

Copy Fail Flaw: 5 YARA Rules for Detection

reversinglabs.com/blog/copy-fail-5-yara-rules

Maik Morgenstern

May 1, 2026

Here's what you need to know about the Linux kernel privilege escalation — and how to use YARA rules to get on top of it.



The cybersecurity firm Theori publicly disclosed Copy Fail (CVE-2026-31431), a local privilege escalation vulnerability in the Linux kernel, on April 29. It affects every major distribution shipped since 2017. A working exploit — a single 732-byte Python script — was released alongside the disclosure.

What makes Copy Fail particularly alarming is how easy it is to exploit. Any user with a local account on a vulnerable Linux system can gain full administrative control of that machine in seconds — no password required; no misconfiguration to exploit; no traces left on disk.

Here is what ReversingLabs has observed — and YARA rules your team can use to detect Copy Fail in your environment.

The Impact of Copy Fail

The working exploit is a single Python script of 732 bytes using only standard library functions, requires no special tools or privileges, works reliably first time without race conditions or retries, and runs unchanged across every major Linux distribution released since 2017.

Linux underpins the majority of the world's server infrastructure, cloud computing, and enterprise workloads — making the potential attack surface enormous. The highest-risk environments are those where multiple users or workloads share a kernel: multi-tenant servers, CI/CD pipelines running untrusted code, and container clusters.

An attacker who gains an initial foothold on a Linux system can immediately use the Copy Fail flaw to take full control of the underlying host. The bug affects Linux kernel versions 4.14 through 7.0-rc. Patches are available but had not yet shipped for all major distributions at the time of disclosure.

Understanding The Copy Fail Vulnerability

Copy Fail is a logic bug in the Linux kernel's *authencfsn* cryptographic template, reachable by any unprivileged local user via the *AF_ALG* socket interface. By combining an *AF_ALG* socket bound to *authencfsn(hmac(sha256),cbc(aes))* with *splice()* and *sendmsg()*, an attacker can trigger a controlled 4-byte write into the page cache of any readable file — including *setuid* binaries such as */usr/bin/su*.

The write bypasses the normal VFS write path. The kernel never marks the corrupted page dirty, so the on-disk file is unchanged and standard file integrity tools that compare on-disk checksums will not detect the modification. Only the in-memory page cache is affected, but that is what the kernel uses when loading and executing binaries.

The bug exists at the intersection of three kernel changes spanning 2011 to 2017. It was discovered by Theori researcher Taeyang Lee with assistance from the Xint Code AI research platform. The upstream fix (commit *a664bf3d603d*) reverts *algif_aead.c* to out-of-place operation, removing cache pages from the writable scatterlist entirely.

Affected distributions confirmed by Theori include Ubuntu 24.04 LTS, Amazon Linux 2023, RHEL 14.3, and SUSE 16. A Kubernetes container escape using the same primitive is documented in a forthcoming Part 2 from Theori.

For a full technical analysis of the root cause and exploit mechanics, see the Theori writeup at [copy.fail](#), and the CERT-EU advisory [2026-005](#).

Observed Copy Fail Samples

ReversingLabs observed Copy Fail samples using Spectra Intelligence on the day of the disclosure. The original Theori proof of concept (PoC), `copy_fail_exp.py`, 732 bytes, SHA1 `83194d178f4b9c6fcdfaed0ea4ae3ec2ca3db6f4`, was ingested on April 29, with additional copies and minor variants appearing throughout April 30. In total, four distinct Python files were identified matching our YARA rules.

The four functional Python samples are copies of the same proof of concept (PoC). The differing hashes are the result of trivial differences such as line endings (LF vs CRLF), the presence or absence of a shebang line, or minor whitespace changes. None of these affect execution. This illustrates why hash-based detection is insufficient for tracking exploit distribution — the same functional code produces a different hash with every minor copy-paste variation, while the underlying exploit strings remain identical across all samples.

The following unique files were observed across both our ruleset and the Florian Roth signature-base rule [EXPL_LNX_Copy_Fail_Artefacts_CVE_2026_31431_Apr26](#):

SHA1	SHA256	Format	Firs See (UT
83194d178f4b9c6fcdfaed0ea4ae3ec2ca3db6f4	a567d09b15f6e4440e70c9f2aa8edec8ed59f53301952df05c719aa3911687f9	Text/Python	202 04-21:1
39e8878922b183047d9be0da04402b23da289405	3c5ec61632d0699e048d8428461c4d65f89988a370396db2f070f63ebbf9dbae	Text/Python	202 04-203:0
a4d4c899a00f948016279448356c49e8454a09f7	d401e7d1c00605749d6c617ace73ab20a762b72e41c2e1590331596e38219a61	Text/Python	202 04-209:5
11596758ecc621a946c4f262a6d49ff3b45887c9	e097ce64b1bf0933e69c9d342038fb52f4b278da62b265daa3adf22c00658a9c	Text/Python	202 04-210:4

HTML and JavaScript files that matched the broad hunting rule were identified as crawled copies of the Xint blog post and the copy.fail website — expected noise from rules that cast a wide net.

YARA Rules for Copy Fail

Hash-based detection is of limited value here. The original PoC is a clean Python script with no prior malicious history, and any trivial modification produces a different hash. We developed a tiered YARA ruleset designed for hunting, with rules ranging from exact PoC matching to broad primitive detection.

The key detection anchor across all rules is the `authencesn(hmac(sha256),cbc(aes))` string. This is a hard functional constraint of the exploit — it cannot be substituted without losing the page cache write primitive — making it stable across obfuscation, renaming, and repackaging.

The YARA ruleset is structured in five tiers:

Rule 1 — Theori PoC exact match (high confidence)

Anchored on the obfuscated import alias, the `authencesn(hmac(sha256),cbc(aes))` algorithm string, the hex AEAD key `0800010000000010`, the zlib shellcode blob, and `/usr/bin/su` as the target. All strings must be present. Catches the original PoC and near-identical copies.

Rule 2. rootsecdev variant (high confidence)

Covers the second public PoC toolkit, which targets `/etc/passwd` rather than `/usr/bin/su` and uses a `PWND` marker string for its detector script.

Rule 3. Core exploit primitive (medium confidence)

Requires the `authencesn` algorithm string combined with `os.splice` and an `AF_ALG` socket bind to the `aead` type. This combination has no legitimate use case outside kernel testing or exploit development, and will catch renamed or refactored variants that preserve the functional kernel interface.

Rule 4. Compiled and dropper variants (medium confidence)

Targets PyInstaller-packed versions and shell droppers that fetch and execute the exploit.

Rule 5. Broad hunting (low confidence)

The *authencesn* string combined with *splice* usage in Python, C, or Go, and any known *setuid* target path. Designed for proactive corpus sweeps; analyst review required on hits.

```

/*
YARA Rules: CVE-2026-31431 "Copy Fail"
Linux kernel local privilege escalation via AF_ALG + authencsn + splice()
page-cache scratch-write primitive.

Author   : ReversingLabs Threat Research
Date     : 2026-04-30
Version  : 1.0
Reference: https://xint.io/blog/copy-fail-linux-distributions
          https://copy.fail/

Rule tiers:
- HIGH confidence : exact PoC or near-exact clone (safe to alert on)
- MED confidence  : core exploit primitive, likely malicious in context
- LOW confidence  : broad hunting signal, expect FP on crypto/test code
*/

import "math"

rule CopyFail_PoC_Theori_Exact : CVE_2026_31431 exploit lpe linux
{
    meta:
        description = "Detects the original Theori copy_fail_exp.py PoC for CVE-2026-31431"
        author      = "ReversingLabs Threat Research"
        date        = "2026-04-30"
        version     = "1.0"
        severity    = "HIGH"
        confidence  = "HIGH"
        tlp         = "WHITE"
        reference   = "https://copy.fail/"
        cve         = "CVE-2026-31431"
        hash_sha1   = "83194d178f4b9c6fcdfaed0ea4ae3ec2ca3db6f4"
        hash_sha256 = "a567d09b15f6e4440e70c9f2aa8edec8ed59f53301952df05c719aa3911687f9"

    strings:
        $import_alias = "import os as g,zlib,socket as s" ascii
        $aead_algo     = "authencsn(hmac(sha256),cbc(aes))" ascii
        $aead_key      = "0800010000000010" ascii
        $zlib_magic    = { 78 DA }
        $target_su     = "/usr/bin/su" ascii
        $shellcode_blob = "78daab77f571" ascii nocase

    condition:
        filesize < 10KB and
        $import_alias and
        $aead_algo and
        $aead_key and
        $target_su and
        ($shellcode_blob or $zlib_magic)
}

rule CopyFail_PoC_rootsecdev : CVE_2026_31431 exploit lpe linux
{
    meta:
        description = "Detects the rootsecdev CVE-2026-31431 toolkit (exploit + detector)"
        author      = "ReversingLabs Threat Research"
        date        = "2026-04-30"
        version     = "1.0"
        severity    = "HIGH"
        confidence  = "HIGH"
        tlp         = "WHITE"
        reference   = "https://github.com/rootsecdev/cve_2026_31431"
        cve         = "CVE-2026-31431"

    strings:
        $aead_algo     = "authencsn(hmac(sha256),cbc(aes))" ascii
        $pwnd_marker   = "PWND" ascii
        $etc_passwd    = "/etc/passwd" ascii
        $vuln_msg      = "VULNERABLE to CVE-2026-31431" ascii
        $precond_msg   = "Precondition not met" ascii
        $pagecache_msg = "Page cache MODIFIED via in-place AEAD splice path" ascii
        $fadvise       = "POSIX_FADV_DONTNEED" ascii

    condition:
        filesize < 50KB and
        $aead_algo and

```

```

(
    ($pwdn_marker and $etc_passwd) or
    ($vuln_msg and $precond_msg) or
    ($pagecache_msg and $fadvise)
)
}

rule CopyFail_Primitive_Python : CVE_2026_31431 exploit lpe linux hunting
{
    meta:
        description      = "Detects Python scripts using the AF_ALG authencsn + os.splice() primitive (CVE-2026-31431 core technique)"
        author           = "ReversingLabs Threat Research"
        date             = "2026-04-30"
        version          = "1.0"
        severity         = "HIGH"
        confidence       = "MEDIUM"
        tlp              = "WHITE"
        reference        = "https://xint.io/blog/copy-fail-linux-distributions"
        cve              = "CVE-2026-31431"

    strings:
        $aead_algo       = "authencsn(hmac(sha256),cbc(aes))" ascii
        $splice_call     = "os.splice" ascii
        $splice_import   = "from os import splice" ascii
        $af_alg_int      = "socket(38," ascii
        $af_alg_const    = "AF_ALG" ascii
        $aead_bind       = ".bind(\\\"aead\\\"" ascii
        $aead_bind2      = ".bind(('aead'" ascii

    condition:
        filesize < 500KB and
        $aead_algo and
        (1 of ($splice_call, $splice_import)) and
        (1 of ($af_alg_int, $af_alg_const)) and
        (1 of ($aead_bind, $aead_bind2))
}

rule CopyFail_Compiled_Or_Dropper : CVE_2026_31431 exploit lpe linux hunting
{
    meta:
        description      = "Hunting rule for compiled/packed Copy Fail exploits or shell droppers embedding the authencsn AEAD path"
        author           = "ReversingLabs Threat Research"
        date             = "2026-04-30"
        version          = "1.0"
        severity         = "HIGH"
        confidence       = "MEDIUM"
        tlp              = "WHITE"
        reference        = "https://copy.fail/"
        cve              = "CVE-2026-31431"

    strings:
        $aead_algo       = "authencsn(hmac(sha256),cbc(aes))" ascii wide
        $curl_exp        = "curl https://copy.fail/exp" ascii
        $wget_exp        = "wget.*copy.fail" ascii
        $pyinstaller     = "PyInstaller" ascii
        $pyi_archive     = "MEI\\x0d\\x0a\\x1a\\x0a"
        $exec_pattern1   = "python3 copy_fail" ascii
        $exec_pattern2   = "python3 exploit_cve_2026" ascii

    condition:
        $aead_algo and
        (
            any of ($curl_exp, $wget_exp, $exec_pattern1, $exec_pattern2) or
            ($pyinstaller or $pyi_archive)
        )
}

rule CopyFail_Hunting_Broad : CVE_2026_31431 lpe linux hunting lowconfidence
{
    meta:
        description      = "HUNTING – broad signal for authencsn + splice + setuid target combination. Expect FP on crypto test code."
        author           = "ReversingLabs Threat Research"
        date             = "2026-04-30"
        version          = "1.0"
        severity         = "MEDIUM"

```

```

confidence      = "LOW"
tlp              = "WHITE"
reference        = "https://copy.fail/"
cve              = "CVE-2026-31431"

strings:
  $aead_algo      = "authencesn(hmac(sha256),cbc(aes))" ascii wide
  $target_su      = "/usr/bin/su" ascii
  $target_passwd  = "/etc/passwd" ascii
  $target_sudo    = "/usr/bin/sudo" ascii
  $target_passwd2 = "/usr/bin/passwd" ascii
  $splice_py      = "os.splice" ascii
  $splice_c       = "splice(" ascii
  $splice_go      = "syscall.Splice" ascii

condition:
  $aead_algo and
  (1 of ($target_su, $target_passwd, $target_sudo, $target_passwd2)) and
  (1 of ($splice_py, $splice_c, $splice_go))
}

```

Beyond the Python PoCs, ReversingLabs identified a set of compiled ELF binaries containing the same shellcode payload embedded in the original Theori exploit. These were detected using the YARA rule *CopyFail_Shellcode_Reference_1*, authored by Malware Utkonos, which targets the three core syscall sequences in the x86-64 shellcode: *setuid(0)* (syscall 105), *execve* (syscall 59), and *exit* (syscall 60).

YARA

```

rule CopyFail_Shellcode_Reference_1
{
  meta:
    author = "Malware Utkonos"
    date = "2026-04-30"
    description = "Detects CopyFail shellcode patterns."

  strings:
    $op1 = { b069 0f05 }
      // 0040007c b069      mov     al, 0x69 // syscall 105 = setuid
      // 0040007e 0f05      syscall

    $op2 = { 6a3b 58 99 0f05 }
      // 00400089 6a3b      push    0x3b // syscall 59 = execve
      // 0040008b 58      pop     rax
      // 0040008c 99      cdq
      // 0040008d 0f05      syscall

    $op3 = { 6a3c 58 0f05 }
      // 00400091 6a3c      push    0x3c // syscall 60 = exit
      // 00400093 58      pop     rax
      // 00400094 0f05      syscall

  condition:
    all of them
}

```

The two smallest files are the raw shellcode ELF files — standalone 160- and 161-byte x86-64 executables containing only the shellcode itself, with no exploit logic. These are the payloads that get injected into the page cache of the target *setuid* binary. The remaining 12 files are compiled C implementations of the shellcode, statically linked and stripped, ranging from 30KB to 308KB. The size variation reflects different compiler flags, optimization levels, and minor code differences rather than fundamentally different functionality.

SHA1	SHA256
14cbc8ac13b7378a040fd12dcb99f6810c0253ce	7ef953069578beef7daf2d984d16351b0c60c6adcbf8062ffbfac6ce944c1da
6bc9a09cf2753d6d2325f9db18dcb48503cb0ae9	01881b737c106a6d84e8e12cb417671b5f612f13adcb6cb1edd8a46704ddd252
fb35dbd2973d96d0899dc6318fc2f3cdb080fac2	624d27965a570bc891a87bd62dab28fe401b145e1cbeea6caebcc53a7689bd9d
3f866ee920940b388ddd70b7a8fbf7462f1f5bf2	8813ad1310ed2e1c1e58cdef0169bef5f38f58b740bd170a370d4baf03dfde99
28d304df5855632d5e7b2801d1092460da7dff22	fd27a1d93dfc6afb3f705421449c87069595282c91e279660cb06c384d2acaea
dbf2e97d93ed025cf8ab782f0834ffb272f38833	32a2bd2c5f64b05d5db729a42633b792e00e56dd3e4644d21a6e45273b235dbd

81e9ca6d22f0323eec32c6254dd84cd437246a84	989854c46208c6c1304e2d7038ecabea0cf19f39e6fda3d28576837c22961f21
9490c835a03ad3b874f978a4c7a18120bf0f284d	44900c631391f0d60eb6d271b8374a08dc1d9be76e403390d27a91ed5f179be9
5878d816b2772285c79801a0706740d8bda469f4	30b0f5b5a054c4df65b48ca792863bf7054b4d793f15f57163792ba6c2b151ae
f2fe4dc6b1823231414b941de723e225268c4f81	d86251e90db613329d4be9cf0355caa69ae88f7416be82a6880b8f9f7b7e691b
047ac05d0cb9c2ee3fc479339210caf36c0f171d	f6dda491e85981feb1c0900f2b16f556e7aa2ca79b569b1180549cae58c22f98
86d25436e074e8f999a3bab9d3f3a25f7e969073	1ce3dc778637cd78e20f115fe1e47debf16d6af538daf95dff14243111779e28
9d387ad2b64da532867f60a97506ce6daec89c51	60b1645b1f3ad7f6391243d58a2a31f974e1599364564a69594b96ba21abe375
220a249f8791d060a2f33d59f52fe281f6598dcc	c6dd71f1f41c5d6fca0d3261a5ae9e49d01d6755c62d487e9a0cd7de933d6f64

Compiled Implementations: goodcopy.c by blasty

A notable C reimplement of the exploit — *goodcopy.c* — was published by security researcher blasty shortly after disclosure. The code is a deliberate port of the Python PoC to C, adding x86-64 and AArch64 architecture support.

The key difference from the original Theori shellcode is the payload. Where the original executes *setuid(0)* followed by *execve("/bin/sh")*, the blasty shellcode runs:

```
setuid(0) → execve("/bin/sh", ["/bin/sh", "-c", "echo 3 >/proc/sys/vm/drop_caches; exec sh -i"])
```

The *echo 3 >/proc/sys/vm/drop_caches* call flushes the page cache after execution, cleaning up the corruption so that the target binary is restored to its original state in memory. This is a meaningful operational difference — it prevents other users on the system from inadvertently triggering the injected shellcode after the exploit has been used. The implementation also allows specifying a custom *setuid* target binary rather than hardcoding */usr/bin/su*.

These files are detected by the YARA rule *CopyFail_Shellcode_BlastyAMD64_1* (and corresponding ARM and RISCv64 variants) by Malware Utkonos. The AMD64 rule targets the slightly different opcode pattern used in the blasty shellcode compared to the reference shellcode:

YARA

```
rule CopyFail_Shellcode_BlastyAMD64_1
{
  meta:
    author = "Malware Utkonos"
    date = "2026-04-30"
    description = "Detects CopyFail shellcode patterns found in the Blasty x86_64 implementation."
    reference = "https://gist.github.com/blasty/d7b5d0599b154c9ec83c182acbd56e8b"
  strings:
    $op1 = { b069 0f05 }
    $op2 = { b03b 0f05 }
    $op3 = { b03c 0f05 }
    $ssh = "echo 3 >/proc/sys/vm/drop_caches;exec sh -i"
  condition:
    all of them
}
```

Two compiled x86-64 binaries of *goodcopy.c* have been identified in Spectra Intelligence, built with different compiler flags. No ARM or RISCv64 compiled samples have been observed yet, though the YARA rules for those architectures are in place.

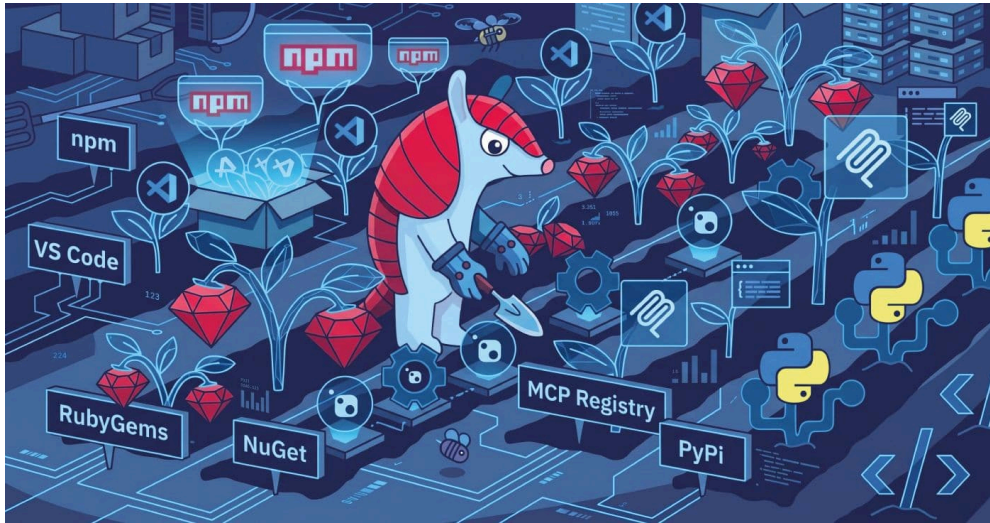
SHA1	SHA256	Size	Type
87915e67782ebd34d91a2b557b22eccf0f6e3de1	b35ddf2ecd035faf9b38af62779502b7fa19037115054a00ed8d5327a3f2ec03	6.7KB	C source
de824c48ca8e2d41dcbb99468148370b1f2c1497	ed0018054d8e7058b299b7591bc32364dbc439c25be4067450189b1a73033c67	930KB	ELF64, static

Key References for Further Review

- Theori disclosure: <https://copy.fail>
- Theori GitHub PoC: <https://github.com/theori-io/copy-fail-CVE-2026-31431>
- CERT-EU Advisory 2026-005: <https://cert.europa.eu/publications/security-advisories/2026-005/>
- Upstream kernel fix: <https://github.com/torvalds/linux/commit/a664bf3d603dc3bdcf9ae47cc21e0daec706d7a5>
- Florian Roth YARA rule: https://github.com/Neo23x0/signature-base/blob/master/yara/expl_copy_fail_cve_2026_31431.yar
- blasty goodcopy.c: <https://gist.github.com/blasty/d7b5d0599b154c9ec83c182acbd56e8b>

Tags: [Threat Research](#)

Topics



Why RL Built Spectra Assure Community

We set out to help dev and AppSec teams secure the village: OSS dependencies, malware, more. Learn how.

[Read More about Why RL Built Spectra Assure Community](#)

[Why RL Built Spectra Assure Community](#)

SUBSCRIBE

Get the best of RL Blog delivered to your in-box weekly. Stay up to date on key trends, analysis and best practices across threat intelligence and software supply chain security.