CITS3007 Secure Coding
Copy Fail (CVE-2026-31431) - safe things combining unsafely

Unit coordinator: Arran Stewart

Copy Fail

- A major kernel vulnerability, reported April 29
- A 732-byte Python script, no root required, no race conditions
- Any unprivileged local user -> root on essentially every major Linux distro
- Works very reliably, every time
- Corrupts the in-memory image of any readable file – *without* touching the file on disk
- (Checking file contents will show it as unchanged)

Copy Fail

- A major kernel vulnerability, reported April 29
- A 732-byte Python script, no root required, no race conditions
- Any unprivileged local user -> root on essentially every major Linux distro
- Works very reliably, every time
- Corrupts the in-memory image of any readable file – *without* touching the file on disk
- (Checking file contents will show it as unchanged)

How does it do this?

Individual “ingredients”

Each feature is individually reasonable, but dangerous when combined

Page cache – the kernel caches file contents in memory, shared system-wide

`splice()` – zero-copy I/O: passes references to those cached pages between kernel subsystems, rather than copying

`AF_ALG` – a socket interface exposing the kernel's cryptographic routines to unprivileged userspace

Ingredient 1 – page cache

- When you read a file, you usually get an in-memory **cached copy** – not a fresh read from disk
- That cached copy is shared across the whole system – i.e., all processes, all containers on the same host
- Under normal circumstances, userspace cannot write directly to these pages
- If something *can* write there: the on-disk file is untouched, `sha256sum` reports the correct hash – but any reads, or calls to e.g. `execve()` load the corrupted version

Ingredient 1 – page cache

- When you read a file, you usually get an in-memory **cached copy** – not a fresh read from disk
- That cached copy is shared across the whole system – i.e., all processes, all containers on the same host
- Under normal circumstances, userspace cannot write directly to these pages
- If something *can* write there: the on-disk file is untouched, `sha256sum` reports the correct hash – but any reads, or calls to e.g. `execve()` load the corrupted version

Developers not familiar with the page cache can be surprised that "the file" in memory and "the file" on disk can differ.

Ingredient 2 – `splice()`

Ordinary I/O copies data twice on the way out:

```
page cache  —>(read)—>  user buffer  —>(write)—>  socket buffer
```

`splice()` skips the copy – passes a *reference* to the page cache page:

```
page cache  —>(splice)—>  pipe  —>(splice)—>  socket buffer
                ^
                |
          same physical page —— no copy made
```

Designed for performance (e.g. static file web servers).
Side effect: you can hand a page cache page by reference to another kernel subsystem.

Ingredient 3 – AF_ALG

Exposes the kernel's crypto engine to unprivileged userspace via sockets. The API is fairly unusual, to say the least:

```
sock = socket(AF_ALG, SOCK_SEQPACKET, 0)
sock.bind(("aead", "authencesn(hmac(sha256),cbc(aes))"))
op_sock, _ = sock.accept()   # not a network connection
op_sock.sendmsg([data], ...)
result = op_sock.recv(n)
```

- `bind()` selects an *algorithm*, not an address.
- `accept()` creates an "operation handle", not a connection. Basically, an opaque "thing" that lets us later use a particular operation

Ingredient 3 – AF_ALG

Exposes the kernel's crypto engine to unprivileged userspace via sockets. The API is fairly unusual, to say the least:

```
sock = socket(AF_ALG, SOCK_SEQPACKET, 0)
sock.bind(("aead", "authencesn(hmac(sha256),cbc(aes))"))
op_sock, _ = sock.accept()   # not a network connection
op_sock.sendmsg([data], ...)
result = op_sock.recv(n)
```

- `bind()` selects an *algorithm*, not an address.
- `accept()` creates an "operation handle", not a connection. Basically, an opaque "thing" that lets us later use a particular operation

Eric Biggers (Linux kernel crypto maintainer): "AF_ALG . . . should not exist. It exposes a massive attack surface to unprivileged userspace programs. And it's almost completely unnecessary."

AF_ALG issues

Arguments for:

- Access to hardware crypto accelerators (can only be driven from kernel mode)
- Key material can live entirely in the kernel – never readable by any userspace process
- Avoids duplicating crypto code on memory-constrained embedded systems

Arguments against:

- Exposes enormous, complex kernel subsurface to unprivileged users
- Userspace crypto libraries (OpenSSL, libgcrypt, ...) cover most needs
- Pattern of recurring exploits – Copy Fail is not the first
- authencesn is an IPsec internals detail that has no business being a general-purpose userspace API

AF_ALG issues

Recommendation currently is: disable `CONFIG_CRYPTO_USER_API_*` unless you have a specific reason not to

How the ingredients combine

Three changes were made 6 years apart by different authors:

2011 – `authencesn` added for IPsec Extended Sequence Numbers.
It uses its output buffer as scratch space, writing 4 bytes past the declared end. Seems hamrless - only in-kernel callers, scratch area is throwaway.

2015 – `AF_ALG` gains AEAD support with `splice()` path.
Page cache pages can now enter the crypto subsystem. But as read-only input.

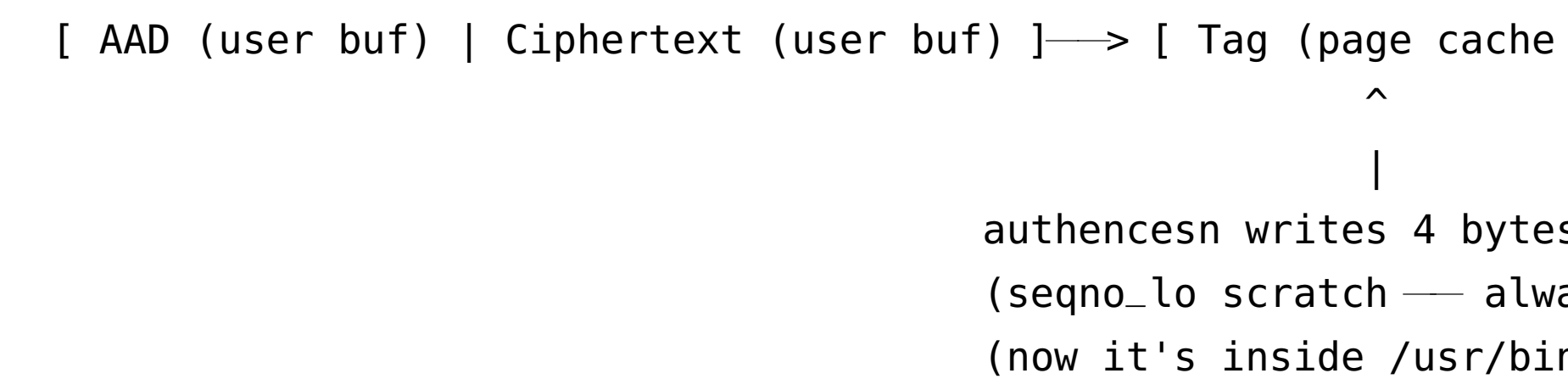
2017 – In-place optimisation added.
Source and destination scatterlists are merged. Page cache pages from `splice()` are now chained into the writable destination scatterlist.

How the ingredients combine

The bug occurs due to *all three* things in combination. No single change was obviously wrong.

Scratch write

authencesn's scratch write - harmless for a decade, now lands in a page cache page:



The HMAC check fails (we provided fake input – we don't care).
The write already happened before the error is returned.
It is not rolled back.

Exploit

Repeat, 4 bytes at a time, to overwrite su's .text section with shellcode:

```
for chunk in shellcode_chunks:
    sendmsg(op_sock, aad=b'\x00'*4 + chunk + ...)
        # 4 bytes we want written
    splice(su_fd, pipe_w, length, offset=target)
        # page cache page into pipe
    splice(pipe_r, op_sock, length)
        # pipe into AF_ALG
    try: op_sock.recv(n)
        # trigger; HMAC fails; irrelevant
    except: pass

execve("/usr/bin/su")   # kernel loads from (corrupted) page cache
```

Exploit

On-disk `/usr/bin/su` will be unchanged, abd `sha256sum` still correct.

So integrity monitoring sees nothing.

Takeaways

Implicit invariants are dangerous
`authencesn`'s "I write past my output boundary" was never documented. Discoverable only by reading three separate codebases simultaneously.

Shared mutable state is a risk multiplier
One write to the page cache corrupts what every process sees, system-wide.

Performance optimisations can change security properties
The 2017 in-place change was a reasonable efficiency improvement. It silently violated an undocumented invariant in a different subsystem.

Attack surface reduction is a real defence
Disabling `AF_ALG` entirely prevents this exploit and every past and future exploit in this class – regardless of what `authencesn` does.