I. 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 (**??**), the two weakness classes that appear in the chain (**??**), the iOS media-processing pipeline in which both vulnerabilities fire (**??**), the mitigations that are supposed to prevent exploitation (**??**), and the 2021 FORCEDENTRY incident which is the historical precedent for the 2025 attack (**??**).

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" [**?**] 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 [**?**]. The heap-specific variant of this idea was formalised in 2001 by the Phrack article "Once upon a free()" [**?**], 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 **??** shows the minimal C pattern that produces this class of bug.

```
char *buf = malloc(64);  /* allocate 64-byte
    heap chunk */

/* 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.     */
```

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).

B. Vulnerability Taxonomy

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

CWE-787 — Out-of-bounds write [**?**] 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 accounts for the majority of critical remote-code-execution bugs patched in commodity operating systems [**?**]. CVE-2025-43300, the Apple ImageIO vulnerability at the core of the chain studied here, is a CWE-787 instance.

CWE-863 — Incorrect authorization [**?**] 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 [**?**], [**?**]. 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 [**?**]. 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 [**?**], [**?**]. 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.