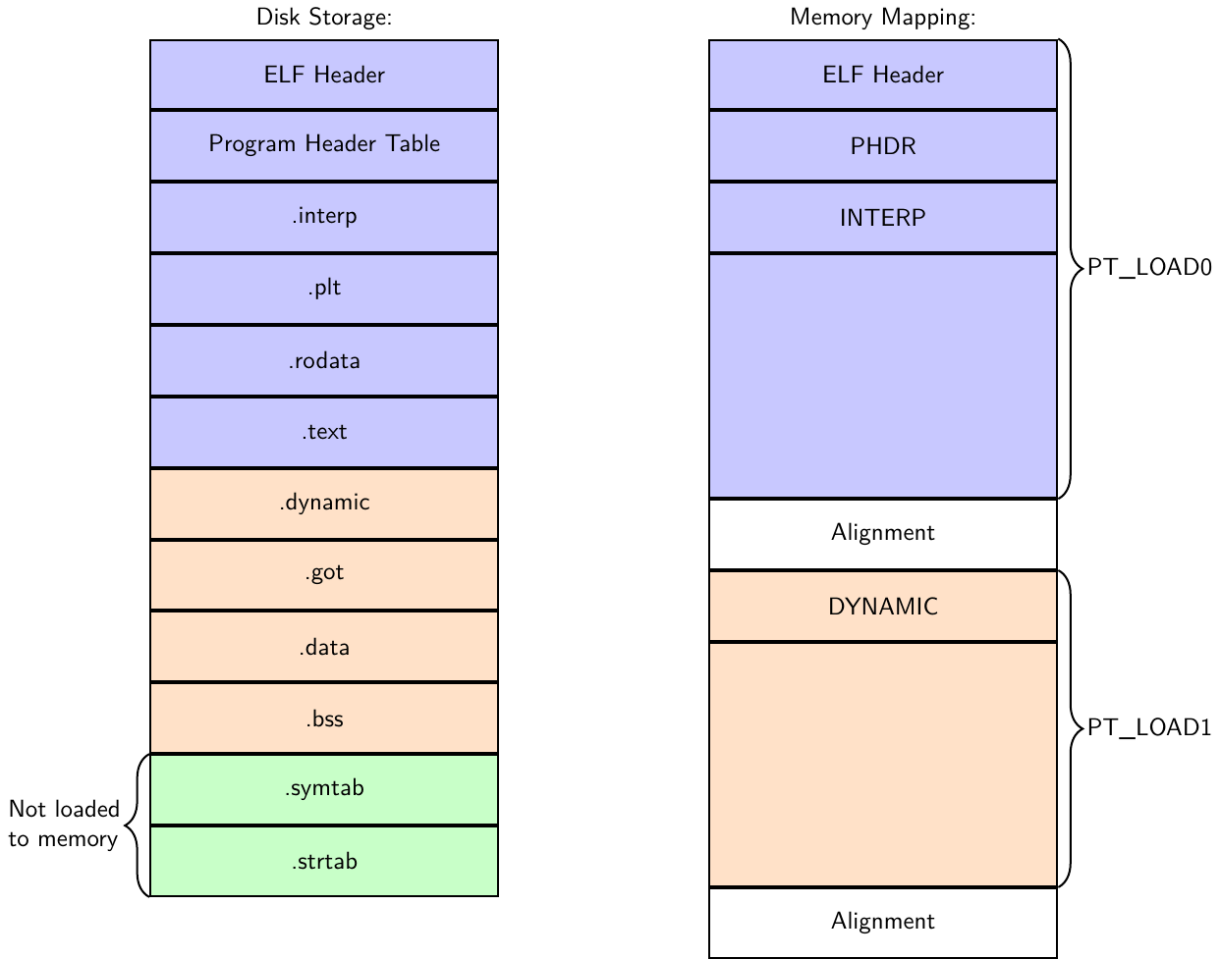
Create a fresh stack, synthesize auxv, and transfer control.

1. Parse the ELF as data

- Open the target through a file descriptor.
- Read it using `mmap(PROT_READ)`.
- Validate ELF type, program headers, entry point, and interpreter.
- Locate each `PT_LOAD` segment.

No code executes from the original mapping. The file is treated only as **input data**.

ELF: on disk vs. in memory



2. Build an in-memory image

`memfd_create()` provides an anonymous, RAM-backed file descriptor.

- Copy each loadable ELF segment to the required offset.
- Zero-fill memory that belongs to a segment but is not stored in the file.
- Use the resulting `memfd` as the mapping source.

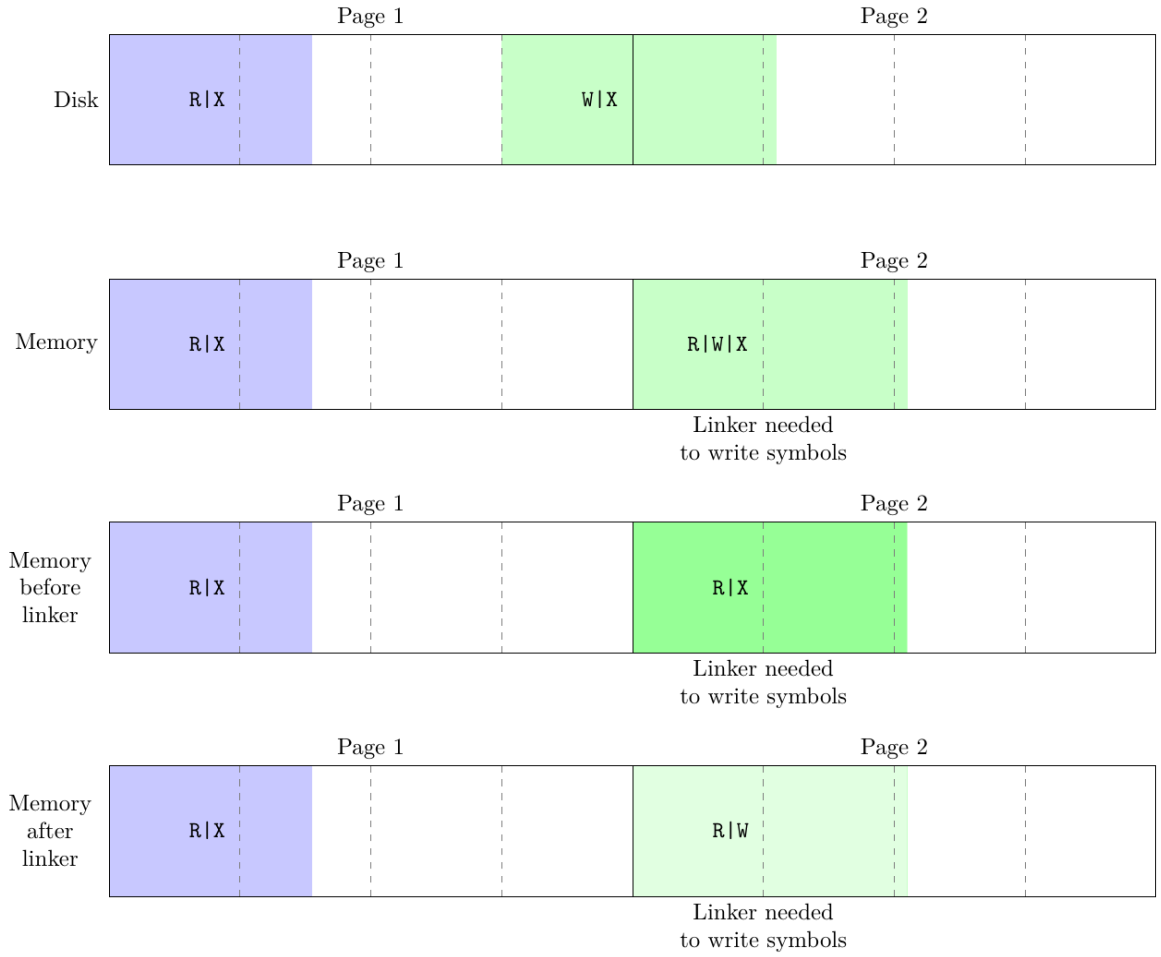
This changes **where the executable mappings come from**: they are no longer backed by the original path on disk.

3. Rebuild the address space

- Map every `PT_LOAD` segment at its expected virtual address.
- Request permissions derived from the ELF segment flags.
- Keep executable pages **readable and executable**, never writable at the same time.
- Handle writes through separate copy-on-write-style pages where needed.

> **The design aims to preserve the kernel-visible W^X separation.**

Overlapping ELF mappings



4. Rebuild startup state

The target expects the stack layout normally created by the kernel:

- `argc`, `argv`, and `envp`;
- a valid auxiliary vector (`AT_PHDR`, `AT_ENTRY`, `AT_BASE`, and more);
- a correctly aligned stack pointer.

For dynamically linked ELF files, control first passes to the dynamic linker. For static binaries, it passes to the program entry point.

Linux source: SELinux mapping decisions

`file_map_prot_check`

```
if (default_noexec && (prot & PROT_EXEC) &&
    (!file || IS_PRIVATE(file_inode(file)) ||
     (!shared && (prot & PROT_WRITE)))) {
    rc = avc_has_perm(..., SECCLASS_PROCESS,
                      PROCESS__EXECMEM, NULL);
}

if (file && (prot & PROT_EXEC))
    av |= FILE__EXECUTE;
```

Meaning: executable requests that are anonymous or writable-private require `execmem`; executable file mappings also require permission to execute the backing file.

`selinux_file_mprotect`

```
if (default_noexec && (prot & PROT_EXEC) &&
    !(vma->vm_flags & VM_EXEC)) {
    if (vma->vm_start >= mm->start_brk &&
        vma->vm_end <= mm->brk)
        rc = avc_has_perm(..., PROCESS__EXECHEAP, ...);
    else if (!vma->vm_file && vma_is_stack_for_current(vma))
        rc = avc_has_perm(..., PROCESS__EXECSTACK, ...);
    else if (vma->vm_file && vma->anon_vma)
        rc = file_has_perm(..., FILE__EXECMOD);
}
return file_map_prot_check(...);
```

Meaning: when `mprotect()` adds execute access, SELinux separately checks heap (`execheap`), stack (`execstack`), and modified file mappings (`execmod`).

Condensed from Linux `security/selinux/hooks.c`. Evaluate the exact kernel release and deployed SELinux policy; the source shows the checks, not a universal outcome.

Example: bypassing `noexec`

`noexec` prevents a direct launch:

```
execve("/tmp/payload") → EACCES
```

With userland-execve, the file is opened as **data**, rebuilt into a `memfd`, and mapped from memory. The original `/tmp` file is not executed directly.

A controlled proof of concept uses a `noexec,nosuid,nodev` tmpfs to demonstrate this distinction. It evaluates a post-compromise execution primitive, not initial exploitation. Research article

Android and constrained systems

The approach was tested on **Android 14 / ARM64** in enforcing mode.

- Application data directories are normally `noexec`.
- The payload runs within the host app's existing process and SELinux domain.
- Restrictive seccomp profiles can block required primitives such as `memfd_create` or mapping operations.

This makes syscall filtering a practical mitigation layer for high-risk processes.

Portability and cost

Platform	Architecture	Result
Ubuntu 24.04, Arch, CentOS	x86_64	Successful
Raspberry Pi OS	ARM64	Successful
Samsung S23, Android 14	ARM64	Successful

- Core loader: approximately **800 lines of C** plus small architecture-specific startup code.
- Measured overhead: roughly **2–3 ms** for a small dynamic binary and **8–10 ms** for an 800 KB static binary.

Detection opportunities

Look beyond `execve`:

- executable mappings backed by `memfd` or anonymous sources;
- a process that creates a `memfd`, writes ELF-like data, then maps it executable;
- divergence between `/proc/<pid>/exe` and the active executable mappings;
- abrupt changes in a long-lived process's behavior.

Runtime monitoring should correlate these signals rather than depend on a single event.

Mitigations

Kernel and policy

- Treat sensitive executable mappings as security-relevant events.
- Audit executable `memfd` and anonymous mappings.
- Limit executable mappings in untrusted SELinux domains.

Process sandboxing

- Use seccomp to deny unnecessary primitives, including `memfd_create`.
- Apply hardened-kernel controls where available.
- Design Endpoint Detection and Response (EDR) and Runtime Application Self-Protection (RASP) for memory-backed execution paths.

Policies must balance these restrictions with legitimate just-in-time (JIT), shared-memory, and runtime use cases.

Hardening beyond `noexec`

PaX / grsecurity

- Trusted Path Execution (TPE) and Role-Based Access Control (RBAC) can constrain execution from untrusted paths and shared-memory-like mappings.
- The cited threat-model write-up reports these controls effective against its tested `memfd` paths.
- `noexec` is not a complete boundary once an already-authorized interpreter or Foreign Function Interface (FFI)-enabled process executes native code.

Upstream Linux: `mseal`

- `mseal()` seals existing virtual-memory mappings against `mprotect`, `munmap`, `mremap`, and `MAP_FIXED` changes.
- It is useful after initialization to protect fixed code and read-only regions.
- It is defense in depth; combine it with Linux Security Module (LSM) policy and seccomp.

Userland-exec: a post-exploitation component

After code execution, an in-process loader can become a reusable **post-exploitation framework component**: load native modules without a new process or an `execve` transition.

- Reusable layers: delivery, module loading, capability selection, and operator control.
- In drones and robots, modules can affect mission, sensor, network, or control-adjacent services.
- Process-creation, executable-path, and `execve` telemetry alone can miss this activity.

> **Limit post-compromise blast radius with least privilege, service isolation, seccomp, mapping policy, and behavior-aware monitoring.**

Limits

- No exploit delivery or privilege escalation.
- Requires a native, Application Binary Interface (ABI)-compatible ELF target.
- Cannot overcome a restrictive seccomp policy that blocks required operations.
- Depends on a compatible ELF ABI and dynamic linker.
- Memory maps can reveal the technique to capable monitoring.

This is a **post-compromise execution technique**, not a replacement for every execution path.

Takeaway

> **File-centric controls remain essential - but they are not complete execution-provenance controls.**

Defenses should treat **memory-backed executable mappings** as first-class security events, with policy, auditing, and behavioral monitoring designed around them.

Thank you

Rafael Junio da Cruz
rafa.junio@hardenedlinux.org
github.com/hardenedlinux/userland-exec

Co-author Security researcher: Shawn Chang (citypw) · Research: “Bypassing File-System and MAC-Based Security Assumptions with Userland Execution Techniques” · eCrime 2026

References

- Project code: https://github.com/hardenedlinux/userland-exec
- Controlled `noexec` proof of concept: https://hardenedlinux.org/blog/2026-04-13-stealthy-rce-on-hardened-linux-noexec--userland-execution-poc/
- Rapid7, *Santa's Elfs: Running Linux Executables Without Execve* https://www.rapid7.com/blog/post/2019/01/03/santas-elfs-running-linux-executables-without-execve/
- Linux documentation: `memfd_create(2)`, dm-verity, and seccomp(2)
- Android Verified Boot: https://source.android.com/security/verifiedboot
- N. Schiller et al., *Drone Security and the Mysterious Case of DJI's DroneID*, NDSS 2023: https://doi.org/10.14722/ndss.2023.24217
- Olivier Laflamme, *UniBLEed* Unitree G1 research repository: https://github.com/OlivierLaflamme/UniBLEed
- PaX/grsecurity and FFI/noexec discussion: https://github.com/hardenedlinux/grsecurity-101-tutorials/blob/master/threat_model/userland_exec_noexec_bypass.md
- Upstream Linux memory sealing: https://docs.kernel.org/userspace-api/mseal.html
- HardenedLinux community and Intel ME research: https://hardenedlinux.org/about/; https://hardenedlinux.org/blog/2016-11-17-neutralize-me-firmware-on-sandybridge-and-ivybridge-platforms/

Backup: likely questions - scope

Does this exploit a kernel vulnerability?

No. It is a post-compromise execution technique that uses ordinary user-space interfaces after code execution is already obtained.

Does it provide privilege escalation?

No. The loaded program retains the process's existing credentials, SELinux domain, capabilities, and sandbox constraints.

Is this the same as `fexecve()`?

No. `fexecve()` still asks the kernel to create an execution transition. This technique reconstructs the required process state in user space.

Backup: likely questions - defense

Does `noexec` still help?

Yes. It blocks direct execution from writable mounts, but it is not a complete control against a process that is already compromised.

Is SELinux ineffective?

No. The result depends on the active policy and requested mappings. Strong policies, syscall filtering, and sandboxing remain valuable layers.

What should defenders monitor?

Correlate `memfd` creation, ELF-like writes, executable mappings, signal-handler changes, and behavior inconsistent with the process identity.