Zero-click exploit: the 2025 WhatsApp-ImageIO chain

Write-Primitive Exploits in Production: From Zero-Click iOS Chains to Linux Kernel Root, 2016–2026

WAGNER Ştefan-Daniel
OLTEAN Dan-Gabriel MATVEEV Victor-Nicolae
Facultatea de Ştiinţe Aplicate
Teoria Codării şi Stocării Informaţiei
Universitatea Naţională de Ştiinţă şi Tehnologie POLITEHNICA Bucureşti (UNSTPB)
Coordonator: conf. univ. dr. Emil Simion, FSA, `emil.simion@upb.ro`
2025–2026

***Abstract*—Zero-click exploits (attacks that compromise a device without any user interaction) represent the most asymmetric threat in mobile security. This paper analyzes the 2025 WhatsApp–ImageIO zero-click exploit chain, which combined a WhatsApp linked-device authorization bypass (CVE-2025-55177, CWE-863) with a heap out-of-bounds write in Apple's ImageIO framework (CVE-2025-43300, CWE-787) to achieve remote code execution on iOS devices, targeting fewer than 200 journalists and civil-society figures over 90 days. We reconstruct the chain from public primary sources (vendor advisories, patch diffs, and independent reverse engineering) and use the Exploit Education Phoenix `heap-two` exercise as a pedagogical bridge to demonstrate that the core exploitation primitive (heap overflow into an adjacent function pointer) is identical at the textbook and production levels. Comparing the 2025 chain with the structurally identical FORCEDENTRY chain (2021, CVE-2021-30860), we argue that the "delivery primitive plus image parser memory corruption" pattern is inherent to the current architecture of mobile media processing and that the deployed mitigation stack (ASLR, sandboxing, pointer authentication) raises the cost of exploitation without eliminating the root cause. We recommend incremental rewriting of high-exposure C/C++ parsers in memory-safe languages and deployment of hardware memory tagging as structural countermeasures.**

I. Introduction

In August 2025, Apple and WhatsApp released emergency advisories for CVE-2025-43300 and CVE-2025-55177, a pair of vulnerabilities that, chained together, had already been used in the wild to compromise the mobile devices of fewer than 200 users over the preceding 90 days [1], [2]. Independent reporting identified many of the targets as journalists, human-rights defenders, and civil-society figures [3]. The chain required no clicks, no downloads, and no error on the victim's part: receiving a WhatsApp message was sufficient. Both CVEs were added within weeks to the U.S. Cybersecurity and Infrastructure Security Agency (CISA) Known Exploited Vulnerabilities (KEV) catalog [4], mandating remediation across federal networks.

Zero-click chains of this shape have become the defining offensive capability of commercial spyware. Unlike phishing, which relies on social engineering, and unlike drive-by compromise, which relies on the victim visiting an attacker-controlled URL, a zero-click chain weaponizes the infrastructure that modern mobile operating systems use to render inbound media. Knowledge of the victim's phone number is sufficient; no further action on the victim's part is required. The asymmetry of this attack model (bounded defender effort against unbounded attacker opportunity) makes zero-click delivery the most consequential primitive in targeted surveillance today [5].

What makes the 2025 chain technically striking is its genealogy. Both vulnerabilities fire in code written in C, a language first standardized in 1989 whose memory model (Section II) provides no enforcement of bounds, type safety, or lifetimes. The exploitation stage is a heap out-of-bounds write in an image parser, a bug class whose stack-based ancestor was first formalized by Aleph One in 1996 [6] and whose heap-resident variant has ranked second in the MITRE CWE Top 25 in 2024 (score 45.20, 18 entries in the CISA Known Exploited Vulnerabilities catalog) [7], [8] and remained the highest-ranked memory-corruption class in 2025 [9]. Industry-wide, memory-safety bugs account for approximately 70 % of all CVEs patched annually by major software vendors—a figure confirmed independently by Microsoft's Security Response Center across twelve years of patch analysis [10] and by Google's Chromium team across hundreds of high-severity browser bugs [11]. Despite repeated calls from CISA and the U.S. Office of the National Cyber Director for migration to memory-safe languages [12], the media-parsing frameworks at the core of every major mobile platform remain overwhelmingly C and C++. The result is a structural

mismatch: a decades-old memory model is entrusted with the first parsing step of every image delivered to every networked device.

This is not the first time the pattern has produced high-profile civilian casualties. In 2021, Citizen Lab and Google Project Zero reverse-engineered FORCEDENTRY (CVE-2021-30860), an NSO Group Pegasus zero-click chain with nearly identical architecture: iMessage delivery, CoreGraphics image parsing, heap corruption [13], [14]. Four years later, the 2025 WhatsApp–ImageIO chain reproduces the same three-step pattern in a different ecosystem, messenger, image format, and parser, with the same outcome. This cross-platform recurrence is the central empirical observation motivating this study. As this manuscript was being finalized, two further write-primitive Linux kernel root exploits were disclosed in quick succession: Copy Fail (CVE-2026-31431, April 2026, `AF_ALG` crypto subsystem) [15] and DirtyFrag (CVE-2026-43284, May 2026, xfrm-ESP networking) [16]—providing real-time evidence that the broader write-primitive exploit class documented in Section II-F is ongoing and not confined to the iOS ecosystem.

Research question. This paper asks: *how do decades-old memory-corruption patterns in C enable modern zero-click exploit chains, and what does the 2025 WhatsApp–ImageIO attack reveal about the persistence of these vulnerabilities in industry-critical software?*

Contributions. This paper makes three contributions:

1) A reconstruction of the 2025 zero-click chain from public primary sources (vendor advisories, patch diffs, and independent reverse engineering [17]), covering the delivery primitive (CVE-2025-55177, CWE-863) and the exploitation primitive (CVE-2025-43300, CWE-787), and showing how the two stages compose into a complete, interaction-free compromise.
2) A pedagogical bridge from the Exploit Education Phoenix `heap-two` exercise to CVE-2025-43300, demonstrating that the core mechanic (a heap buffer overflowed into adjacent function-pointer-bearing data) is the same primitive at two drastically different complexity levels. The bridge is intended for readers whose prior exposure to memory corruption stops at textbook stack smashing.
3) A structural argument that the FORCEDENTRY → WhatsApp–ImageIO lineage, spanning four years and two independent ecosystems, is evidence that the current mitigation stack (ASLR, sandboxing, pointer authentication) has not closed the fundamental attack class; it only raised its price.

Scope and non-goals. This paper does not present a new exploit, a new vulnerability, or proprietary reverse engineering. All technical material is drawn from publicly disclosed sources. The Phoenix `heap-two` exercise is treated as a controlled pedagogical artifact, not a substitute for the real chain.

Roadmap. Section II introduces the C memory model, the two CWE classes at play, the iOS media pipeline, the FORCEDENTRY precedent, and the decade-long recurrence of write-primitive exploits across iOS and Linux. Section III describes our CVE selection criteria, analysis framework, and Phoenix setup. Section IV is the core technical analysis of the 2025 chain. Section V reflects on why the current mitigation stack fails to close this attack class. Section VI concludes.

II. Background

This section establishes the minimum technical vocabulary needed to follow the two-vulnerability chain dissected in the case study. Readers are assumed to be computer-science graduates without a specialization in systems security; every term is introduced the first time it is used. We cover, in order, the C memory model (Section II-A), the two weakness classes that appear in the chain (Section II-B), the iOS media-processing pipeline in which both vulnerabilities fire (Section II-C), the mitigations that are supposed to prevent exploitation (Section II-D), the 2021 FORCEDENTRY incident which is the historical precedent for the 2025 attack (Section II-E), and the broader decade-long recurrence of write-primitive exploits across iOS and Linux (Section II-F).

A. The C Memory Model: A Brief Tour

C is a language that trades safety for control. Unlike managed languages such as Java, Python, or Swift, C gives the programmer direct access to raw memory through pointers, with no runtime enforcement of object bounds, type safety, or lifetimes. A running C program organizes its memory into several regions; two are central to exploitation. The *stack* is a last-in-first-out region where local variables and function call frames live. Every function call pushes a new frame containing parameters, locals, and the saved return address: the instruction to jump to after the callee returns. The *heap* is a region for dynamically allocated memory (via `malloc`, `calloc`, `new`). Blocks are allocated and freed explicitly by the programmer; the heap allocator maintains bookkeeping metadata (block size, free/used flags, chunk pointers) interleaved with or adjacent to user data.

A pointer in C is simply an integer that names a memory address. Dereferencing a pointer (`*p` for read, `*p = v` for write) performs raw memory access with no check that `p` points to a valid, in-bounds location of the expected type. This is the root of the entire memory-corruption family: if an attacker can drive a write slightly past where it was supposed to land, they can alter state (other variables, function pointers, return addresses, allocator metadata) that the program will later trust and dereference. Aleph One's 1996 essay "Smashing the Stack for Fun and Profit" [6] first formalized the stack-based version of this idea; three decades later, the same pattern continues to produce exploitable bugs at the top of the

MITRE CWE Top 25 list [8]. The heap-specific variant of this idea was formalised in 2001 by the Phrack article "Once upon a free()" [18], which showed how `dlmalloc`'s bookkeeping metadata (stored adjacent to user data in the same heap region) could be weaponised into a write-anywhere primitive via the allocator's own unlink routine; three decades later, the structural insight remains valid. Listing 1 shows the minimal C pattern that produces this class of bug.

Listing 1. Minimal heap out-of-bounds write. If `len > 64`, bytes spill from `buf` into adjacent heap memory, corrupting allocator metadata or attacker-reachable data (CWE-787).

```
/* Allocate a 64-byte heap chunk. */
char *buf = malloc(64);

/* No bounds check: if len > 64, bytes
   at buf[64..len-1] are written into
   the *next* heap chunk.              */
memcpy(buf, attacker_input, len);

/* The corruption is invisible here --
   it surfaces later, when the program
   dereferences overwritten state.   */
```

B. Vulnerability Taxonomy

Two CWE classes drive the attack chain studied in this work.

CWE-787: Out-of-bounds write [8] is the class in which the program writes past the intended end of a buffer. On the heap, an out-of-bounds write may corrupt adjacent allocations or allocator bookkeeping; on the stack, it may corrupt saved registers or the return address. CWE-787 has held the #1 position in the MITRE Top 25 Most Dangerous Software Weaknesses for multiple consecutive years, and memory corruption remains one of the most consequential bug classes in systems software [19]. CVE-2025-43300, the Apple ImageIO vulnerability at the core of the chain studied here, is a CWE-787 instance.

CWE-863: Incorrect authorization [20] is categorically different. No bits are smashed and no memory is corrupted; the bug is a purely logical failure to enforce an access-control policy on a correctly authenticated principal. CVE-2025-55177, the WhatsApp linked-device synchronization bug, is a CWE-863 instance: the WhatsApp client trusts a synchronization message whose authorization scope was never fully verified, and the attacker uses this trust to coerce the victim device into processing attacker-controlled content. The relevance for this paper is that CWE-863 provides the *delivery* primitive that places the malicious image into the victim's media pipeline in the first place. Memory corruption alone is not enough for a zero-click exploit: the bug must be *reachable* without user interaction, and reachability is exactly what the authorization bug provides.

C. The iOS Media Processing Pipeline

On Apple platforms, almost every application that displays an image (Messages, Safari, Mail, WhatsApp, Signal) delegates decoding to a shared framework: ImageIO, together with its sister framework CoreGraphics. ImageIO is a C / Objective-C library responsible for parsing image container formats (JPEG, PNG, HEIF, DNG, GIF, TIFF, and several camera-specific RAW variants) and producing decoded pixel buffers. Because ImageIO is shared across the operating system, any parser bug inside it is reachable from every application that displays an image received from the network.

The critical observation is that ImageIO is invoked *automatically* the moment an incoming image enters the pipeline for thumbnail generation, *before* the user has seen, tapped, or acknowledged the message. This turns any parser bug in ImageIO into a zero-click attack primitive: the mere delivery of an image is sufficient to trigger code execution if the bug can be driven to memory corruption. In the WhatsApp–ImageIO chain, the end-to-end pipeline is:

1) WhatsApp receives a message via its linked-device synchronization protocol.
2) The message carries an image attachment.
3) To render a thumbnail, WhatsApp hands the raw bytes to CoreGraphics/ImageIO.
4) ImageIO dispatches to the parser for the declared format (DNG in this case).
5) The parser reads attacker-controlled header fields and triggers an out-of-bounds write.

None of these steps requires user interaction.

D. Modern Mitigations and Their Limits

Three decades of research have produced a layered defense against memory corruption [19], [21]. Three mitigations are most relevant for iOS and together define the landscape in which the 2025 chain had to operate.

Address Space Layout Randomization (ASLR) randomizes the base addresses of code and data regions at process start, so an attacker who wants to redirect control flow to a specific function must first *leak* an address from the victim process. ASLR does not prevent memory corruption; it raises the cost of turning corruption into useful code execution.

Stack canaries place a random value just before the saved return address on each stack frame. The function checks the canary before returning, and a mismatch terminates the process. Canaries protect the return address specifically, not arbitrary function pointers, not heap metadata, and not object vtables.

Application sandboxing runs each app inside a kernel-enforced container that restricts file-system, network, and IPC access. A successful RCE inside an app is therefore not immediately full device compromise: a separate sandbox escape is needed to reach kernel or cross-app territory.

These defenses raise the exploitation bar but leave structural gaps. ASLR can be bypassed with an information-leak primitive: often another parser bug in the same process [19]. Stack canaries do nothing for heap-resident

function pointers, which are precisely what a heap overflow targets. Sandboxes reduce blast radius but do not prevent the initial compromise. The 2025 chain studied here operates *entirely within* these constraints: the parser bug is exploited inside a sandboxed process, and follow-on stages (privilege escalation, persistence) are layered on top.

E. Historical Precedent: FORCEDENTRY (2021)

In 2021, Citizen Lab and Google Project Zero documented *FORCEDENTRY* (CVE-2021-30860), a zero-click exploit deployed against Saudi activists and journalists via NSO Group's Pegasus spyware [13], [14]. The architecture is nearly identical to the 2025 chain studied in this paper:

- **Delivery:** the attacker sends an iMessage to the victim's phone number; no interaction is required.
- **Exploitation:** iMessage invokes CoreGraphics to render a malicious PDF disguised as a GIF. The PDF contains a JBIG2-compressed stream; a bug in the JBIG2 parser lets the attacker construct an arbitrary computation primitive inside the parser's memory, eventually achieving code execution.
- **Outcome:** full device compromise before the victim sees the message.

FORCEDENTRY proved that a memory-corruption bug in a shared image parser, combined with a delivery primitive that needs no user interaction, is sufficient for complete device takeover. Four years later, the 2025 WhatsApp–ImageIO chain reproduces the same architecture with different components: a different messenger (WhatsApp instead of iMessage), a different authorization bypass (linked-device sync confusion instead of direct delivery), a different parser (DNG in ImageIO instead of JBIG2 in CoreGraphics), and a different carrier format, but the same structural weakness and the same outcome. The persistence of this pattern across four years and across two independent messaging ecosystems is the central empirical observation of this paper, and motivates the technical analysis that follows.

F. The Recurrence of Write-Primitive Exploits, 2016–2026

The structural parallel between FORCEDENTRY and the 2025 chain is not an isolated coincidence but a reflection of a broader, decade-long trend. Two complementary data sets establish the baseline.

At the *vulnerability-class level*, memory-corruption weaknesses have dominated the MITRE CWE Top 25 Most Dangerous Software Weaknesses list throughout the period. CWE-787 (Out-of-bounds Write) ranked second in the 2024 edition with a score of 45.20 and 18 entries in the CISA Known Exploited Vulnerabilities catalog [7], [8], and remained the highest-ranked memory-corruption class in 2025 [9]. At the industry level, memory-safety bugs account for approximately 70 % of all CVEs patched annually by major software vendors: a figure confirmed independently by Microsoft's Security Response Center across twelve years of patch data [10] and by Google's Chromium team across 912 high-severity browser bugs since 2015 [11].

TABLE I
A DECADE OF WRITE-PRIMITIVE EXPLOITS IN PRODUCTION (2016–2026). DIVERSE VULNERABILITY CLASSES CONVERGE ON THE SAME OUTCOME: AN UNAUTHORIZED WRITE TO PRIVILEGED MEMORY ENABLING LOCAL ROOT (LINUX) OR SANDBOXED RCE (IOS). CWE LEGEND: 120 = BUFFER COPY W/O SIZE CHECK, 123 = WRITE-WHAT-WHERE, 190 = INTEGER OVERFLOW, 362 = RACE CONDITION, 416 = USE-AFTER-FREE, 665 = IMPROPER INITIALIZATION, 669 = INCORRECT RESOURCE TRANSFER, 787 = OUT-OF-BOUNDS WRITE.

Year	Chain	CVE	CWE	Outcome
2016	DirtyCow [22]	CVE-2016-5195	362	Linux root
2021	FORCEDENTRY [13]	CVE-2021-30860	190	iOS RCE
2022	DirtyPipe [23]	CVE-2022-0847	665	Linux root
2023	BLASTPASS [5]	CVE-2023-41064	120	iOS RCE
2023	Triangulation [5]	CVE-2023-32434	190	Kernel RCE
2023	StackRot [24]	CVE-2023-3269	416	Linux root
2025	This work	CVE-2025-43300	787	iOS RCE
2026	Copy Fail [15]	CVE-2026-31431	669	Linux root
2026	DirtyFrag [16]	CVE-2026-43284	123	Linux root

At the *exploit level*, the decade 2016–2026 produced at least one high-profile write-primitive root or RCE exploit per year across the Linux kernel and the iOS media-processing ecosystem. Table I lists nine representative chains. The entries span eight distinct CWE classes (defined in the table caption) across unrelated subsystems: kernel memory management, pipe page cache, cryptographic API, IPsec networking, and image parsers. Their common denominator is not a single weakness class but a shared exploitation outcome—an unauthorized write to privileged memory that converts into either local root (Linux) or remote code execution (iOS). The four iOS-targeting chains all use parser bugs as the initial write primitive; the five Linux kernel chains exploit different subsystems but reach the same privilege- escalation endpoint.

The recurrence is ongoing. Copy Fail (CVE-2026-31431) was disclosed on April 29, 2026, exploiting an in-place optimization flaw in the kernel `AF_ALG` crypto interface (CWE-669) to achieve a deterministic 4-byte write into arbitrary kernel page-cache frames [15], [25], achieving local root on a fully patched Ubuntu 24.04 with SMAP, SMEP, and KASLR enabled. Nine days later, DirtyFrag (CVE-2026-43284) was disclosed on May 7, 2026, exploiting the xfrm-ESP IPsec networking stack (CWE-123) via the same page-cache write primitive [16], [26]. Both were actively exploited within days of disclosure. These two disclosures—arriving within a single week—illustrate that the write-primitive exploit class is not converging toward zero: the subclass of bug and the target subsystem change with every disclosure; the exploitation outcome does not.

III. Methodology

This paper is a technical case study, not an empirical measurement study. Our methodology is therefore organized around the evidence we use and the inferential

framework we apply to it, rather than around an experimental protocol.

A. Case Selection Criteria

We selected the CVE-2025-55177 + CVE-2025-43300 chain as the primary case using four inclusion criteria:

1) **Real-world exploitation**, confirmed by inclusion in the CISA KEV catalog [4] and by contemporaneous reporting on civil-society victims.
2) **Public primary documentation** sufficient to reconstruct the chain without privileged access: vendor advisories, NVD entries, patch diffs, and at least one independent reverse-engineering report.
3) **C-class memory corruption at exploitation**, so that the case speaks directly to the research question.
4) **Zero-click end-to-end**, with no user action required between message receipt and code execution.

FORCEDENTRY (CVE-2021-30860) also satisfies all four criteria and is used as a historical comparison point (Section II-E) rather than as a second primary case. Isolated CVEs that require tap-to-open interaction, browser navigation, or installation of an attacker-controlled application were excluded, because their threat model differs qualitatively from the zero-click class studied here.

B. Analysis Framework

Each CVE is dissected along three axes. The *root cause* is the specific source-level condition (call site, allocation, missing check) that makes the bug possible. For CVE-2025-43300, this is the `SamplesPerPixel` vs. `SOF3` component-count mismatch in Apple's `CDNGLosslessJpegUnpacker`, reconstructed by Quarkslab from patch diffing of iOS 18.6.2 and macOS Sequoia 15.6.1 [17]. The *patch* is the vendor-supplied fix, treated as evidence of root cause: which check was added, which path was restricted, which validation was strengthened. Patch diffs are the most reliable external ground truth available to analysts without vendor access. *Position in the chain* locates the bug end-to-end: delivery primitive (CVE-2025-55177 grants reachability without interaction), exploitation primitive (CVE-2025-43300 turns reachability into remote code execution), or a glue step connecting the two.

The three-axis framework is applied symmetrically to both CVEs in Section IV and to FORCEDENTRY in Section II-E, enabling a like-for-like comparison across four years and two independent ecosystems.

C. Pedagogical Bridge: Phoenix `heap-two`

To make the exploitation mechanics legible to readers without prior heap-exploit experience, we conduct a full end-to-end walkthrough of the `heap-two` exercise from the Exploit Education Phoenix VM. The exercise is chosen for three reasons:

Listing 2. Illustrative heap-two structure. Two malloc calls produce adjacent chunks; strcpy performs no bounds check, so input longer than DATA_SZ overwrites Chunk B's function pointer.

```
#define DATA_SZ 64

struct item    { char data[DATA_SZ]; };
struct handler { void (*fp)(void);   };

struct item    *A = malloc(sizeof *A);
struct handler *B = malloc(sizeof *B);
B->fp = &safe_function;   /* default */

/* No bounds check: if strlen(input)
   exceeds DATA_SZ, bytes spill from
   A->data into B->fp.                */
strcpy(A->data, user_input);

B->fp();   /* attacker-chosen target */
```

1) Its vulnerability class is CWE-787, the same class as CVE-2025-43300.
2) The primitive it grants (overwrite of a function pointer adjacent on the heap) is the closest textbook analogue of what the ImageIO bug ultimately yields.
3) It is small enough (50 lines of C) for the entire heap state to be inspected in `gdb` without losing the thread.

a) Setup: Installing and building `heap-two`.: The Exploit Education Phoenix VM is freely available from exploit.education [27]. The `heap-two` source code resides in `/opt/phoenix/heap-two/heap-two.c`. We build the binary with:

```
gcc -o heap-two heap-two.c
```

The Phoenix VM ships the exercise binary compiled without modern protections, by design. The mitigation state is confirmed with `checksec`:

```
$ checksec --file=heap-two
```

All major flags are disabled: no ASLR, no NX, no stack canary, no PIE. This configuration makes addresses deterministic and overflow mechanics fully observable without randomization obscuring the analysis.

b) Program structure and vulnerability.: The `heap-two` binary allocates two heap chunks and prompts for user input. The first chunk is a `char` buffer; the second is a struct containing a function pointer. A call to `strcpy` copies user input into the first buffer without bounds checking. Input longer than the allocated size causes overflow, corrupting the function pointer in the second chunk with an attacker-chosen value. Listing 2 shows the essential pattern.

c) Execution workflow: `gdb` session.: We run the binary under `gdb` to observe heap layout and exploitation step-by-step:

1) Start the binary: `gdb ./heap-two`
2) Set breakpoint before vulnerable `strcpy`
3) Continue to breakpoint: `run`

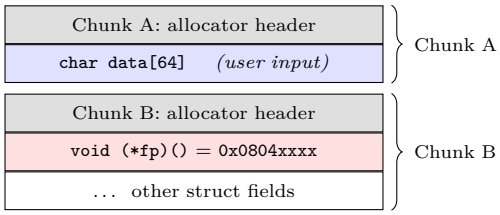


Fig. 1. Heap layout *before* overflow. Chunk A (64-byte user buffer) is adjacent to Chunk B (struct with function pointer). A conforming write stays within Chunk A's bounds.

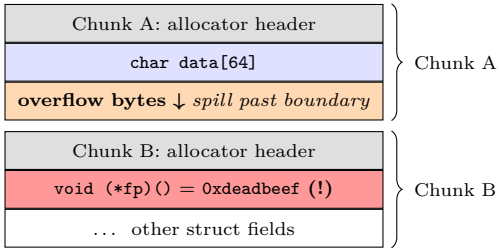


Fig. 2. Heap layout *after* overflow. Attacker input exceeds 64 bytes; excess bytes spill into Chunk B, overwriting the function pointer with an attacker-chosen value (`0xdeadbeef`).

4) Inspect heap state before overflow: `x/20x` at Chunk A's address (exact address obtained from prior breakpoint)
5) Identify the function pointer location
6) Step through `strcpy`: `ni`
7) Re-examine memory after overflow
8) Observe the overwritten function pointer

d) Heap layout and the overflow.: Figure 1 shows the heap before overflow: two `malloc` calls produce chunks A (user buffer, 64 bytes) and B (struct with function pointer). When `strcpy` copies attacker input without bounds checking, bytes beyond chunk A's boundary overwrite chunk B's function pointer. Figure 2 shows the heap after overflow, with chunk B's pointer rewritten to the address of a `winner()` function.

e) Exploitation: crafting the payload.: The attacker constructs a payload filling chunk A with padding, then encodes the `winner()` function address in little-endian format to overwrite the pointer in chunk B. The program crashes at the attacker-chosen address: a successful exploit. Figure 3 confirms this: the left panel shows the heap state after overflow, with the victim buffer (blue), the overflowing bytes that corrupt Chunk B's header (orange), and the function pointer rewritten to `winner()`'s address (red); the right panel shows GDB at the `call` instruction, with `$eax` pointing to `winner()`.

f) Connection to CVE-2025-43300.: The structural parallel is exact: both grant a controllable heap write that overwrites an adjacent function pointer, confirmed above by Figure 3. Section IV draws the structural comparison to CVE-2025-43300 in Figure 5, which aligns the two heap layouts side-by-side against Quarkslab's reconstruction of the ImageIO decoder [17].

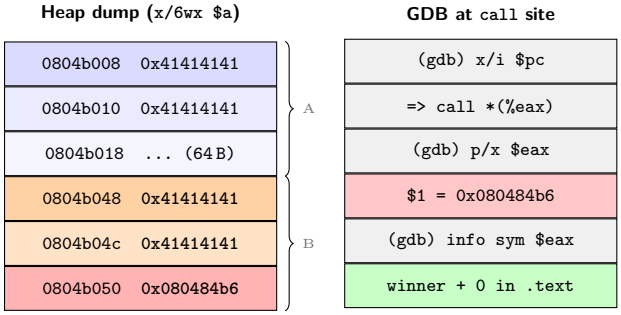


Fig. 3. `gdb` session at the moment of the overwritten function pointer in `heap-two`. The target chunk address and the overwriting bytes are visible in the heap dump (left); the corrupted function pointer is visible in the register state after the `call` instruction (right).

TABLE II
TOOLS AND PRIMARY SOURCES USED IN THIS STUDY.

Tool / source	Role
`gdb`	Heap and register inspection
`checksec`	Mitigation-state verification
Phoenix VM [27]	Environment for `heap-two`
Quarkslab report [17]	CVE-2025-43300 root cause
Apple, WhatsApp advisories	Canonical CVE descriptions
NVD entries	CWE class, CVSS, affected versions
CISA KEV [4]	Exploitation confirmation

D. Tools and Primary Sources

All materials used in this study are publicly available; a summary is given in Table II. No proprietary data, no non-public code, and no new reverse engineering of Apple or WhatsApp binaries is produced or relied upon. The paper's analytical contribution is in synthesis and pedagogical framing, not primary discovery.

IV. Case Study: The 2025 Zero-Click Chain

In late August 2025, WhatsApp began notifying a set of users that their devices had been targeted in a "sophisticated cyber attack" combining a WhatsApp-specific authorization flaw (CVE-2025-55177) with a separately patched Apple ImageIO memory-corruption bug (CVE-2025-43300) [1]–[3]. The chain is the central object of study in this paper. Figure 4 overviews the end-to-end flow, which we dissect in three stages: delivery (Section IV-A), exploitation (Section IV-B), and the mapping from the textbook `heap-two` primitive to the real CVE (Section IV-C).

A. Stage 1 – Delivery: CVE-2025-55177

WhatsApp supports a multi-device model in which a primary account (a phone number registered on a phone) can be paired with up to four additional endpoints: WhatsApp Web, Desktop, iPad, and so on. The feature is built on an extension of the Signal Protocol: the primary



Fig. 4. End-to-end flow of the 2025 zero-click chain. Red nodes mark the two patched CVEs. Only the victim's phone number is required; no action by the victim is needed for the chain to reach code execution.

device holds the master identity key, and each linked device receives derived session material plus a stream of *synchronization messages* that replicate state changes (new chats, read receipts, media attachments) between devices [28].

The pairing handshake is the security-critical step. A legitimate pairing requires the primary device to scan a QR code displayed by the candidate linked device; the two devices then exchange public keys and the primary device records the new endpoint in its enrolled-device set. Synchronization messages are subsequently sent over the established secure channel and, by design, are authenticated against that enrolled set.

a) The bug.: Per WhatsApp's August 2025 security advisory [1], an "incomplete authorization of linked device synchronization messages" allowed a remote attacker to cause the victim's WhatsApp client to process sync content from a device whose enrollment had not been verified. Concretely, the client's handling of a subset of sync message types trusted an identifier embedded *in the message itself* to decide whether the sender was an authorized linked device, rather than consulting an independently maintained enrollment record. An attacker in possession of the victim's WhatsApp account identifier could therefore deliver crafted sync messages that the client would accept and process, including messages referencing attacker-controlled URLs for media attachments. The weakness is classified under CWE-863 (Incorrect Authorization) [20]; CVSS v3.1 scored 5.4 (Medium) on its own, without accounting for downstream impact when chained.

b) From authorization bypass to content processing.: The delivery path ends when the bypassed sync message instructs the victim's client to fetch and render an attacker-chosen image. WhatsApp, like most modern messengers, displays thumbnails for received images automatically in the chat-list preview; rendering invokes the system's shared image decoder (on iOS and macOS, CoreGraphics and ImageIO). The victim's client therefore downloads and parses the attacker's image *before* the victim has seen the message, producing the interaction-free attack surface on which Stage 2 operates.

c) Why this stage is zero-click.: Three properties together are what make Stage 1 zero-click in practice:

1) **No interaction to receive:** the sync message is delivered and processed by a background WhatsApp component, not by user-level code that requires the app to be foregrounded.
2) **No interaction to process attachments:** thumbnail generation runs at message arrival, not at message tap.
3) **No evidence before compromise:** the vulnerable code path executes before any visible UI change; the victim has no opportunity to decline.

d) Patch.: WhatsApp's fix, shipped in WhatsApp for iOS 2.25.21.73 and matching versions on Android, Desktop, and Business, tightened the authorization check on the affected sync message types so that only messages signed by a verifiably enrolled linked device are accepted [1]. After patching, the delivery primitive is closed: the exploitation primitive studied in Stage 2 remains reachable only through other code paths that feed attacker-controlled bytes to ImageIO (e.g., other messengers, browser download previews, Mail attachments).

B. Stage 2 – Exploitation: CVE-2025-43300

Once Stage 1 has forced the victim's device to fetch and render an attacker-chosen image, the second CVE takes over. CVE-2025-43300 is a heap out-of-bounds write (CWE-787) in Apple's ImageIO framework, specifically in the handling of lossless-JPEG streams embedded inside DNG (Digital Negative) raw image containers. It was patched by Apple on 20 August 2025 in iOS 18.6.2 and macOS Sequoia 15.6.1 with the advisory note that "Apple is aware of a report that this issue may have been exploited in an extremely sophisticated attack against specific targeted individuals" [2].

a) Container format: DNG.: DNG is a TIFF-based container defined by Adobe for raw camera images. A DNG file consists of one or more Image File Directories (IFDs), each carrying tagged metadata describing the stored pixel data. Two tags are directly relevant:

- `SamplesPerPixel` (tag 277): the number of samples per pixel, typically 1 for single-channel grayscale or 3 for RGB.
- `Compression` (tag 259): when set to 7 (lossless JPEG), the pixel payload is a lossless-JPEG-compressed stream embedded inside the TIFF and referenced through `StripOffsets`/`TileOffsets`.

Inside the lossless-JPEG stream, the component count is given *independently*, by the `Nf` field of the SOF3

(Start of Frame 3, lossless JPEG) marker. In a well-formed DNG, `Nf` and `SamplesPerPixel` must agree: the container-level count of samples per pixel and the stream-level count of JPEG components describe the same quantity.

b) The bug.: The ImageIO decoder for this configuration is implemented in `CDNGLosslessJpegUnpacker` (private to ImageIO; symbol names here follow Quarkslab's reconstruction [17]). The decoder allocates its output buffer using `SamplesPerPixel` taken from the TIFF IFD, then drives the decoding loop using `Nf` taken from the SOF3 marker of the embedded lossless JPEG. When an attacker crafts a DNG in which `SamplesPerPixel = 1` but `SOF3.Nf = 3`, the decoder allocates a buffer sized for one component and writes three components' worth of decoded data into it, overflowing by a factor of three [17].

The overflow is useful because it is:

- **Controllable in size:** the attacker chooses both `SamplesPerPixel` and `Nf`, so the overflow length is set by simple arithmetic on attacker-chosen inputs.
- **Controllable in content:** the overflowing bytes are derived from the lossless-JPEG payload, which is attacker-chosen end to end.
- **Heap-adjacent:** the target allocation lives on the heap used by ImageIO for pixel buffers; adjacent objects on that heap include decoder state, buffer metadata, and function pointers that are dereferenced by subsequent decoding code.

Quarkslab summarizes the pattern as "two bytes that make size matter" [17]: the disagreement between a single-byte count in the TIFF header and a single-byte count in the SOF3 marker is the entire bug.

c) From overflow to code execution.: The gap between "heap OOB write" and "code execution on an iPhone" is large in general but has a well-understood shape. First, shape the heap (via prior allocations performed by the parser itself) so that an object whose type contains a function pointer lands within overflow range of the victim buffer and will later be dereferenced by the decoder. Second, overflow precisely enough to overwrite that pointer without corrupting intervening metadata. Third, when the decoder later invokes the overwritten pointer, control transfers to the attacker-chosen address.

On iOS, these steps are further constrained by ASLR, by pointer authentication codes (PAC) on ARMv8.3+ Apple silicon that sign pointers with a process secret, and by the app sandbox which limits the power of a code-execution primitive inside ImageIO. Full Stage 2 exploitation in the wild therefore requires additional primitives (information leaks, PAC bypasses, sandbox escapes) that are not part of CVE-2025-43300 itself but are necessary in practice. Section IV-C draws out the structural parallel to the textbook `heap-two` exercise explicitly.

d) Patch.: Apple's fix, inferred from diffing iOS 18.6.1 against iOS 18.6.2 per Quarkslab's analysis [17], adds a consistency check ensuring that `SamplesPerPixel` and the embedded `SOF3.Nf` agree before the decoding loop begins. Inputs for which the two quantities disagree are rejected at parse time; no allocation/decoding mismatch is reachable on the patched code path. The patched binary adds on the order of a few bytes of validation to prevent an overflow whose severity is measured in complete device compromise, an asymmetry common to parser bugs of this shape.

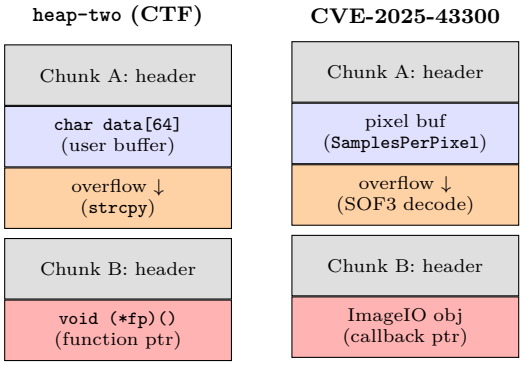


Fig. 5. Side-by-side heap layout. Left: `heap-two`: chunk A holds an attacker-copied string, chunk B holds a function pointer `fp`; `strcpy` overflows A into B. Right: CVE-2025-43300, where chunk A is the pixel buffer sized by `SamplesPerPixel`, chunk B is an internal ImageIO object with a dereferenced callback; the SOF3-driven decode loop overflows A into B.

C. From CTF to CVE

At the level of exploit primitive, `heap-two` and CVE-2025-43300 are the same bug. Both grant an attacker a controllable heap write that extends past the allocated bounds of an attacker-sized buffer into attacker-reachable adjacent state. Both terminate in a function-pointer overwrite that is later dereferenced by the vulnerable program. The difference between the 20-line Phoenix binary and a production zero-click chain is engineering, not mechanism.

a) The mechanism, aligned.: Figure 5 aligns the two heap layouts. In `heap-two`, `malloc` is called twice; the first chunk is the user-controllable buffer and the second contains a function pointer `fp` that the program later calls. The call to `strcpy` copies attacker-controlled data into the first chunk without respecting its allocated size, and bytes past the end of that chunk fall exactly onto `fp` in the second chunk. Overwriting `fp` with the address of the target function `winner()` redirects execution. In CVE-2025-43300, the first chunk is the pixel buffer sized by `SamplesPerPixel`; the second chunk is, depending on heap shape, an internal ImageIO object whose virtual-method pointer or callback is subsequently dereferenced during image processing. The rest is the same.

b) What scales from textbook to production.: Productionizing the primitive focuses on two engineering problems. First, the textbook exercise runs with predictable addresses because ASLR is disabled in the educational binary; a real exploit must first acquire an information leak, often by exploiting a second, milder bug in the same

process, to locate its target. Second, the textbook exercise enjoys a deterministic heap layout because `malloc` is called twice in the attacker's order; the real exploit must shape a live, concurrent allocator (`libmalloc` on iOS) so that the attacker's target object reliably lands within the overflow range of the victim buffer. The standard technique is *heap grooming*: forcing the vulnerable parser to allocate a series of predictable-size buffers that reshape free-list structure, then triggering the vulnerable allocation at a point where the target object is the next chunk the allocator will return.

c) What does not scale.: The bug class itself (out-of-bounds write on the heap with attacker control over size and content) is identical across cases. This is the core observation driving the research question: the fundamental memory-corruption mechanism that a student can learn in an afternoon on an educational VM is the same one that compromised fewer than 200 mobile devices in mid-2025. Mitigations (ASLR, heap randomization, PAC, sandboxing) raise the cost of Stage 2 exploitation but do not alter the structural vulnerability. Section V examines why this mitigation model has structural limits.

V. Discussion

The 2025 WhatsApp–ImageIO chain and its 2021 FORCEDENTRY predecessor demonstrate a consistent architectural pattern: a delivery primitive (messaging protocol bypass) composes with an exploitation primitive (image parser memory corruption) to achieve zero-click code execution. This section examines why the current mitigation stack did not prevent the 2025 chain, what lies beyond the initial RCE, what structural alternatives exist, and the limitations of this study.

A. Why the Mitigation Stack Did Not Prevent This Chain

Section II-D introduced three mitigations deployed on iOS: ASLR, stack canaries, and application sandboxing. None of them target the root cause of CVE-2025-43300.

ASLR randomizes memory layout but does not prevent the out-of-bounds write itself. The write occurs regardless of where the buffer is placed; ASLR only makes it harder to know where to redirect control flow afterward. In practice, a second, milder bug in the same process (an information leak) suffices to defeat ASLR by revealing a code or data address, after which the attacker can construct a valid target for the hijacked pointer. Information leaks in image parsers are common: the same complexity that produces out-of-bounds writes routinely produces out-of-bounds reads [19].

Stack canaries are entirely irrelevant: the overflow occurs on the heap, not the stack. Canaries protect saved return addresses; they do not monitor heap metadata or function pointers stored in heap-allocated objects.

Application sandboxing limits what the attacker can do after RCE but does not prevent the RCE itself. The sandbox around WhatsApp restricts file-system and IPC access, but the attacker's initial foothold (code execution inside the process) is already sufficient to exfiltrate data accessible to that process (contacts, messages, media, location) and to attempt a sandbox escape via a separate vulnerability.

Pointer Authentication Codes (PAC), deployed on ARMv8.3+ Apple silicon [29], sign pointers with a process-specific secret so that corrupted pointers fail verification before use. PAC raises the cost of control-flow hijacking significantly but has been bypassed in academic and in-the-wild settings, typically by reusing legitimately signed pointers or by targeting code paths that do not enforce PAC on every indirect call.

The pattern is consistent: each mitigation addresses a *symptom* (where code lives, whether the return address is intact, what permissions the process has, whether a pointer is signed) rather than the *root cause* (the program wrote past the end of a buffer because no bounds check existed). So long as the parser is written in a language that permits unchecked memory access, the vulnerability class persists and mitigations can only raise the cost of exploitation, not eliminate it.

This failure is not confined to iOS. Table I in Section II-F documents nine write-primitive chains from 2016 to 2026 spanning five distinct CWE classes across the Linux kernel and the iOS media pipeline. The Linux entries cover kernel memory management, pipe page cache, the cryptographic API, and IPsec networking—entirely different subsystems, yet all bypass identical mitigation sets (SMAP, SMEP, KASLR) and reach the same local root primitive. Two of those entries arrived during the final week of preparation of this manuscript. Copy Fail (CVE-2026-31431, disclosed April 29, 2026) [15] exploits an incorrect resource transfer in the `AF_ALG` crypto interface to write four bytes into arbitrary kernel page-cache frames, achieving deterministic root on a fully patched Ubuntu 24.04. Nine days later, DirtyFrag (CVE-2026-43284, disclosed May 7, 2026) [16], [26] repeats the same page-cache-write primitive through the xfrm-ESP IPsec subsystem. Both exploit the same SMAP/SMEP/KASLR-bypassing technique as DirtyPipe [23] four years earlier. The cadence—at least one high-profile root or RCE exploit per year since 2016, across Linux and iOS—indicates that the deployed mitigation approach has not reduced the annual production rate of exploitable write primitives.

B. Beyond RCE: Sandbox Escape and Persistence

A zero-click RCE inside a sandboxed process is not, by itself, full device compromise. The reported FORCEDENTRY and 2025 chains both achieved persistent, cross-application surveillance (call recording, message exfiltration, camera/microphone access), implying that additional stages followed the initial RCE: at minimum, a sandbox escape and a privilege-escalation step. These stages are outside the scope of this paper: they involve kernel vulnerabilities and system-daemon bugs unrelated to the two

CVEs studied, but their existence must be acknowledged: the image-parser RCE is the entry point, not the endpoint, of the full attack.

C. Structural Alternatives

If mitigations cannot close the vulnerability class, what can? Two categories of structural change address the root cause.

Memory-safe languages (Rust, Swift, Go, Java) enforce bounds checking, type safety, and lifetime management at compile time or runtime, making out-of-bounds writes impossible in safe code by construction [12]. Rewriting media parsers in a memory-safe language would eliminate CWE-787 from the attack surface entirely. The industry is already moving in this direction. Apple has begun shipping Swift-based replacements for some system components. Google's Android team has documented that the share of memory-safety vulnerabilities in new Android code fell to near zero in modules rewritten in Rust [12]. Most significantly, the Linux kernel (the canonical example of a safety-critical C codebase) merged initial Rust infrastructure in Linux 6.1 (December 2022) and shipped its first production Rust-written driver (a network PHY driver for the Asix ax88796b chipset) in Linux 6.8 (March 2024) [30]; Rust was declared a permanent, non-experimental part of the kernel at the 2025 Kernel Maintainer Summit in Tokyo [31]. At the distribution level, Ubuntu 25.10 (October 2025) became the first major distribution release to ship *uutils*, a Rust reimplementation of the GNU core utilities, as the default [32]. Ubuntu 26.04 LTS (April 2026) extended this to the long-term-support release and additionally shipped *sudo-rs*, a memory-safe Rust reimplementation of `sudo`, as the *default* privilege-escalation binary [33], replacing two C codebases that have long been high-value targets for local privilege escalation. The challenge for media frameworks is scale: ImageIO is a large, performance-critical C/Objective-C codebase with decades of accumulated optimisation, and incremental rewriting is slow, but uutils, sudo-rs, and the Linux Rust drivers demonstrate that production-grade, security-critical C code can be incrementally replaced.

Hardware-enforced memory safety, specifically ARM Memory Tagging Extension (MTE) on ARMv8.5+, assigns a 4-bit tag to every 16-byte memory granule and checks the tag on every access. A heap overflow that crosses a granule boundary with the wrong tag traps immediately, preventing the corruption from reaching adjacent objects. MTE is deployed on some Android devices but is not yet enabled on iOS. If deployed, MTE would make the CVE-2025-43300 overflow detectable at the hardware level regardless of the implementation language. In synchronous checking mode, a tag mismatch faults precisely on the violating instruction and yields defender-usable telemetry; asynchronous mode defers the trap to the next context switch for substantially lower overhead, trading temporal precision for deployability and making it the mode Android ships by default.

D. Copy Fail: A Concrete Kernel Walk-Through

Copy Fail (CVE-2026-31431) provides a concrete illustration of how the write-primitive pattern in Table I manifests at the Linux kernel level [34]. Its mechanism is instructive precisely because it has no structural connection to iOS or to image parsing—yet its exploitation outcome and the reason current mitigations did not stop it are identical to those of CVE-2025-43300.

a) The vulnerable optimization.: Linux exposes its in-kernel cryptographic algorithms to unprivileged userspace via `AF_ALG` sockets. A 2017 kernel commit to `algif_aead.c` introduced an in-place AEAD (Authenticated Encryption with Associated Data) optimization: instead of maintaining separate input and output scatterlists, the kernel chains both into one combined scatterlist using `sg_chain()`. The optimization is correct so long as the source data resides in kernel-managed bounce buffers—but the `splice()` system call can insert raw page-cache page references directly into that scatterlist, bypassing the buffer abstraction entirely.

b) The write.: The `authencesn` AEAD template performs a byte-rearrangement step during IPsec ESP decryption: it writes four bytes of the Extended Sequence Number (`seqno_lo`, taken from attacker-controlled AAD bytes 4–7) to the offset `assoclen + cryptlen` in the destination scatterlist via `scatterwalk_map_and_copy()`. In the in-place configuration this offset lands in the chained page-cache pages, producing an attacker-controlled 4-byte write to any page-cache page the attacker can name via `splice()`. Listing 3 contrasts the vulnerable and fixed kernel paths.

Listing 3. Vulnerable (in-place) vs. fixed (out-of-place) path in `algif_aead.c`. The single-line change that keeps `req->src != req->dst` eliminates the OOBW.

```
/* Vulnerable: in-place -- src and dst share
   the same scatterlist, which may contain
   page-cache pages inserted via splice().
      */
req->src = req->dst;
scatterwalk_map_and_copy(       /* OOBW:   */
    &seqno_lo, req->dst,        /* writes  */
    assoclen + cryptlen, 4, 1); /* into PC */

/* Fixed: out-of-place -- TX and RX SGL
   are kept separate; page-cache pages
   never enter the writable dst path.     */
req->src = tsgl_src; /* TX: user-supplied  */
req->dst = rsgl_dst; /* RX: kernel buffer  */
```

c) Exploitation.: The exploit requires no privileges [34]:

1) Open an `AF_ALG` socket bound to `authencesn(hmac(sha256),cbc(aes))`.
2) Encode the desired 4-byte payload in AAD bytes 4–7 of a `sendmsg()` call.

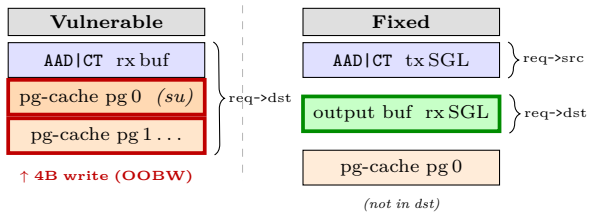


Fig. 6. Scatterlist layout in `algif_aead.c` before and after the CVE-2026-31431 patch. *Left (vulnerable):* `req->src = req->dst`; page-cache pages arrive via `splice()` in the combined SGL, so the `authencesn` 4-byte write crosses into page-cache memory (OOBW). *Right (fixed):* TX and RX SGLs are kept separate; the write is confined to the kernel output buffer.

3) Use `splice()` to feed the target file's page-cache pages (e.g., the `/usr/bin/su` setuid binary) as the ciphertext/tag input.
4) Call `recv()`: the kernel decrypts in-place and `authencesn` writes the 4 controlled bytes into the in-memory copy of the target binary.
5) Execute the corrupted binary; `su` is setuid root, so the injected stub runs with root privileges.

A critical stealth property: the corrupted page is never marked dirty, so the on-disk binary is unchanged and disk-based integrity mechanisms (IMA, dm-verity) do not trigger. The exploit is portable—a single 732-byte Python script achieves deterministic root on Ubuntu, Amazon Linux, RHEL, and SUSE—and requires no race condition or timing window [34].

d) Structural parallel.: SMAP, SMEP, and KASLR were all active on the test systems and none of them stopped Copy Fail, for exactly the same reason that ASLR, stack canaries, and PAC did not stop CVE-2025-43300: the write itself is never intercepted. The attacker does not need to redirect control flow—the kernel's own AEAD routine performs the write on the attacker's behalf. This is the same "legitimate-code-does-the-write" pattern as the ImageIO DNG parser: in both cases the memory safety violation is internal to a trusted subsystem, invisible to any mitigation that only monitors control-flow integrity or address-layout randomness.

E. Limitations

Three limitations must be acknowledged. First, our analysis of CVE-2025-43300 is second-hand: we rely on Quarkslab's reconstruction from patch diffing [17], not on independent reverse engineering. Certain exploitation details (heap-spraying strategy, PAC bypass technique) are inferred, not observed. Second, the Phoenix `heap-two` exercise deliberately omits real-world complications (ASLR, PAC, heap randomization, concurrent allocation); it illustrates the core mechanic, not the full exploitation engineering. Third, Section II-F extends comparison to nine chains spanning 2016–2026; further generalization of the "same primitive, different subsystem" thesis remains limited to publicly disclosed cases, and our analysis of several entries (BLASTPASS, Operation Triangulation, Copy Fail, DirtyFrag) relies on vendor advisories and third-party writeups rather than independent reverse engineering.

VI. Conclusion

This paper analyzed the 2025 WhatsApp–ImageIO zero-click exploit chain (CVE-2025-55177 + CVE-2025-43300), which compromised the mobile devices of fewer than 200 targeted individuals without any user interaction. Through a three-axis framework (root cause, patch, chain position), we reconstructed how a WhatsApp linked-device authorization bypass (CWE-863) delivered a crafted DNG image to Apple's ImageIO framework, where a two-byte inconsistency between TIFF metadata and an embedded JPEG marker triggered a heap out-of-bounds write (CWE-787) leading to remote code execution.

Three findings emerge. First, the fundamental exploitation primitive (a heap buffer overflowed into an adjacent function pointer) is identical to the one demonstrated in the educational Phoenix `heap-two` exercise. The difference between a textbook CTF exercise and a production zero-click chain is engineering (heap grooming, ASLR bypass, PAC bypass), not mechanism. Second, the 2025 chain reproduces the architectural pattern of FORCEDENTRY (2021) with different components (a different messenger, a different parser, a different image format) but the same structural vulnerability. The recurrence across four years and two independent ecosystems suggests that the pattern is inherent to the current architecture of media processing on mobile platforms, not an isolated accident. Third, the deployed mitigation stack (ASLR, stack canaries, sandboxing, PAC) addresses symptoms rather than the root cause: none of these defenses prevent the out-of-bounds write itself, and analogous kernel-level mitigations (SMAP, SMEP, KASLR) did not prevent Copy Fail [15] or DirtyFrag [16], two further write-primitive root exploits disclosed in April and May 2026—confirming that the finding is not iOS-specific but a property of the current mitigation-oriented paradigm.

We recommend two structural interventions. The near-term priority is incremental rewriting of high-exposure C/C++ parsers (ImageIO, CoreGraphics, and their Android equivalents) in memory-safe languages such as Rust or Swift. The longer-term intervention is deployment of hardware-enforced memory tagging (ARM MTE) on iOS, which would make heap overflows detectable at the granule level regardless of the implementation language.

References

[1] WhatsApp / Meta, "WhatsApp Security Advisory: CVE-2025-55177," Official advisory, August 2025. [Online]. Available: https://www.whatsapp.com/security/advisories
[2] Apple Inc., "About the security content of iOS 18.6.2 and iPadOS 18.6.2," Official security advisory, August 2025. [Online]. Available: https://support.apple.com/en-us/124925

[3] Lookout Threat Intelligence, "A vulnerability in WhatsApp for iOS and Mac: CVE-2025-55177," Threat intelligence report, August 2025. [Online]. Available: https://www.lookout.com/threat-intelligence/article/cve-2025-55177-whatsapp-for-ios

[4] CISA (Cybersecurity and Infrastructure Security Agency), "Known Exploited Vulnerabilities Catalog," Online catalog, 2025. [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

[5] A. Roy, "Technical Research Report: Advanced iPhone Zero-Click Exploits and Spyware (2019–2025)," Research report, June 2025. [Online]. Available: https://www.researchgate.net/publication/392685431_Technical_Research_Report_Advanced_iPhone_Zero-Click_Exploits_and_Spyware_2019_-_2025

[6] A. One, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, no. 49, November 1996, issue 49, Article 14. [Online]. Available: http://phrack.org/issues/49/14.html

[7] MITRE Corporation, "2024 CWE Top 25 Most Dangerous Software Weaknesses," Online database, 2024. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_top25_list.html

[8] ——, "CWE-787: Out-of-bounds Write," Online database, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/787.html

[9] ——, "2025 CWE Top 25 Most Dangerous Software Weaknesses," Online database, 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

[10] G. Thomas, "A Proactive Approach to More Secure Code," Microsoft Security Response Center blog, July 2019. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[11] Google Chrome Security Team, "Memory Safety – Chromium," Chromium project documentation, 2024. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[12] CISA, "The Urgent Need for Memory Safety in Software Products," Policy document, December 2023. [Online]. Available: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

[13] Citizen Lab, "FORCEDENTRY: NSO Group iMessage Zero-Click Exploit Captured in the Wild," Research report, September 2021. [Online]. Available: https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/

[14] Google Project Zero, "A Deep Dive into an NSO Zero-Click iMessage Exploit: Remote Code Execution," Research blog, December 2021. [Online]. Available: https://projectzero.google/2021/12/a-deep-dive-into-nso-zero-click.html

[15] Microsoft Threat Intelligence, "CVE-2026-31431: Copy Fail Vulnerability Enables Linux Root Privilege Escalation," Microsoft Security Blog, May 2026. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2026/05/01/cve-2026-31431-copy-fail-vulnerability-enables-linux-root-privilege-escalation/

[16] Wiz Research, "DirtyFrag: Linux Kernel Local Privilege Escalation via ESP and RxRPC (CVE-2026-43284)," Wiz security blog, May 2026. [Online]. Available: https://www.wiz.io/blog/dirty-frag-linux-kernel-local-privilege-escalation-via-esp-and-rxrpc

[17] Quarkslab, "Reverse Engineering of Apple's iOS 0-click CVE-2025-43300: 2 Bytes That Make Size Matter," Research blog, September 2025. [Online]. Available: https://blog.quarkslab.com/patch-analysis-of-Apple-iOS-CVE-2025-43300.html

[18] Anonymous, "Once upon a free()," *Phrack Magazine*, no. 57, August 2001, issue 57, Article 9. [Online]. Available: https://phrack.org/issues/57/9.html

[19] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 48–62.

[20] MITRE Corporation, "CWE-863: Incorrect Authorization," Online database, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/863.html

[21] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2012, pp. 86–106. [Online]. Available: https://vvdveen.com/publications/RAID2012.pdf

[22] Red Hat Customer Portal, "Kernel Local Privilege Escalation "Dirty COW" – CVE-2016-5195," Security advisory, October 2016. [Online]. Available: https://access.redhat.com/security/vulnerabilities/DirtyCow

[23] M. Kellermann, "The Dirty Pipe Vulnerability (CVE-2022-0847)," Technical writeup, CM4All, March 2022. [Online]. Available: https://dirtypipe.cm4all.com/

[24] R. Li, "StackRot (CVE-2023-3269): Linux Kernel Privilege Escalation Vulnerability," oss-security mailing list, July 2023. [Online]. Available: https://www.openwall.com/lists/oss-security/2023/07/05/1

[25] Cloudflare, "Copy Fail Linux Vulnerability Mitigation," Cloudflare blog, May 2026. [Online]. Available: https://blog.cloudflare.com/copy-fail-linux-vulnerability-mitigation/

[26] Red Hat Customer Portal, "DirtyFrag Linux Kernel Local Privilege Escalation – CVE-2026-43284," Red Hat Security Bulletin RHSB-2026-003, May 2026. [Online]. Available: https://access.redhat.com/security/vulnerabilities/RHSB-2026-003

[27] Exploit Education, "Exploit Education Phoenix VM," Free educational platform, 2024. [Online]. Available: https://exploit.education/phoenix/

[28] Engineering at Meta, "How WhatsApp Enables Multi-Device Capability," Engineering blog, July 2021. [Online]. Available: https://engineering.fb.com/2021/07/14/security/whatsapp-multi-device/

[29] Qualcomm Technologies, Inc., "Pointer Authentication on ARMv8.3," Whitepaper, 2017. [Online]. Available: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

[30] Phoronix, "The First Rust-Written Network PHY Driver Set To Land In Linux 6.8," Online news, March 2024. [Online]. Available: https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver

[31] LWN.net, "The (successful) end of the kernel Rust experiment," Online news, December 2025. [Online]. Available: https://lwn.net/Articles/1049831/

[32] ——, "Oxidizing Ubuntu: Adopting Rust Utilities by Default," Online news, 2025. [Online]. Available: https://lwn.net/Articles/1014002/

[33] Canonical, "What's new in security for Ubuntu 26.04 LTS?" Engineering blog, April 2026. [Online]. Available: https://canonical.com/blog/ubuntu-26-04-lts-security-updates

[34] Xint Security Research, "Copy Fail (CVE-2026-31431): Technical Walkthrough for Linux Distributions," Security research blog, May 2026. [Online]. Available: https://xint.io/blog/copy-fail-linux-distributions