I. 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 [**?**] 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 **??**) 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 [**?**]. 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 **??** and to FORCEDENTRY in Section **??**, 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:

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. 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
```

By default, the educational binary is compiled without modern protections. We verify the mitigation state using `checksec`:

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

Expected output: ASLR disabled, NX disabled, stack canary disabled, PIE disabled. This pedagogically useful configuration makes addresses deterministic and overflow mechanics 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 **??** shows the essential pattern.

Listing 1. Illustrative heap-two structure. Two `malloc` calls produce adjacent chunks. `strcpy` performs no bounds check; 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;   /* legitimate default */

/* Unchecked: if strlen(input) > DATA_SZ, bytes
   overflow from A->data into B->fp
                    */
strcpy(A->data, user_input);

B->fp();  /* attacker controls this call target */
```

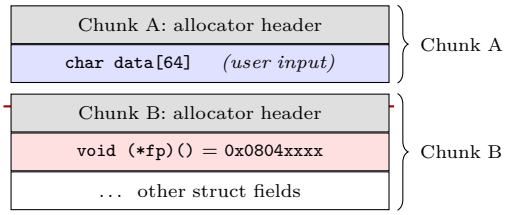

Fig. 1. Heap layout *before* overflow. Chunk A (64-byte user buffer) is adjacent to Chunk B (struct with function pointer). The dashed line marks the chunk boundary; a conforming write stays above it.

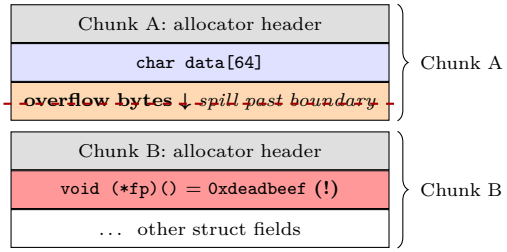

Fig. 2. Heap layout *after* overflow. Attacker input exceeds 64 bytes; excess bytes spill into Chunk B, overwriting the function pointer with an attacker-chosen value (`0xdeadbeef`). The red arrow shows the overflow path.

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`
4) Inspect heap state before overflow: `x/20x &chunk1`
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 **??** 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 **??** 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.

f) Connection to CVE-2025-43300.: The structural parallel is exact: both grant controllable heap write that overwrites an adjacent function pointer. Figures in Section **??** present (i) a side-by-side diagram aligning `heap-two`'s layout with the ImageIO layout reconstructed from Quarkslab's analysis [**?**]; and (ii) screenshots of a `gdb` session showing the overwritten function pointer at the moment of control-flow transfer.

TABLE I
TOOLS AND PRIMARY SOURCES USED IN THIS STUDY.

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

D. Tools and Primary Sources

All materials used in this study are publicly available; a summary is given in Table **??**. 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.