CVE-2026-31431 (Copy Fail) Vulnerability FAQ (Part 2) — Vulnerability Mechanism, Historical Background, and Strategic Implications

The original report is in Chinese, and this version is an AI-translated edition.

Disclaimer: This article is based on a FAQ list compiled by Antiy CERT engineers, completed collaboratively by multiple AI agents, with some content manually revised and rewritten. Although we have manually verified the information, due to the large amount of information, the AI-generated content may contain factual inaccuracies or timeliness errors. Technical verification cannot be completed in a short time, so readers are advised to cross-verify. If you find any errors or have any questions, please leave a message directly. We will continue to revise the website version of the FAQ after verification.

Editor's Note

On April 29, 2026, South Korean security company Theori and its AI security research platform Xint publicly disclosed a Linux kernel native privilege escalation vulnerability, CVE-2026-31431, codenamed "Copy Fail". Antiy CERT is closely monitoring and responding to this vulnerability.

This vulnerability exists in the Linux kernel cryptosystem, affecting almost all major Linux distributions released since 2017. Attackers, requiring only local user privileges, can reliably gain root access using a Python script of less than 800 bytes under uncontested conditions. The vulnerability achieves execution hijacking by polluting the setuid function in the page cache and, under certain configurations, possesses the potential to escape through containers. This FAQ, compiled by Antiy CERT, aims to provide precise and actionable threat awareness and response guidance to diverse audiences.

Explanation of the article's structure:

■ **Part 1**: Primarily aimed at the public, network administrators, network users, industry regulatory authorities, and corporate IT decision-makers. It focuses on basic vulnerability awareness, impact assessment, asset identification, remediation and mitigation measures, and threat posture evaluation. The writing style strives for clarity and accuracy and can be directly used as reference material for internal communications and management reports.

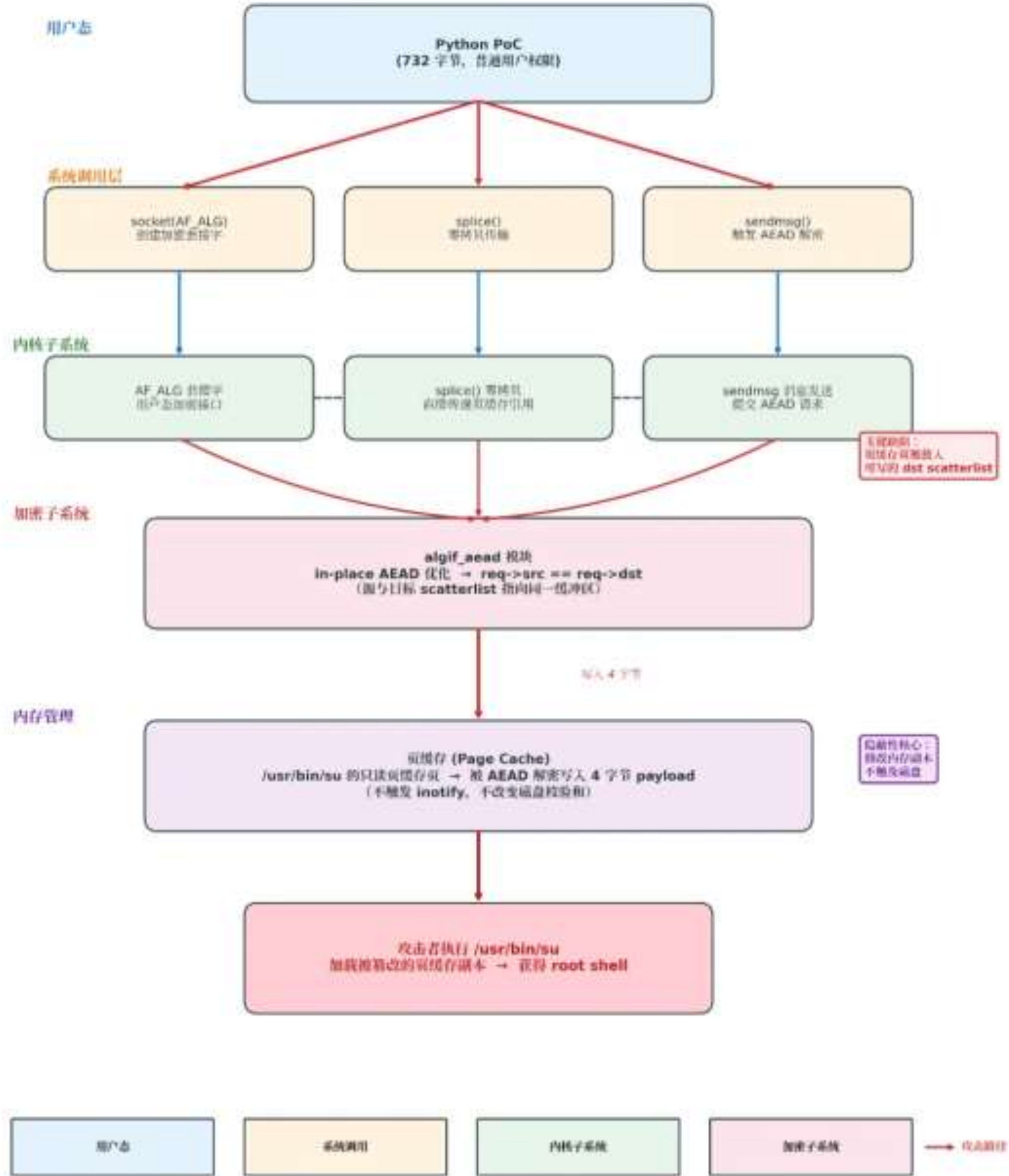
■ **Part 2 (This Article)**: Targeting security researchers, analysis engineers, kernel developers, and advanced blue team personnel. It focuses on the underlying technical mechanisms of vulnerabilities, exploit chain disassembly, comparative analysis with historical vulnerabilities, and strategic reflections on the Linux kernel security auditing mechanism, offering a high level of technical depth.

The two articles complement each other and cite the same sources, both based on cross-validation of publicly disclosed information, the oss-security mailing list, upstream kernel patches, and community testing feedback.

1. Vulnerability technical mechanism and root causes

Q1: Which specific subsystems or mechanisms of the Linux kernel (AF_ALG, splice, page cache, etc.) are affected by this vulnerability?

A: CVE-2026-31431 is a typical **cross-subsystem emergency vulnerability**, involving abnormal coupling of five independent kernel subsystems:



Caption: Technical architecture diagram of CVE-2026-31431 (Copy Fail)

This is a cross-subsystem coupled emergency vulnerability. The red arrow indicates the attack path, which penetrates from the user-space Python PoC through the system call layer (socket/splice/sendmsg) to the kernel subsystem, and finally uses the in-place optimization defect of algif_aead to write the payload to the page cache, achieving root privilege escalation.

Roles of each subsystem:

Subsystem	Core mechanism	Role in the vulnerability
AF_ALG	User-space encrypted interface (socket address family)	Exposing kernel scatterlist operation capabilities to non-privileged users provides a legitimate path from user space to the kernel cryptosystem.
algif_aead	AEAD algorithm interface module	The in-place optimization introduced in 2017 resulted in req->src == req->dst, breaking the source/destination buffer isolation.
authencesn	AEAD algorithm implementation (authencesn(hmac(sha256),cbc(aes)))	When performing AEAD decryption, the seqno_lo field (4 bytes) of AAD is written to the destination scatterlist.
splice()	Zero-copy data transfer	Pass a reference to the page cache (instead of a copy) directly to the AF_ALG socket to maintain the page's "page cache identity".
Page Cache	File page cache	It was mistakenly placed into a writable scatterlist,

		becoming a target for an attacker to write to.

Q2: What role does the AF_ALG encryption interface play in this exploit chain?

A: AF_ALG is **the entry point** for exploitation and **the capability amplifier**.

AF_ALG basics:

- AF_ALG (Address Family ALGorithm) is a socket address family introduced by the Linux kernel in 2011, allowing user-space programs to call the kernel crypto API through the standard BSD socket API.
- Supported algorithm types: skcipher (symmetric encryption), aead (authentication encryption), hash (hash), rng (random number).
- AF_ALG sockets can be created **without root privileges (provided that** CONFIG_CRYPTO_USER_API_AEAD is enabled in the kernel).

The three roles in the vulnerability:

- **Capability Exposure**: Exposes the ability to access the kernel's scatterlist mechanism to ordinary user-space programs. Normally, user-space programs cannot directly manipulate the physical pages of the page cache.
- **Boundary Crossing**: Provides a legitimate path from user space to the kernel scatterlist. Attackers can access internal kernel data structures using only the standard socket API, without needing kernel modules, /dev/mem, or /proc/kcore.
- **Algorithm Trigger**: By binding the authencesn(hmac(sha256),cbc(aes)) algorithm, algif_aead writes data to the target scatterlist when performing decryption, thereby transforming the "read-only page cache page" into a "writable attack surface".

Key code path:

```
// Create AF_ALG socket in user space
```

```
int sock = socket(AF_ALG, SOCK_SEQPACKET, 0);



// Binding AEAD algorithm

struct sockaddr_alg sa = {

.salg_family = AF_ALG,

.salg_type = "aead",

.salg_name = "authencesn(hmac(sha256),cbc(aes))"

};

bind(sock, (struct sockaddr *)&sa, sizeof(sa));



// Pass the file page to the socket using splice()

splice(fd_in, NULL, sock, NULL, size, SPLICE_F_MOVE);
```

Q3: Why did the splice() system call become the critical path that triggered this vulnerability?

A: splice() is **the key bridge connecting the page cache** and **the AF_ALG socket**, and its "zero-copy" feature is a necessary condition for the vulnerability to be triggered.

splice() mechanism:

• splice() is a zero-copy data transfer system call introduced in Linux 2.6.17, used to move data between two file descriptors without going through a user-space buffer.

• When fd_in is a regular file, splice() directly passes **a reference to its page cache pages** instead of copying the data content.

• Page cache pages are usually **read-only** (high reference count, mapped as read-only).

Key differences that trigger the vulnerability:

Data transmission	Has it been	Is the page cache page	Has the vulnerability

method	cached?	passed to AF_ALG?	been triggered?
read() + write()	Yes	❌ No (Data is copied to the user-space buffer)	❌ No
sendfile()	Yes	❌ No (typically used for network sockets)	❌ No
splice()	Yes	✅ Yes (directly passing page cache page references)	✅ Yes
mmap() + direct write	Yes	❌ No (User space cannot directly write to the page cache)	❌ No

Why use splice() instead of other methods?

• read() **copies** the page cache data to the user-space buffer. When write() is called back to the socket in user space, the data has entered a new anonymous memory page and **is no longer a page cache page**.

• splice() **does not copy data**; it directly passes a reference to the page cache, thus maintaining the page's "page cache identity" (the mapping field of the struct page points to address_space).

Only splice() can directly send "read-only pages belonging to the page cache" into the processing path of AF_ALG, causing algif_aead to mistakenly treat the page cache page as a writable target.

Q4: How can an attacker exploit this vulnerability to write data to the page cache of a "read-only" or "trusted" file?

A: The attack process consists of four stages, forming a complete exploitation chain:

Phase 1: Goal Selection

The attacker selects any **readable** setuid binary file on the system (such as /usr/bin/su). The page cache for this file already exists in memory (loaded when the system is first executed after startup).

Phase 2: Construction of AF_ALG Request

```
# Create AF_ALG socket

sock = socket.socket(socket.AF_ALG, socket.SOCK_SEQPACKET, 0)



# Binding the AuthenceSn algorithm

sock.bind(("aead", "authencesn(hmac(sha256),cbc(aes))"))



# Set the AAD authentication tag length (16 bytes)

sock.setsockopt(socket.SOL_ALG,
socket.ALG_SET_AEAD_AUTHSIZE, 16)
```

Phase 3: Splice Page Caching Page

```
# Open the target file (read-only)

target_fd = os.open("/usr/bin/su", os.O_RDONLY)



# Pass page cache pages to the AF_ALG socket via splice

# Key point: Splice passes a reference to the page cache page, not a copy
of the data.

os.splice(target_fd, None, sock.fileno(), None, 4096,
os.SPLICE_F_MOVE)
```

Phase 4: Trigger Write Operation

```
# Sending a message triggers AEAD decryption

# When the kernel performs authenticationsn decryption, it writes the AAD's seqno_lo to the dst scatterlist (i.e., page cache pages).

sock.sendmsg_afalg(op=socket.ALG_OP_DECRYPT, iv=b"x"*16, assoclen=16)
```

Characteristics of the data written:

- **Write position: The specific offset in the target file page cache (the position of the** seqno_lo field in the AAD structure).
- **Write content: 4 bytes of controlled data (the** seqno_lo field in the AAD structure, i.e., bytes 4–7)
- **Write count**: Can be repeated, gradually overwriting more bytes to achieve arbitrary code injection.

Q5: Why can modifying the setuid binary file in the page cache enable the user to gain permissions from a regular user to root?

A: It utilizes a combination of the Linux **setuid mechanism** and the characteristics of **page cache execution**.

setuid mechanism:

- setuid (Set User ID) is a file permission mechanism in Unix/Linux that allows ordinary users to execute programs with the permissions of the file owner.
- Files such as /usr/bin/su, /usr/bin/sudo, and /usr/bin/passwd usually belong to root and have the setuid bit set (permission 4755).
- When ordinary users execute these programs, the process's effective UID (euid) is temporarily promoted to 0 (root).

Attack logic chain:

```
1. The attacker tampered with the page cache copy of /usr/bin/su.
    └── Injecting malicious instructions into the code path of su

2. The attacker executes /usr/bin/su
    └── When the kernel loads /usr/bin/su, it prioritizes using the page cache copy in memory.
    └── The disk files have not been modified, therefore file permissions and checksums are normal.

3. `su` runs with root privileges (euid=0) under the setuid mechanism.
    └── Triggers the attacker's injected payload
    └── Execute the attacker-controlled path (e.g., execve("/bin/sh"))
```

Why page cache tampering works:

• When the kernel executes a file, it first checks whether the page of the file already exists in the page cache (by looking up `the address_space` of `inode->i_mapping`).

• If the page exists in the page cache (and has not been reclaimed), the kernel uses the copy in the page cache directly and **does not read it from the disk again**.

• The attacker modifies a copy in the page cache, while the original file on disk remains unchanged.

• Therefore, tools such as `ls -l`, `sha256sum`, `AIDE`, and `Tripwire` show perfectly normal results when checking disk files.

• Because the modification is done on the page cache rather than the disk, **it does not trigger inotify/fanotify**, **does not change the disk file checksum**, and traditional integrity monitoring is completely ineffective.

Q6: Does this vulnerability exploitation rely on a race condition? How does its stability compare to historical vulnerabilities such as Dirty COW?

A: **It does not depend on race conditions** and its stability far exceeds that of Dirty COW.

Historical vulnerability stability comparison

Vulnerability	Competitive conditions	Success probability	Utilize complexity	PoC volume	Cross-version adaptation
CVE-2026-31431 (Copy Fail)	✗None	Extremely high single-shot success rate	732-byte Python	Minimalism	✓ No offset required
CVE-2022-0847 (Dirty Pipe)	✗None	Extremely high single-shot success rate	~400 bytes C	Minimalism	□ Specific version required
CVE-2016-5195 (Dirty COW)	✓ Required	Multiple attempts are required.	More complicated	medium	□ Multiple attempts are required.
CVE-2021-4034 (PwnKit)	✗None	Extremely high	~200 bytes C	Minimalism	✓ General

		single-shot success rate			

The source of the stability of Copy Fail

• **Deterministic execution path**: socket() → bind() → splice() → sendmsg() are sequential system calls, with no concurrent operations and no timing dependencies.

• **No kernel state contention**: It does not depend on kernel memory layout, specific timing, the number of CPU cores, or system load.

• **Cross-version compatibility**: No need to adjust offset addresses or ROP gadgets for different kernel versions; the same PoC can run on all affected versions.

• **Pure Python implementation**: It can be used with a 732-byte Python script, without compiling C code, kernel modules, or a specific compiler.

• **VFS bypass**: The write path completely bypasses the VFS layer and does not trigger any file system access control hooks (DAC/MAC are both ineffective).

2. Historical coordinates and strategic implications

Q7: What are the similarities and differences between Copy Fail, Dirty COW (CVE-2016-5195), and Dirty Pipe (CVE-2022-0847) in terms of triggering path, exploitation threshold, and stability?

A:

Comprehensive comparison table

Comparison dimensions	CVE-2026-31431 (Copy Fail)	CVE-2022-0847 (Dirty Pipe)	CVE-2016-5195 (Dirty COW)
Discovery time	April 2026	February 2022	October 2016
Incubation	~9 years (2017–	~2 years (Linux	~9 years (2007–

period	2026)	5.8+)	2016)
Vulnerability type	Kernel logic defects	Kernel logic defects	Kernel race conditions
Trigger path	AF_ALG + splice + page cache	Pipe buffer flags abuse	COW Page Race Conditions
Subsystems involved	Encryption subsystem, splice, page cache	Pipe, page cache	MMU, COW, Page Cache
Competitive conditions	✗None	✗None	✓ Required
Utilizing thresholds	Extremely low (Python script)	Extremely low (C program)	medium
PoC volume	732-byte Python	~400 bytes C	More complicated
Stability	Extremely high single-shot success rate	Extremely high single-shot success rate	Multiple attempts are required.
Cross-version adaptation	✓ No offset required	□ Requires a specific kernel version	□ Multiple attempts are required.
Write to target	Page caching (non-dirty pages)	Page caching (non-dirty pages)	COW Private Mapping
Disk traces	✗None	✗None	□ May be
Concealment	⋆⋆⋆⋆⋆ Extremely high	⋆⋆⋆⋆ High	⋆⋆⋆ Medium

VFS bypass	✅ Yes	✅ Yes	❌No
Container escape	✅ Yes (as claimed by the disclosing party)	❌No	❌No
Post-persistence	❌ None (will disappear upon restart)	❌ None (will disappear upon restart)	☐ May become persistent
Repairing complexity	In-place rollback optimization	Limit pipe flags	Fix COW race condition

Key Difference Analysis

Copy Fail vs Dirty Pipe:

• **Similarity**: Both utilize the "non-dirty page" feature of page caching to modify file copies in memory without touching the disk, thus bypassing traditional file integrity monitoring.

• **Differences**: Copy Fail's trigger path is less common (AF_ALG encrypted interface), while Dirty Pipe utilizes a more general pipe mechanism. Copy Fail also has **container escape** capabilities (page cache shared across containers), while Dirty Pipe is limited to page cache tampering within a single host.

Copy Fail vs Dirty COW:

• **Similarities**: Both have been lurking for about 9 years, both involve page caching/memory management mechanisms, and both have affected a large number of Linux distributions.

• **Differences**: Dirty COW requires race conditions, is unstable, and its success rate depends on luck and system state; Copy Fail is a deterministic vulnerability with an extremely high single-attack success rate. Dirty COW modifies the private COW mapping

(which may be persisted to disk), while Copy Fail modifies the global page cache (which disappears upon restart and is more stealthy).

Q8: The vulnerability originates from the password encryption security mechanism. Why would the corresponding security mechanism become a security weakness?

A: This is a profound case about **"security mechanisms themselves becoming attack surfaces"**.

The original purpose of cryptographic security mechanisms:

• The original design purpose of AF_ALG was to expose the kernel's FIPS-certified encryption implementation to user space, thus avoiding potential vulnerabilities that user-space libraries (such as OpenSSL) might introduce by implementing their own encryption algorithms.

• The in-place optimization of algif_aead aims to reduce memory copying overhead and improve AEAD decryption performance by approximately 10-20%.

• The zero-copy design of splice() aims to reduce the copying of data between kernel mode and user mode, thereby improving I/O performance.

Why it has become a security vulnerability:

• **Over-reliance on internal paths**: Kernel developers assumed that the "data buffer passed through AF_ALG" and the "normal file page cache" are mutually exclusive. That is, AF_ALG deals with "anonymous memory pages", not "file-backed page cache pages". However, splice() breaks this assumption.

• **Performance optimization violates safety invariants**: The core assumption of in-place optimization is that "the source buffer and the destination buffer are in the same memory region, and that region is writable". However, this assumption fails when a page cache page is passed to splice() - the page cache page is "read-only" but is treated as a "writable destination buffer".

• **Abstraction layer leakage**: AF_ALG abstracts the encryption operation but leaks the implementation details of the underlying scatterlist. Attackers do not need to understand the AEAD algorithm, only the side effect of "writing data to the target scatterlist".

• **Violation of the principle of least privilege**: AF_ALG grants **all non-privileged users** the ability to manipulate the kernel scatterlist. In microkernel or capability-based system designs, such operations should be limited to privileged processes or processes with explicit authorization.

Lesson learned: Security mechanisms must be designed to account for **potential misuse scenarios**. Performance optimization should not come at the expense of breaking security invariants, and any interface that exposes internal kernel mechanisms (such as scatterlist and page cache references) to user space should undergo rigorous security boundary analysis.

Q9: What implications does the nine-year latency period of this vulnerability have for Linux kernel code auditing and regression testing mechanisms?

A:

Warning 1: Blind spots in cross-subsystem coupling

Copy Fail involves the interaction of **five independent subsystems** (AF_ALG, algif_aead, splice, authenticationsn, and page cache). Traditional code auditing is usually the responsibility of each subsystem maintainer, making it difficult to discover emergent vulnerabilities across subsystems.

Suggestions for improvement:

• Establish **a cross-subsystem interaction test matrix**, focusing on testing edge scenarios where "the output of subsystem A is used as the input of subsystem B".

• For "universal connector" system calls like splice(), establish combination tests with all possible target subsystems (character devices, sockets, cryptographic interfaces, BPF maps, etc.).

• Introducing "Data Flow Tracing" auditing: Tracing the complete path of the page cache page from splice() to AF_ALG to authenticationsn, verifying access permissions at each stage.

Warning 2: The security cost of performance optimization

The in-place optimization in 2017 (req->src == req->dst) was a reasonable improvement from a performance perspective, but it broke the safety invariant of "source buffer and destination buffer isolation".

Suggestions for improvement:

• For performance optimizations involving "shared buffers", "zero copy", and "in-place", mandatory **security reviews are required**.

• Introduce "invariant checks": In the optimized code path, assert src != dst or page_is_writable(dst) using BUG_ON() or WARN_ON().

• Establish a "Performance Optimization Security Review Checklist" that requires optimization submitters to clearly explain "which security assumptions the optimization breaks, and whether these assumptions still hold true in other code paths".

Warning 3: Insufficient regression test coverage

The kernel test suite (kselftest, LKFT) failed to cover the edge case of " splice() passing a page cache page to AF_ALG AEAD decryption".

Suggestions for improvement:

• Extend kselftest to add combined testing of splice() with various character devices, sockets, and encryption interfaces.

• Introduce **combinatorial fuzzing**: For example, Syzkaller should be extended to cover the combination of AF_ALG + splice() + sendmsg().

• Establish a blacklist of "dangerous system call combinations" and automatically generate and execute fuzz test cases.

Warning 4: Security debts of "niche" subsystems

Since its introduction in 2011, AF_ALG has received far less attention from security research than its network and file system subsystems. Nearly 15 years of code accumulation has created a huge "security debt".

Suggestions for improvement:

• Conduct a specialized security audit on the interfaces that expose the internal mechanisms of the kernel to the user space (such as AF_ALG, io_uring, BPF, perf).

• Establish a list of "user-mode triggerable kernel paths" and audit them regularly.

• Introduce the concept of "attack surface budget": Each new user-space-kernel interface must be evaluated for its increased attack surface, and corresponding tests and monitoring must be implemented.

Warning 5: The value and limitations of AI-assisted auditing

Copy Fail was detected from the crypto/ subsystem by Xint's AI-assisted code auditing tool in about 1 hour, demonstrating the advantages of AI in **large-scale scanning** and **pattern recognition.**

Suggestions for improvement:

• Integrate AI-assisted auditing into the core security process as a pre-screening tool for manual audits.

• High-risk candidate vulnerabilities discovered by AI are given priority for manual review.

• However, we should not rely too much on AI: AI discovers "candidate vulnerabilities", and the final verification, exploitation, development, and fixes still need to be done by human experts.

Q10: How should future vulnerability discovery and defense systems evolve to address emergent vulnerabilities caused by "multi-mechanism coupling"?

A:

Evolution of vulnerability discovery

Direction	Specific measures	Priority
Combinatorial fuzz testing	Extend fuzzers such as Syzkaller to focus on testing system call combinations (such as splice() + socket() + sendmsg()).	High
Static analysis enhancement	Develop cross-subsystem data flow analysis tools to detect patterns where "read-only data is treated as writable targets".	High
Formal verification	Formal proofs are performed using Coq/Isabelle for critical kernel paths (such as scatterlist processing).	Middle
AI-assisted auditing	Train a large language model to recognize high-risk code patterns such as "in-place optimization", "zero copy", and "src==dst".	Middle
The red team continued testing.	Establish a kernel red team to conduct adversarial tests regularly on "user-mode triggerable kernel paths".	High
Supply chain audit	Perform a special audit of the kernel crypto subsystem, similar to the post-Heartbleed audit of OpenSSL.	High

Evolution direction of defense system

Direction	Specific measures	Priority
Runtime	**a default disable + on-demand enable** strategy for high-risk interfaces such as	High

micro-isolation	AF_ALG and io_uring.	
eBPF Real-time Protection	Deploy eBPF probes to monitor abnormal system call combinations (such as splice() to AF_ALG).	High
Page Cache Integrity	Research kernel-level page cache integrity protection mechanisms to prevent user-mode page cache writes.	Middle
Supply chain security	Integrate kernel vulnerability scanning into CI/CD to ensure that container images and firmware use secure kernel versions.	High
Zero Trust Kernel	Even within the kernel, the principle of "least privilege" is adhered to, restricting data flow between subsystems.	Middle
Rapid repair capability	Kernel Live Patching, container image hot reloading, and other technologies should become standard features of infrastructure.	High

Strategic Implications

Copy Fail reveals a fundamental challenge of modern operating systems: **the tension between performance optimization and security**. Optimization techniques such as zero-copy, in-place computing, and shared memory have greatly improved system performance, but have also compromised traditional security boundaries.

Future kernel design should follow these principles:

- **"Default safety" takes precedence over "default performance"**: High-risk optimizations should be disabled by default and enabled only after explicit configuration. For example, in-place optimization for AF_ALG could be a startup parameter option, rather than the default behavior.

- **"Auditability" is a design constraint**: any new user-mode-kernel-mode interface must be accompanied by an auditable log path. The kernel should provide "system call composition auditing" capabilities to log data flows across subsystems.
- **"Rapid patching" capabilities**: Kernel Live Patching, container image hot updates, and firmware OTA should be standard infrastructure features. One of the fundamental reasons for the nine-year latency period is the "excessive cost of patching" (requiring reboots and distribution coordination). If the kernel had true online patching capabilities, the window of opportunity for such vulnerabilities would be significantly shortened.
- **"Attack surface minimization"**: Kernel interfaces exposed to user space should adhere to the "minimum necessary" principle. AF_ALG exposes all encryption algorithms to all non-privileged users, and this "overly open" design philosophy needs to be re-examined.