#! Anatomy of a Bug

Autopsy of the Triad

"Liquid Glass" Zero-Days

The Patient

- **Name:** iPhone 12+
- **CVE:** CVE-2025-14174, CVE-2025-43529, CVE-2025-46285
- **Diagnosis:** Full Chain Compromise
- **Vector:** WebKit & Kernel
- **Severity:** 🔥 10.0

What does 10.0 mean?

- **Auth:** None
- **User Interaction:** Little or None
- **Attack Vector:** Remote
- **Result:** Permanent Root access

#1 The Context

The "Liquid Glass" Era

#1.1 The Context

The Attack Surface Expansion

👉 iOS 26 introduced the "Liquid Glass" design language, heavily relying on advanced GPU acceleration and complex compositing.

👉 This visual overhaul required substantial rewrites in Core Animation and WebKit.

👉 These new graphical frameworks expanded the attack surface significantly.

#1.2 The Context

The Threat Landscape

👉 Late 2025 saw a "Zero-Day Arms Race", with Apple patching over a dozen zero-days.

👉 This campaign falls into the "Targeted Surveillance" category, distinct from mass-market exploits.

👉 The exploit was hoarded by Private Sector Offensive Actors (PSOAs) for use against high-value targets.

#2 The Architecture

A14 Bionic Defenses

#2.1 The Architecture

Pointer Authentication (PAC)

👉 The A14 Bionic enforces ARMv8.3-A Pointer Authentication.

👉 It cryptographically signs pointers using a secret key inaccessible to user mode.

👉 Impact: Standard ROP and JOP are neutralized.

#2.2 The Architecture

Page Protection Layer (PPL)

👉 PPL sandboxes the Kernel itself. So even with Kernel Read/Write, the attacker cannot modify Page Tables.

👉 Only the PPL (hypervisor-like layer) can modify page tables.

👉 Impact: Even with Kernel Read/Write, attackers cannot simply remap kernel memory as executable.

#3 The Vulnerability

CVE-2025-14174 (ANGLE)

#3.1 The Vulnerability (1/3)

What is ANGLE?

👉 ANGLE is a translator. It takes Web Graphics (OpenGL ES) and translates them to Native Graphics (Metal).

👉 It acts as the "Interpreter" between the website's code and your iPhone's GPU.

👉 Because it handles complex data from the web, it is a prime target.

#3.2 The Vulnerability (1/3)

The Two Variables

👉 To upload a texture, the code needs to answer two questions:

- 👉 "How big is the texture?"
- 👉 "How should I unpack it?"

👉 The Critical Dependency: The engine uses the Unpack Settings to calculate the Memory Allocation. It assumes this configuration perfectly matches the actual dimensions of the Raw Image Data.

#3.3 The Vulnerability (1/3)

The Logic Trap

👉 The developer made a fatal assumption: "The User's Unpack Setting matches the Actual Texture Size."

👉 The Formula: The code calculated pixelsDepthPitch (the byte size of one 3D depth layer) using the User-Supplied Height (GL_UNPACK_IMAGE_HEIGHT).

👉 The Mismatch: The exploit sets this user height to 1. The engine calculates: Buffer_Size = Row_Pitch * 1. This results in a Critical Under-Allocation.

#3.4 The Vulnerability (1/3)

The Copy (The Crash)

👉 The Copy Logic ignores the user setting. It looks at the Actual Texture Data.

👉 It attempts to copy a Full-Sized Texture (e.g., Height = 1000).

👉 1000 lines of data are forced into a buffer built for 1 line.

👉 Result: It overwrites everything after the buffer (Heap Overflow).

#4 The Vulnerability

CVE-2025-43529 (WebKit)

#4.1 The Vulnerability (2/3)

What is a LayerPool?

👉 In iOS rendering, creating objects is slow.

👉 LayerPool is a "recycle bin". It holds old CALayer objects so they can be reused instead of destroyed.

👉 To work correctly, the Pool must live as long as the rendering process needs it.

#4.2 The Vulnerability (2/3)

Stack vs. Heap (The Basics)

👉 **Stack Memory: Temporary and Scope-Bound.** It is automatically created when a function frame initializes and is destroyed immediately when the function returns.

👉 **Heap Memory: Persistent.** It exists independent of function calls and lives until explicitly freed.

👉 **The Bug:** The developer placed the LayerPool on the Stack (or inside a temporary object), causing it to be freed prematurely while background threads still tried to use it.

#4.3 The Vulnerability (2/3)

The Use-After-Free Setup

👉 Step 1: Function Render() initializes. A LayerPool object is allocated on the temporary Stack (local scope).

👉 Step 2: A Reference to this pool is captured by an Asynchronous Closure or passed to a Background Thread for compositing.

👉 Step 3: The function Render() returns and exits. The Stack Frame is destroyed, effectively freeing the memory while the thread continues.

#4.4 The Vulnerability (2/3)

The Fatal Disconnect

👉 Because Render() finished, the Stack Frame is destroyed (popped). The memory address that held the LayerPool is invalidated and marked "Free" for reuse by other functions.

👉 Step 4: The Background Thread (or asynchronous closure) wakes up to perform its task. Crucially, it still holds the dangling pointer to the now-destroyed stack memory.

👉 It blindly dereferences the pointer to allocate a layer. Accessing this attacker-controlled memory triggers the Use-After-Free (UAF) vulnerability.

#4.5 The Vulnerability (2/3)

The Takeover

👉 The exploit targets the predictable timing of the stack frame destruction.

👉 Between Step 3 and Step 4, a Heap Spray is executed.

👉 This operation captures the "Free" memory slot, filling it with controlled Fake Data.

👉 The background thread subsequently operates on this substituted payload, treating it as the valid LayerPool.

#5 The Vulnerability

CVE-2025-46285 (Kernel)

#5.1 The Vulnerability (3/3)

The Integer Trap

👉 **Hardware Constraints:** Computers store numbers in fixed-size containers. A standard uint32_t (Unsigned 32-bit Integer) has a hard mathematical ceiling of $2^{32}-1$.

👉 **The Odometer Effect:** If you add 1 to this maximum value, the CPU silently wraps around to 0.

👉 **The Flaw:** The XNU Kernel dangerously relied on these 32-bit types to track Timestamps. Since time always increases, or when adding a duration, the value exceeds the container's size.

#5.2 The Vulnerability (3/3)

The Equation

👉 The Kernel frequently performs arithmetic on timestamps to manage scheduling deadlines, IPC timeouts, and resource accounting.

👉 The Equation: Deadline = Current Time (System Uptime) + User Duration (Input).

👉 The Vulnerability: The attacker fully controls the User Duration parameter passed via a syscall or kernel driver interaction, allowing them to manipulate the result.

#5.3 The Vulnerability (3/3)

The Overflow

👉 The attacker explicitly provides a massive integer (near the maximum 32-bit value) for the duration parameter to force the calculation out of bounds.

👉 Current Time + Massive Number exceeds the Max 32-bit Limit. The 32-bit variable cannot store the resulting sum.

👉 Consequently, the value wraps around. The intended future deadline becomes a tiny number, confusing the scheduler.

#5.4 The Vulnerability (3/3)

The Logic Break

👉 The Kernel uses the tiny overflowed value to allocate a buffer.

👉 It writes Full Data into this tiny buffer, causing a Kernel Heap Overflow.

👉 This overflow corrupts the adjacent memory, specifically the Process Credentials (struct proc).

👉 The attacker manually changes their User ID (UID) to 0 (Root), becoming the System Admin.

#6 The Kill-Chain

Step-by-Step Execution

#6.1 The Kill-Chain

Step 1: The Lure & Setup

👉 Step 1: A malicious website is hosted containing specially crafted WebGL content, potentially delivered via a "1-click" link in iMessage or WhatsApp.

👉 Step 2: WebKit passes the drawing commands to ANGLE. ANGLE translates these into Apple's native Metal instructions.

👉 Step 3: The exploit sets the parameter GL_UNPACK_IMAGE_HEIGHT to 1. This tricks the allocator into calculating a tiny buffer size, assuming the texture consists of only one row.

#6.2 The Kill-Chain

Step 2: The Memory Corruption

👉 The texture upload logic iterates over the Real Dimensions of the image, causing it to write data past the end of the allocated buffer inside the WebContent process.

👉 This results in a Out-of-Bounds Write (Heap Overflow) that overwrites adjacent memory, corrupting critical internal WebKit structures.

👉 This corruption is leveraged to establish a stable "Write Primitive", giving the ability to manipulate the Heap layout and prepare for the next stage.

#6.3 The Kill-Chain

Step 3: Seizing Control (RCE)

👉 Step 3: The LayerPool UAF is triggered and the Heap sprayed to fill the slot with controlled data.

👉 The PAC Bypass (Data-Only Attack): Since cryptographic signatures (PAC) cannot be forged, the code pointers are not overwriten. Instead, non-protected data fields are corrupted, such as the Length or Capacity of an Array.

👉 The Consequence: The browser executes the code, granting Arbitrary Read/Write access to the entire memory space.

#6.4 The Kill-Chain

Step 4: The Kernel Pivot

👉 To escape the sandbox, the compromised process issues a syscall to the Kernel, likely via IOKit.

👉 It passes a crafted "User Duration" parameter to a time-sensitive function.

👉 The Trap: This duration is set to a massive integer (e.g., 0xFFFFFFF0), just below the maximum 32-bit value.

#6.5 The Kill-Chain

Step 5: Privilege Escalation

👉 The 32-bit integer overflows, resulting in a corrupted state.

👉 The corruption allows the Kernel memory to be overwritten.

👉 By targeting the process credentials (struct proc), privileges are escalated.

👉 Result: Full System Compromise.

#7 The Fix.

Detailed Remediation

#7.1 The Fix (ANGLE)

Commit 95a32cb37: Decoupled buffer sizing from user input.

```
// ANGLE Remediation Logic
// OLD: Used pixelsDepthPitch (derived from user input)
// NEW: Calculates size based on ACTUAL texture dimensions

size_t requiredSize = actualWidth * actualHeight * bytesPerPixel;
if (allocatedSize < requiredSize) {
    // 🛑 Block the upload or reallocate
    ReallocateBuffer(requiredSize);
}
```

#7.2 The Fix (WebKit)

Bug 302502: Enforced Heap Allocation.

```
// WebKit Remediation Logic
// Mandates LayerPool creation on the heap via smart pointers.

// Vulnerable:
// LayerPool pool; // Stack allocated

// Fixed:
RefPtr<LayerPool> pool = LayerPool::create(); // Heap allocated
// The memory remains valid as long as 'pool' is referenced.
```

#7.3 The Fix (Kernel)

XNU Update: 64-bit Adoption.

```
// Kernel Remediation Logic
// Transition from 32-bit to 64-bit for time values.

// Vulnerable:
// uint32_t deadline = current_time + user_duration; // Overflows

// Fixed:
uint64_t deadline = current_time + user_duration; // Safe
```

#8 Developer's Takeaway

Handle the Edge-Cases

#8.1 Developer's Takeaway

Handle the Edge-Cases

👉 **Shared Code is a Shared Threat:** Vulnerabilities in third-party libraries can compromise your entire platform.

👉 **Manage Your Lifetimes:** Never rely on stack allocation for objects that might be referenced asynchronously. Use Smart Pointers.

👉 **Modernize Your Integers:** In 2025, using 32-bit integers for time or size is a liability. Default to 64-bit to prevent overflows.

Status

Discovered by Google TAG & Apple SEAR.
Patched iOS 26.2 (Dec 12, 2025).
Update Immediately.

#! Anatomy of a Bug

Technical Credits:

Google TAG & Apple SEAR

Author: @tralsesec

#AOAB #AnatomyOfABug #ExploitDev #Infosec #Kernel #MobileSecurity #WebKit #ZeroDay #iOS26