I. 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) [**?**], [**?**], [**?**]. The chain is the central object of study in this paper. Figure **??** overviews the end-to-end flow, which we dissect in three stages: delivery (Section **??**), exploitation (Section **??**), and the mapping from the textbook `heap-two` primitive to the real CVE (Section **??**).

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 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 [**?**].

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 [**?**], 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) [**?**]; 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 [**?**]. 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" [**?**].

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

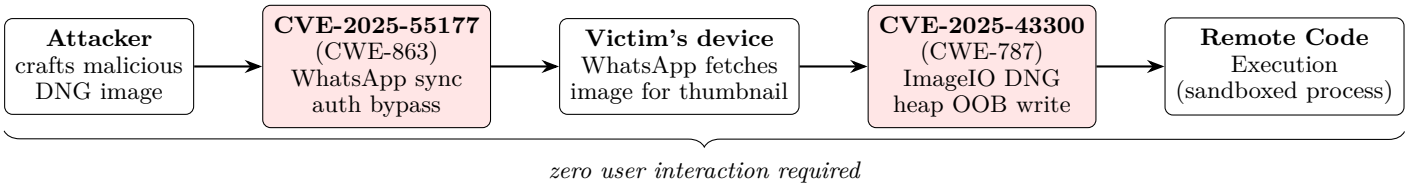

Fig. 1. End-to-end flow of the 2025 zero-click chain. Red nodes mark the two patched CVEs. Only the victim's phone number is required; no action by the victim is needed for the chain to reach code execution.

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 [**?**]). 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 [**?**].

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" [**?**]: 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 the object immediately after the victim buffer is one whose type contains a function pointer that the decoder will later call. 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. For the purpose of this paper, the essential observation is that the *structural* primitive granted by CVE-2025-43300 is the same one granted by the textbook `heap-two` exercise: overflow a heap buffer into adjacent attacker-reachable state, then hijack control flow when the program dereferences the corrupted state.

[Figure placeholder: heap layout before/after, `heap-two` vs ImageIO]

Fig. 2. 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—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.

d) Patch.: Apple's fix, inferred from diffing iOS 18.6.1 against iOS 18.6.2 per Quarkslab's analysis [**?**], 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.

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 **??** aligns the two heap layouts. In `heap-two`, `malloc` is called twice; the first chunk is the user-controllable buffer and the second

[Figure placeholder: `gdb` session on `heap-two`]

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).

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 immediately after the overflowable 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 approximately 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 **??** examines why this mitigation model has structural limits.