What’s in a (Nick)name

Examining a previously unknown
iMessage vulnerability with possible
exploitation in the US and EU

iVerify.

What's in a (Nick)name

iVerify's Discovery Process

An Overview of the NICKNAME Vulnerability

Technical Analysis

SECTION 1

What's in a (Nick)name

Overview

iVerify discovered a previously unknown iOS vulnerability called "NICKNAME" affecting iMessage's handling of contact profile updates. This vulnerability was observed in iOS versions up to 18.1.1 and fixed in iOS 18.3. The bug involves a race condition in how iOS processes "Nickname Updates," the feature that allows users to share personalized contact information with their iMessage contacts.

Our investigation found evidence suggesting this vulnerability may have been remotely exploited in the United States and European Union. We identified exceedingly rare crash logs that appeared exclusively on devices belonging to high-risk individuals including government officials, political campaign staff, journalists, and tech executives. At least one affected European Union government official received an Apple Threat Notification approximately thirty days after we observed this crash on their device, and forensic examination of at least one other device revealed signs of successful exploitation.

The Vulnerability Explained

The NICKNAME vulnerability exists in "imagent," a core process that handles iMessage traffic. When users update their contact profile (nickname, photo, or wallpaper), this triggers a "Nickname Update" that gets processed by the recipient's device.

The technical issue involves how the imagent process handles data associated with these updates. Before the fix, when imagent needed to broadcast a Nickname Update to different parts of the system, it used a mutable data container (NSMutableDictionary) that could be changed while being accessed by other processes. This created a classic race condition: one thread might be reading the Nickname Update details while another thread simultaneously modifies the same data container. This corruption can trigger a memory safety bug known as a Use-After-Free (UAF), causing the imagent process to crash.

This vulnerability is particularly concerning because it has zero-click potential, meaning it could potentially be triggered without any user interaction. Simply receiving specially crafted messages could potentially lead to exploitation. Since Nickname Updates can be processed even from unknown senders, an attacker would only need the target's phone number or Apple ID to attempt an exploit.

Evidence of Exploitation

Between April 2024 and January 2025, we analyzed crash data from nearly 50,000 devices and found that the imagent crashes related to Nickname Updates are exceedingly rare, comprising less than 0.001% of all crash logs collected. What makes these crashes suspicious isn't just their rarity, but their exclusive appearance on devices belonging to individuals likely to be targeted by sophisticated threat actors.

The affected devices belonged to individuals affiliated with political campaigns, media organizations, tech companies, and governments in the EU and US. Some individuals reported additional suspicious activity, including physical surveillance. Most notably, these crashes were observed on at least one device belonging to a senior European Union government official approximately thirty days before they received an Apple Threat Notification.

Forensic examination of at least one additional device provided evidence suggesting exploitation: several directories related to SMS attachments and message metadata were modified and then emptied just 20 seconds after the imagent crash occurred. This pattern of deleting potential evidence mirrors techniques observed in confirmed spyware attacks, where attackers "clean up" after themselves, although there are benign explanations for this specific behavior, too. Similar cleanup activity was noted again about 80 days later on the same device following an urgent submission indicating a crash to Apple.

Finally, the imagent process itself is a core iOS process that has been heavily fortified owing to its role in previous successful exploitation attempts.

While no smoking gun definitively proving exploitation exists, when taken together, this body of evidence gives us moderate confidence these crashes indicate targeted exploitation attempts. Further, it is possible that the NICKNAME vulnerability served as one component in a larger exploit chain, providing attackers with a memory corruption primitive that could be leveraged to compromise iOS devices.

A thorough description of our methodology and findings is provided in the technical analysis.

Recommendations

We recommend all users update to the latest version of iOS. Users in high-risk categories (government officials, journalists, activists, etc.) should be particularly vigilant about keeping their devices updated and consider enabling Apple's Lockdown Mode for additional protection against sophisticated attacks.

SECTION 2

iVerify's Discovery Process

Detection Capabilities and Methodology

iVerify's security platform includes a sophisticated telemetry analysis engine that collects and analyzes crash logs, diagnostic data, and raw kernel-level artifacts from iOS devices. This broad deployment across diverse organizations provides a statistically significant sample for identifying anomalous patterns that might indicate security threats.

Initial Discovery

In late 2024, iVerify's anomaly detection systems flagged a series of unusual "imagent" process crashes occurring on devices belonging to individuals affiliated with political campaigns, media organizations, technology companies, and government officials. These crashes exhibited patterns consistent with sophisticated zero-click attacks typically associated with commercial spyware operations.

Key indicators that prompted further investigation include:

- The extreme rarity of these specific crash patterns (<0.001% of all crash logs)
- Their exclusive appearance on devices belonging to high-value targets
- Distinctive crash signatures showing memory corruption in the "imagent" process
- Similarity to crash patterns seen in known spyware attacks

Investigative Process

Our process included the following components:

1. Telemetry Analysis: iVerify first performed statistical analysis on our telemetry data to confirm the uniqueness of these crash patterns and the correlation with specific iOS versions (17.2.1-18.1.1).

2. Crash Log Investigation: Security researchers symbolicated the crash logs to identify the specific methods and functions involved, revealing the connection to iMessage's Nickname Updates feature.

3. Vulnerability Identification: Technical teams reverse-engineered the relevant code paths using tools like IDA Pro and the ipsw framework, identifying a thread-safety issue in how mutable dictionaries were handled by the IMDNicknameController class.

4. Test Environment Validation: iVerify constructed a test environment with controlled iOS devices to replicate the data flow of Nickname Updates and understand the vulnerability in a controlled setting, revealing the difficulty associated with triggering this vulnerability under organic circumstances.

5. Patch Analysis: Researchers performed differential analysis between iOS versions to identify the fix implemented in iOS 18.3, confirming the nature of the vulnerability.

6. Forensic Examination: For at least one affected device, iVerify conducted in-depth forensic analysis using Sysdiagnose data and encrypted backups, revealing suspicious filesystem modifications occurring immediately after crashes—activity consistent with known spyware cleanup procedures.

Unique Detection Advantages

Several key factors enabled iVerify to identify this zero-click attack:

- The scale and depth of our telemetry collection across both high-risk users and the general population
- Deep technical expertise in iOS internals and experience with previous iOS exploitation techniques and their forensic markers
- Our ability to perform longitudinal analysis across iOS versions and device populations

This investigation exemplifies how telemetry-based anomaly detection, combined with deep technical analysis, can uncover sophisticated zero-click vulnerabilities that would otherwise remain invisible to conventional security monitoring.

SECTION 3

An Overview of the NICKNAME Vulnerability

The Nickname Update Feature

The vulnerability impacts iOS "Share Name and Photo" feature in iMessage, which allows users to personalize how they appear in their contacts' Messages app. Introduced as part of Apple's efforts to enhance iMessage personalization, this feature enables users to:

- Set a custom display name for themselves
- Share a profile photo or Memoji
- Include a custom background/wallpaper for their contact card
- Control sharing settings (with "Always Ask" or "Contacts Only" options)

When enabled, iOS presents a prompt asking if the user wants to share their name and photo when messaging someone new. If approved, the recipient receives a "Nickname Update" containing this personalization data.

From a user perspective, this creates a more personalized messaging experience. However, from a security perspective, it introduces a new complex data processing pipeline that accepts and handles content from external sources—a potential target for attackers.

The Vulnerability in Context

The "NICKNAME" vulnerability (patched in iOS 18.3) is significant because it affects iMessage—a system Apple has heavily fortified in recent years. Apple implemented BlastDoor in iOS 14 as a sandboxed service designed to process untrusted data before it reaches the core messaging system. Despite these protections, sophisticated attacks like FORCEDENTRY and BLASTPASS subsequently emerged, specifically engineered to bypass BlastDoor's security boundaries. Assuming our conclusions about attempted exploitation are correct, NICKNAME continues this trend, demonstrating that determined attackers continue to find narrow vectors through Apple's defenses.

Technical Understanding

At its core, NICKNAME is a thread-safety vulnerability in iOS's "imagent" process—the Instant Messaging Agent responsible for handling iMessage traffic. The vulnerability specifically impacts the processing of contact "Nickname Updates," a feature allowing users to share their name, avatar, and wallpaper with contacts.

The bug manifests as a Use-After-Free (UAF) condition in the IMDNicknameController class. When a device receives nickname updates, the following occurs:

1. The imagent process decrypts and unpacks the message (represented as an NSDictionary).
2. The message is sent to IMTransferAgent to retrieve nickname data from iCloud.
3. Data passes through BlastDoor for sanitization.
4. The resulting data is transformed into an IMNickname object.
5. Several methods process this object, ultimately calling _broadcastNicknamesMapChanged.

The vulnerability arises because IMDNicknameController uses mutable dictionaries (pendingNicknameUpdates, handledNicknames, and archivedNicknames) that can be accessed concurrently by multiple threads. When nickname updates are processed rapidly:

- Thread A begins serializing a dictionary for an XPC message
- Thread B modifies the same dictionary
- Thread A attempts to access now-corrupted memory, triggering a crash

The crash logs consistently show failures in objc_retain() while attempting to access invalid memory addresses—a telltale sign of memory corruption.

Exploit Potential

While a crash alone isn't an exploit, sophisticated attackers could potentially do the following:

1. Send specially crafted nickname updates in rapid succession
2. Trigger the race condition at a precise moment

3. Corrupt memory in a controlled manner
4. Use the corrupted memory as a primitive for code execution

What makes this vulnerability particularly concerning is that:

- It requires no user interaction (zero-click)
- It’s potential to bypass BlastDoor sandboxing
- A body of evidence suggests this vulnerability was exploited in the wild

Apple's patch in iOS 18.3 addressed this vulnerability by using immutable copies of dictionaries when broadcasting nickname updates, effectively preventing the race condition that enabled the exploitation.

This vulnerability highlights the ongoing arms race between Apple's security enhancements and determined attackers finding narrow windows of opportunity in complex messaging systems.

SECTION 4

Technical Analysis

iVerify.

Crash Log Review

A body of insightful research elaborates on the historic targeting of the iMessage remote attack surface. Building upon the knowledge of others, iVerify recognizes that this Apple service - among others, to include third-party encrypted messaging applications - as a priority target for commercial spyware vendors (CSVs) and nation-state adversaries with spyware capabilities. This general understanding of adversarial TTPs underpins our analysis. We start with a review of the `imagent` crash logs themselves, understanding that each one is similar in nature, differentiated only by iOS version and, in some other cases, a different "Exception Type."

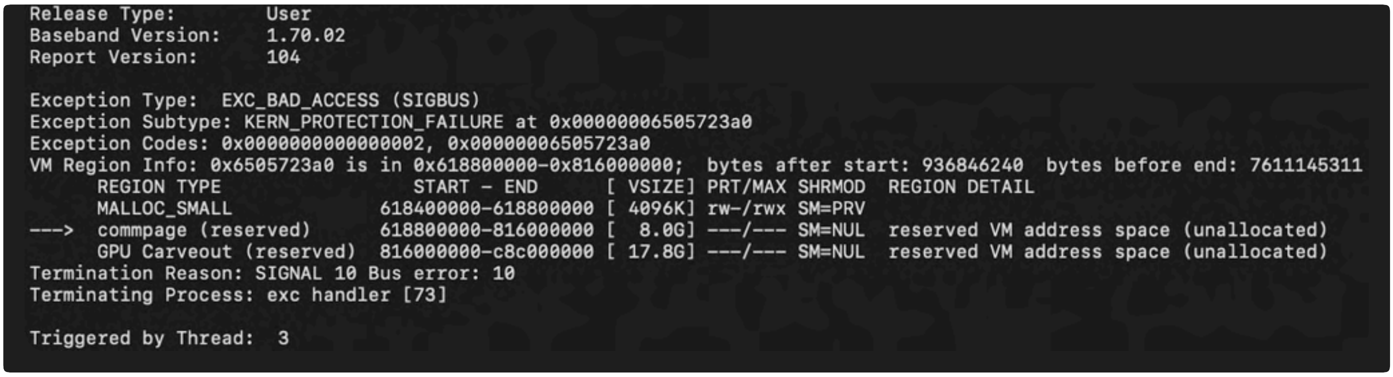
'Imagent' (or the Instant Messaging Agent) is a core process uniquely responsible for handling traffic from iMessage. Its remote attack surface is frequently targeted and, in the context of zero-click exploitation, enables malicious actors to compromise victims by using their phone number and/or Apple ID. Payloads exploiting components of the larger iMessage process flow, even with improved security mechanisms from other services such as BlastDoor, have the potential to crash the 'imagent' process. Documented analysis of FORCEDENTRY, BLASTPASS, and Operation Triangulation make for great case studies and background reading.

With this context in mind, the iVerify team focused our analysis on the 'imagent' crash logs which were present in our telemetry collection. iVerify was interested in this crash log because of its 'Exception Type' field which indicated the 'imagent' incorrectly attempted to access a misaligned or invalid address in memory, or failed pointer authentication. Closer inspection of the 'Exception Subtype' field highlighted KERN_PROTECTION_FAILURE at a specific address.

```
Process:               imagent [73]
Path:                  /System/Library/PrivateFrameworks/IMCore.framework/imagent.app/imagent
Identifier:            com.apple.imagent
Version:               10.0 (1000)
Code Type:             ARM-64 (Native)
Role:                  Unspecified
Parent Process:        launchd [1]
Coalition:             com.apple.imagent [85]

Date/Time:             2024-09-[redacted]
Launch Time:           2024-09-
OS Version:            iPhone OS 17.7 (21H16)
Release Type:          User
Baseband Version:      1.70.02
Report Version:        104

Exception Type:  EXC BAD ACCESS (SIGBUS)
```



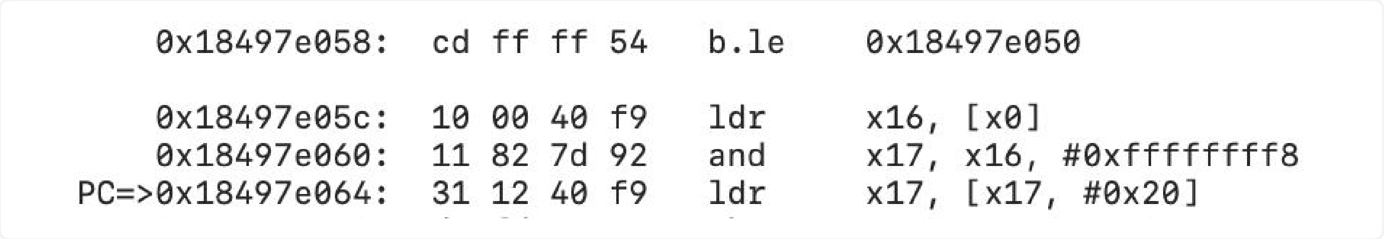
```
Release Type:         User
Baseband Version:     1.70.02
Report Version:       104

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Subtype: KERN_PROTECTION_FAILURE at 0x00000006505723a0
Exception Codes: 0x0000000000000002, 0x00000006505723a0
VM Region Info: 0x6505723a0 is in 0x618800000-0x816000000;  bytes after start: 936846240  bytes before end: 7611145311
      REGION TYPE                 START - END      [ VSIZE] PRT/MAX SHRMOD  REGION DETAIL
      MALLOC_SMALL             618400000-618800000 [ 4096K] rw-/rwx SM=PRV
--->  commpage (reserved)      618800000-816000000 [  8.0G] ---/--- SM=NUL  reserved VM address space (unallocated)
      GPU Carveout (reserved)  816000000-c8c000000 [ 17.8G] ---/--- SM=NUL  reserved VM address space (unallocated)
Termination Reason: SIGNAL 10 Bus error: 10
Terminating Process: exc handler [73]

Triggered by Thread:  3
```

Figure 1: imagent crash log header, device with iOS version 17.7

The backtrace for thread 3 terminated in the ‘objc_retain()’ stack frame, part of ‘libobjc.A.dylib’.



```
    0x18497e058:  cd ff ff 54   b.le    0x18497e050

    0x18497e05c:  10 00 40 f9   ldr     x16, [x0]
    0x18497e060:  11 82 7d 92   and     x17, x16, #0xfffffff8
PC=>0x18497e064:  31 12 40 f9   ldr     x17, [x17, #0x20]
```

Figure 2: instructions before the crash, device with iOS version 17.7 objc_retain

The thread crashes on the fourth instruction of the ‘objc_retain()’ function. ‘x0’ holds the pointer to the passed `NSDictionary` object to `objc_retain()`. The value at the address of the object is loaded into `x16` which should be the ISA pointer. The next instruction clears the last 3 bits and the upper bits of the pointer. The final instruction loads the value at offset `0x20` of the pointer into register `x17`. This is where our code crashes because the address found in `x17` at offset 20 is not valid.

We can validate this by looking at the value and the crash message:

~~~
```
  x16: 0xa31afb5650572383
  x17: 0x0000000650572380


Exception Type:      EXC_BAD_ACCESS (SIGBUS)
Exception Subtype:   KERN_PROTECTION_FAILURE at 0x00000006505723a0
```
~~~

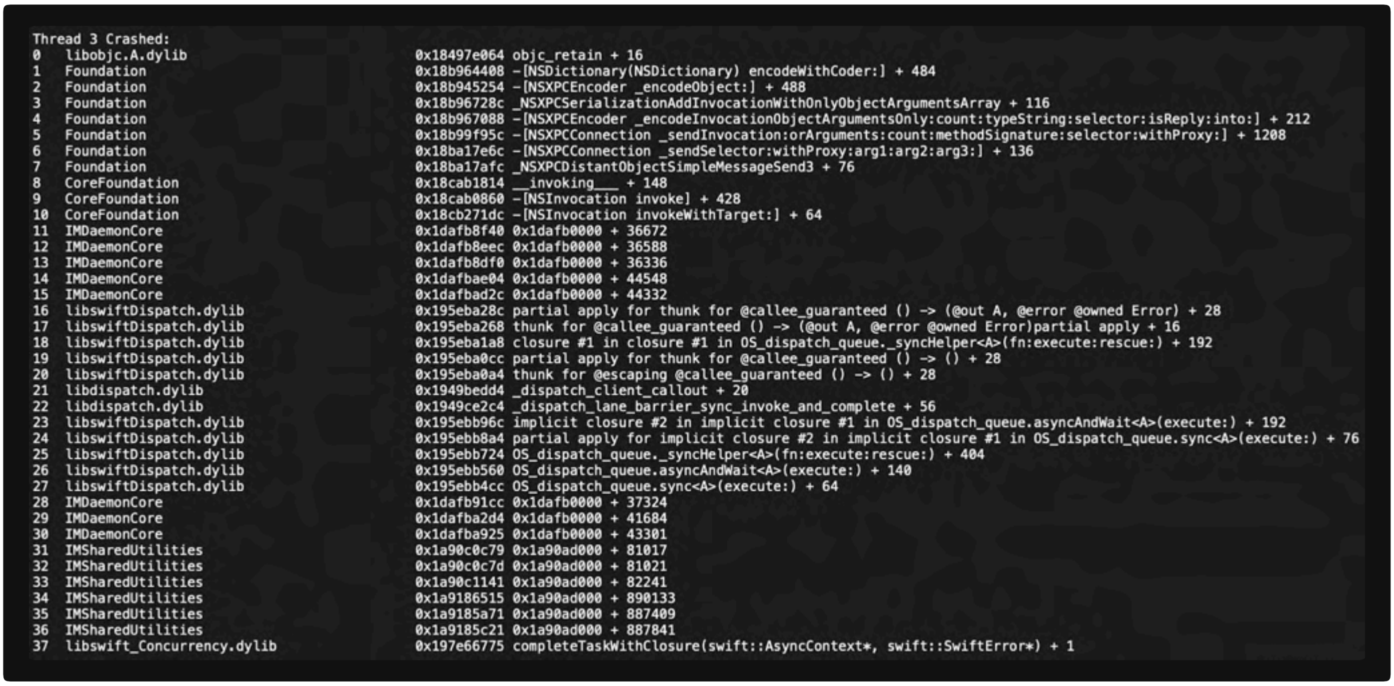
We can see that `x17` is the result of `x16 & 0xFFFFFFFF8`.

Also, `0x0000000650572380 + 0x20 = 0x00000006505723a0`; this value matches the memory address the operating system attempted to resolve when it crashed. When the OS attempted to access the memory at this address, it was unmapped, leading to the crash.

Prior to termination, `IMDaemonCore` likely prepared an object and leveraged `NSInvocation` to pass this object over XPC. This backtrace reveals that the object was an `NSDictionary`, serialized, and ultimately encoded prior to the crash.

This supports our working theory, that the crash was triggered in this particular frame because the pointer - which should resolve to a part of the `NSDictionary` object in memory with a valid reference count as part of the Objective-C runtime - was corrupted or had already been deallocated after a precursory call to `objc_release()`, indicative of a Use-After-Free (UAF) bug.

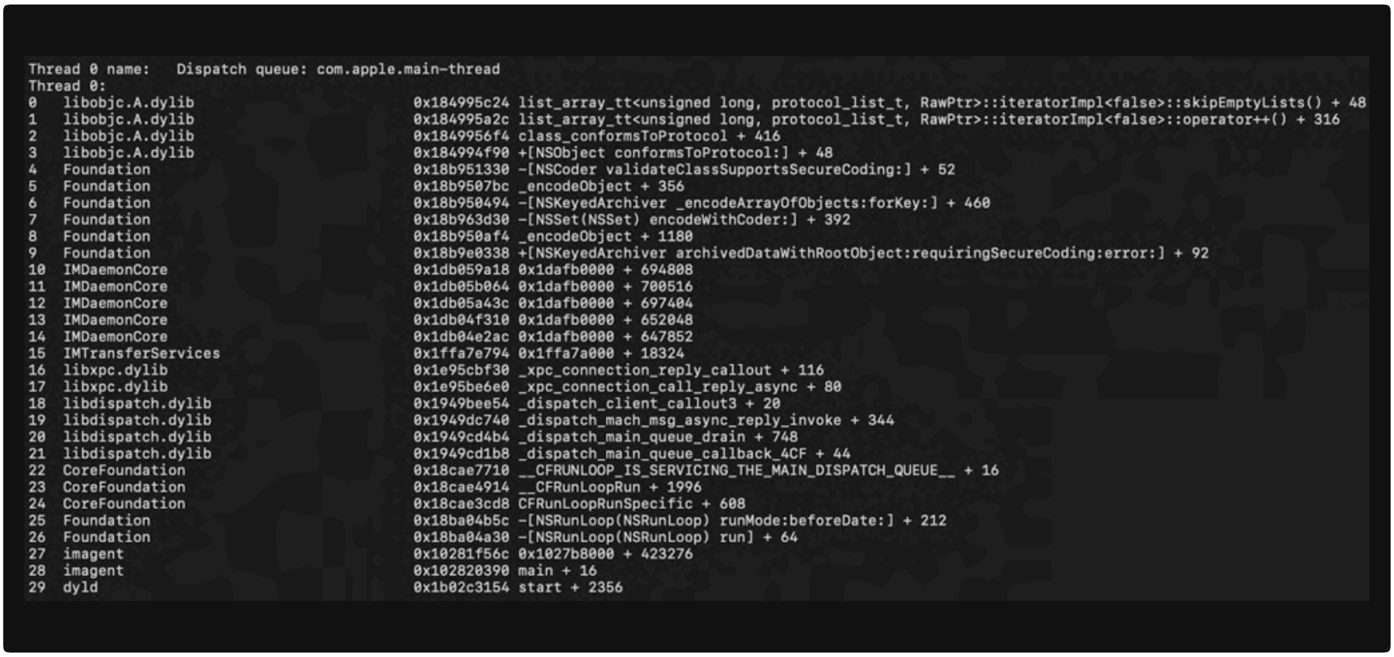
But the questions that remain here are what sort of `NSDictionary` was sent by `imagent`, which service should receive that `NSDictionary`, and how was it corrupted in the first place?



```
Thread 3 Crashed:
0   libobjc.A.dylib                         0x18497e064 objc_retain + 16
1   Foundation                              0x18b964408 -[NSDictionary(NSDictionary) encodeWithCoder:] + 484
2   Foundation                              0x18b945254 -[NSXPCEncoder _encodeObject:] + 488
3   Foundation                              0x18b96728c _NSXPCSerializationAddInvocationWithOnlyObjectArgumentsArray + 116
4   Foundation                              0x18b967088 -[NSXPCEncoder _encodeInvocationObjectArgumentsOnly:count:typeString:selector:isReply:into:] + 212
5   Foundation                              0x18b99f95c -[NSXPCConnection _sendInvocation:orArguments:count:methodSignature:selector:withProxy:] + 1208
6   Foundation                              0x18ba17e6c -[NSXPCConnection _sendSelector:withProxy:arg1:arg2:arg3:] + 136
7   Foundation                              0x18ba17afc _NSXPCDistantObjectSimpleMessageSend3 + 76
8   CoreFoundation                          0x18cab1814 ___invoking___ + 148
9   CoreFoundation                          0x18cab0860 -[NSInvocation invoke] + 428
10  CoreFoundation                          0x18cb271dc -[NSInvocation invokeWithTarget:] + 64
11  IMDaemonCore                            0x1dafb8f40 0x1dafb0000 + 36672
12  IMDaemonCore                            0x1dafb8eec 0x1dafb0000 + 36588
13  IMDaemonCore                            0x1dafb8df0 0x1dafb0000 + 36336
14  IMDaemonCore                            0x1dafbae04 0x1dafb0000 + 44548
15  IMDaemonCore                            0x1dafbad2c 0x1dafb0000 + 44332
16  libswiftDispatch.dylib                  0x195eba28c partial apply for thunk for @callee_guaranteed () -> (@out A, @error @owned Error) + 28
17  libswiftDispatch.dylib                  0x195eba268 thunk for @callee_guaranteed () -> (@out A, @error @owned Error)partial apply + 16
18  libswiftDispatch.dylib                  0x195eba1a8 closure #1 in closure #1 in OS_dispatch_queue._syncHelper<A>(fn:execute:rescue:) + 192
19  libswiftDispatch.dylib                  0x195eba0cc partial apply for thunk for @callee_guaranteed () -> () + 28
20  libswiftDispatch.dylib                  0x195eba0a4 thunk for @escaping @callee_guaranteed () -> () + 28
21  libdispatch.dylib                       0x1949bedd4 _dispatch_client_callout + 20
22  libdispatch.dylib                       0x1949ce2c4 _dispatch_lane_barrier_sync_invoke_and_complete + 56
23  libswiftDispatch.dylib                  0x195ebb96c implicit closure #2 in implicit closure #1 in OS_dispatch_queue.asyncAndWait<A>(execute:) + 192
24  libswiftDispatch.dylib                  0x195ebb8a4 partial apply for implicit closure #2 in implicit closure #1 in OS_dispatch_queue.sync<A>(execute:) + 76
25  libswiftDispatch.dylib                  0x195ebb724 OS_dispatch_queue._syncHelper<A>(fn:execute:rescue:) + 404
26  libswiftDispatch.dylib                  0x195ebb560 OS_dispatch_queue.asyncAndWait<A>(execute:) + 140
27  libswiftDispatch.dylib                  0x195ebb4cc OS_dispatch_queue.sync<A>(execute:) + 64
28  IMDaemonCore                            0x1dafb91cc 0x1dafb0000 + 37324
29  IMDaemonCore                            0x1dafba2d4 0x1dafb0000 + 41684
30  IMDaemonCore                            0x1dafba925 0x1dafb0000 + 43301
31  IMSharedUtilities                       0x1a90c0c79 0x1a90ad000 + 81017
32  IMSharedUtilities                       0x1a90c0c7d 0x1a90ad000 + 81021
33  IMSharedUtilities                       0x1a90c1141 0x1a90ad000 + 82241
34  IMSharedUtilities                       0x1a9186515 0x1a90ad000 + 890133
35  IMSharedUtilities                       0x1a9185a71 0x1a90ad000 + 887409
36  IMSharedUtilities                       0x1a9185c21 0x1a90ad000 + 887841
37  libswift_Concurrency.dylib              0x197e66775 completeTaskWithClosure(swift::AsyncContext*, swift::SwiftError*) + 1
```

Figure 3: unsymbolicated imagent crash log (thread 3), device with iOS version 17.7

A more thorough review of the main thread is helpful, but does not highlight a possible root cause because this crash log remains only partly symbolicated.



```
Thread 0 name:   Dispatch queue: com.apple.main-thread
Thread 0:
0   libobjc.A.dylib                         0x184995c24 list_array_tt<unsigned long, protocol_list_t, RawPtr>::iteratorImpl<false>::skipEmptyLists() + 48
1   libobjc.A.dylib                         0x184995a2c list_array_tt<unsigned long, protocol_list_t, RawPtr>::iteratorImpl<false>::operator++() + 316
2   libobjc.A.dylib                         0x1849956f4 class_conformsToProtocol + 416
3   libobjc.A.dylib                         0x184994f90 +[NSObject conformsToProtocol:] + 48
4   Foundation                              0x18b951330 -[NSCoder validateClassSupportsSecureCoding:] + 52
5   Foundation                              0x18b9507bc _encodeObject + 356
6   Foundation                              0x18b950494 -[NSKeyedArchiver _encodeArrayOfObjects:forKey:] + 460
7   Foundation                              0x18b963d30 -[NSSet(NSSet) encodeWithCoder:] + 392
8   Foundation                              0x18b950af4 _encodeObject + 1180
9   Foundation                              0x18b9e0338 +[NSKeyedArchiver archivedDataWithRootObject:requiringSecureCoding:error:] + 92
10  IMDaemonCore                            0x1db059a18 0x1dafb0000 + 694808
11  IMDaemonCore                            0x1db05b064 0x1dafb0000 + 700516
12  IMDaemonCore                            0x1db05a43c 0x1dafb0000 + 697404
13  IMDaemonCore                            0x1db04f310 0x1dafb0000 + 652048
14  IMDaemonCore                            0x1db04e2ac 0x1dafb0000 + 647852
15  IMTransferServices                      0x1ffa7e794 0x1ffa7a000 + 18324
16  libxpc.dylib                            0x1e95cbf30 _xpc_connection_reply_callout + 116
17  libxpc.dylib                            0x1e95be6e0 _xpc_connection_call_reply_async + 80
18  libdispatch.dylib                       0x1949bee54 _dispatch_client_callout3 + 20
19  libdispatch.dylib                       0x1949dc740 _dispatch_mach_msg_async_reply_invoke + 344
20  libdispatch.dylib                       0x1949cd4b4 _dispatch_main_queue_drain + 748
21  libdispatch.dylib                       0x1949cd1b8 _dispatch_main_queue_callback_4CF + 44
22  CoreFoundation                          0x18cae7710 __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ + 16
23  CoreFoundation                          0x18cae4914 __CFRunLoopRun + 1996
24  CoreFoundation                          0x18cae3cd8 CFRunLoopRunSpecific + 608
25  Foundation                              0x18ba04b5c -[NSRunLoop(NSRunLoop) runMode:beforeDate:] + 212
26  Foundation                              0x18ba04a30 -[NSRunLoop(NSRunLoop) run] + 64
27  imagent                                 0x10281f56c 0x1027b8000 + 423276
28  imagent                                 0x102820390 main + 16
29  dyld                                    0x1b02c3154 start + 2356
```

Figure 4: unsymbolicated imagent crash log (thread 0), device with iOS version 17.7

Crash log symbolication is a crucial first step in identifying the code path which led to the crash, enabling researchers to trace the issue back to functions, methods, relevant frameworks, or even specific ARM64 instructions which received execution. We wanted to dive deeper into the `IMDaemonCore` framework. `ipsw` - the "Swiss Army Knife" of iOS/macOS research - positioned us to understand the `IMDaemonCore` code paths by enumerating memory addresses which point to relevant runtime methods. In this example, we'll briefly step away from our "iOS version 17.7" device to walk through the symbolication of a similar crash log generated on iOS version 17.6.1.

```
$ ipsw symbolicate --no-color <IPS FILE>
```

Figure 5: using `ipsw` to symbolicate a target crash log

The `ipsw` command will generate output similar to the following:

```
Thread 1669: queue: com.apple.main-thread
  00: libsystem_kernel.dylib 0x1e0501424 kevent_id + 8
  01: libdispatch.dylib      0x19f7757d4 _dispatch_kq_poll + 228
  02: libdispatch.dylib      0x19f7761bc _dispatch_event_loop_wait_for_ownership + 436
  03: libdispatch.dylib      0x19f762594 __DISPATCH_WAIT_FOR_QUEUE__ + 340
  04: libdispatch.dylib      0x19f76215c _dispatch_sync_f_slow + 148
  05: IDS                    0x1a58b6ea0
  06: IDS                    0x1a5983370
  07: IDS                    0x1a593389c IDSCopyIDForDevice + 360
  08: IMDaemonCore           0x1e5de4b38
  09: IMDaemonCore           0x1e5de3708
  10: IMDaemonCore           0x1e5de9c24
  11: IMDaemonCore           0x1e5deb064
  12: IMDaemonCore           0x1e5dea43c
  13: IMDaemonCore           0x1e5ddf310
  14: IMDaemonCore           0x1e5dde2ac
  15: IMTransferServices     0x20a815794
```

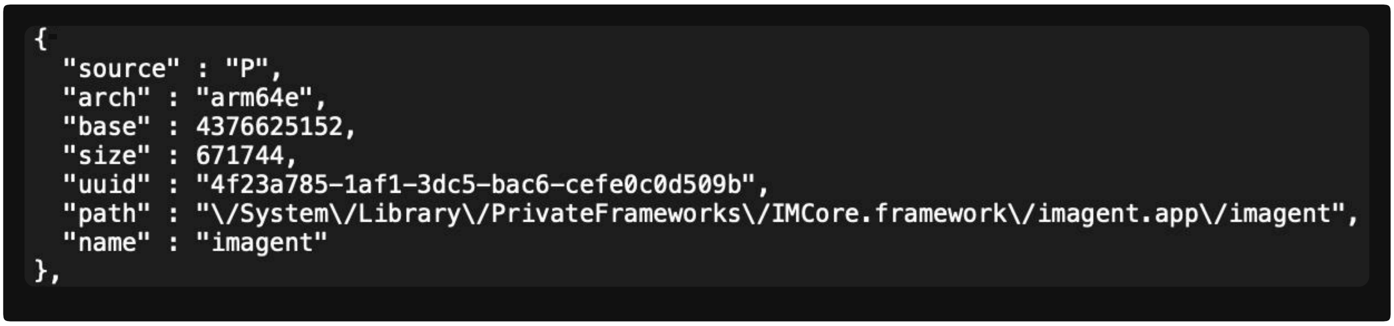
Figure 6: example of symbolicated crash without unslid addresses and relevant symbols; similar to other `imagent` crashes, iOS version 17.6.1

But we have a problem: we can't resolve these specific memory addresses to the `IMDaemonCore` methods because they are "slid" by the presence of ASLR (Address Space Layout Randomization), and the symbols themselves are missing.

We can address the first issue by "unsliding" the memory addresses. If we look at the line that starts with `Shared Cache:` in the `ipsw` output, we can see that it shows the base address of the dyld shared cache (DSC). In this example (featured in figure 4), the address is `0x18f64c000`. By default, the DSC has a base address of `0x180000000`. When the DSC is loaded into memory, it is loaded at a random offset of `0x180000000`. This random offset is the ASLR slide. So for the example above, the slide is `0xf64c000`.

We can write a Python script to compute the ASLR slide from the DSC base address and then use a regular expression to match on the backtrace lines, extract the addresses, and re-write them with their "unslid" values.

There is one caveat to this - the executable images that are not part of the DSC will receive their own ASLR slide, so the DSC slide will not work for those images. The primary example of this will be the main executable image of the running process, `imagent`. Let's briefly look at the `.ips` crash log for this 17.6.1 device.



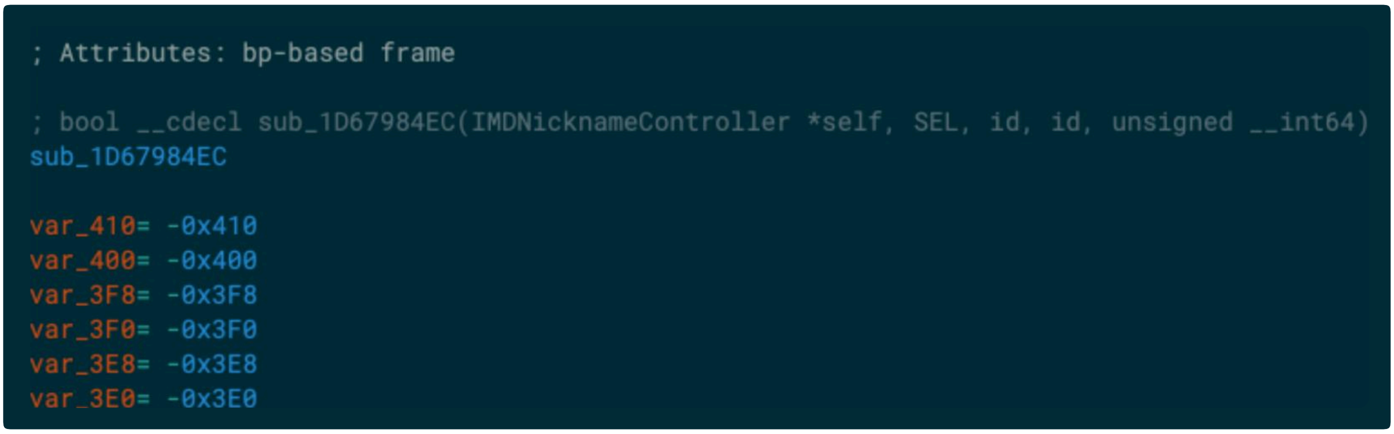
```
{
  "source" : "P",
  "arch" : "arm64e",
  "base" : 4376625152,
  "size" : 671744,
  "uuid" : "4f23a785-1af1-3dc5-bac6-cefe0c0d509b",
  "path" : "\/System\/Library\/PrivateFrameworks\/IMCore.framework\/imagent.app\/imagent",
  "name" : "imagent"
},
```

Figure 7: the binary image of `imagent` as seen in an unsymbolicated crash log

The `base` value represents the address where the image is loaded in memory at runtime. By comparing this base address with the static load address of the Mach-O executable (`imagent`), we can calculate the slide (the random offset applied by ASLR). Once we know the slide, we can adjust the memory addresses from the crash log by subtracting the slide to "unslide" them. While this process could be scripted, we particularly care about the modules in the DSC so we can mostly ignore this calculation.

The other issue is the missing symbols in the backtrace. This can be particularly frustrating because if you load a dylib like `IMDaemonCore` in IDA, you'll notice many symbols are absent. For example, in frame 8 above (figure 6), we see the slid address `0x1e5de4b38`, which corresponds to an unslid address of `0x1d6798b38`.

If we open the DSC in IDA Pro and load IMDaemonCore, we can pinpoint the following function:



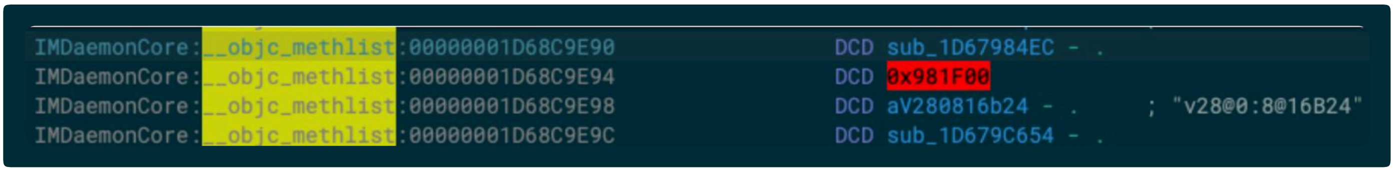
```
; Attributes: bp-based frame

; bool __cdecl sub_1D67984EC(IMDNicknameController *self, SEL, id, id, unsigned __int64)
sub_1D67984EC

var_410= -0x410
var_400= -0x400
var_3F8= -0x3F8
var_3F0= -0x3F0
var_3E8= -0x3E8
var_3E0= -0x3E0
```

Figure 8: function identified after loading IMDaemonCore in IDA; contains the unslid address `0x1d6798b38`

If we look at cross references to this address, we can see that there's a pointer to it in the `__objc_methlist` section of IMDaemonCore, but no corresponding symbols.



```
IMDaemonCore:__objc_methlist:00000001D68C9E90                 DCD sub_1D67984EC - .
IMDaemonCore:__objc_methlist:00000001D68C9E94                 DCD 0x981F00
IMDaemonCore:__objc_methlist:00000001D68C9E98                 DCD aV280816b24 - .      ; "v28@0:8@16B24"
IMDaemonCore:__objc_methlist:00000001D68C9E9C                 DCD sub_1D679C654 - .
```

Figure 9: identifying the same memory address as a `xref` in IDA, but with no symbols

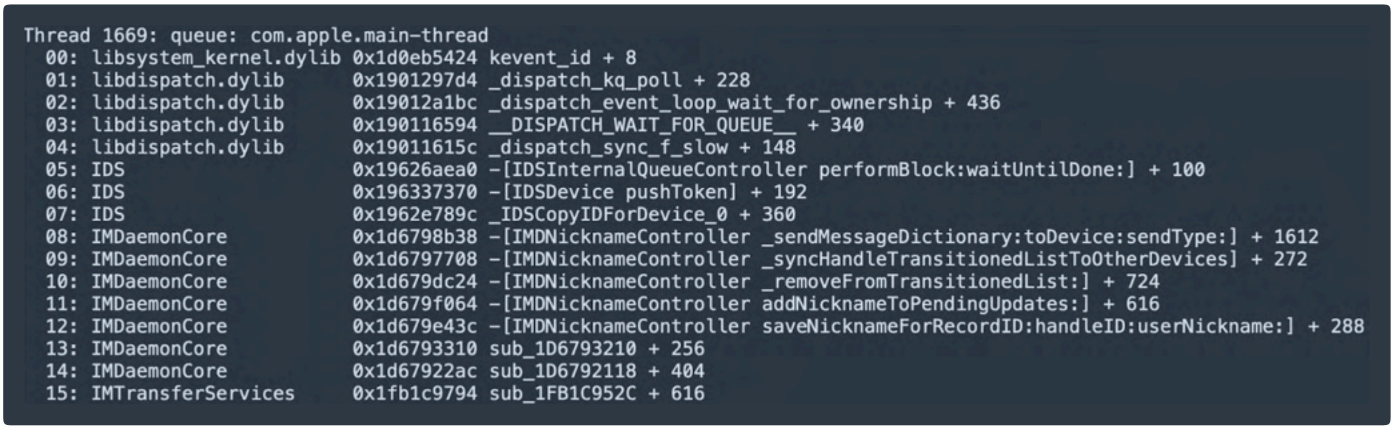
This section contains relative pointers to the `libobjc.A.dylib` shared library. To resolve these pointers to specific methods, go to `File -> Load File -> DYLD Shared Cache Utils -> Load module…` and select the dylib. This will prompt IDA to refresh the section and load the symbols for the Objective-C methods.

```
00000001D68C9E90          DCD __IMDNicknameController__sendMessageDictionary_toDevice_sendType__ - . ; -[IMDN
00000001D68C9E94          DCD sel_markNicknamesAsTransitionedForHandleIDs_isAutoUpdate_ - sel_🤯 ; "markNick
00000001D68C9E94                                    ; "🤯"
00000001D68C9E98          DCD aV280816b24 - .       ; "v28@0:8@16B24"
00000001D68C9E9C          DCD __IMDNicknameController_markNicknamesAsTransitionedForHandleIDs_isAutoUpdate__
00000001D68C9EA0          DCD sel__markCurrentNicknameAsArchived_incrementPendingNicknameVersion_ - sel_🤯 ;
```

Figure 10: newly resolved methods after loading relevant dylibs

We can load the various libraries involved in our backtraces of interest. From there, we can write a simple IDA script to load the backtrace, extract the slid, unslid the addresses, and append the method names to each line.

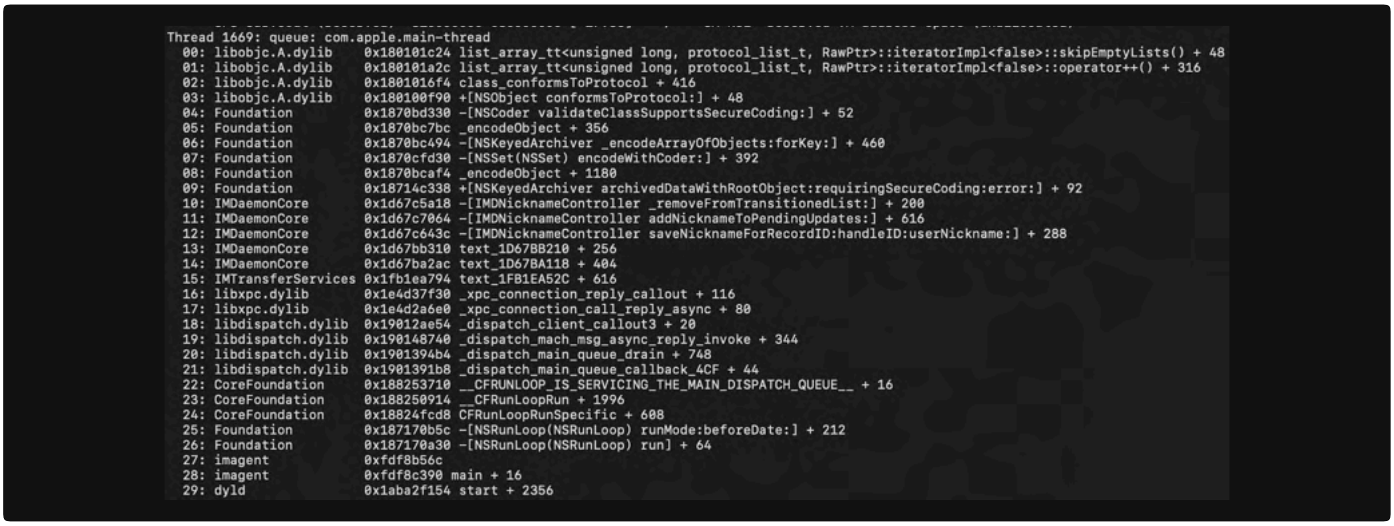
And now we have the newly improved, symbolicated backtrace:



```
Thread 1669: queue: com.apple.main-thread
  00: libsystem_kernel.dylib 0x1d0eb5424 kevent_id + 8
  01: libdispatch.dylib      0x1901297d4 _dispatch_kq_poll + 228
  02: libdispatch.dylib      0x19012a1bc _dispatch_event_loop_wait_for_ownership + 436
  03: libdispatch.dylib      0x190116594 __DISPATCH_WAIT_FOR_QUEUE__ + 340
  04: libdispatch.dylib      0x19011615c _dispatch_sync_f_slow + 148
  05: IDS                    0x19626aea0 -[IDSInternalQueueController performBlock:waitUntilDone:] + 100
  06: IDS                    0x196337370 -[IDSDevice pushToken] + 192
  07: IDS                    0x1962e789c _IDSCopyIDForDevice_0 + 360
  08: IMDaemonCore           0x1d6798b38 -[IMDNicknameController _sendMessageDictionary:toDevice:sendType:] + 1612
  09: IMDaemonCore           0x1d6797708 -[IMDNicknameController _syncHandleTransitionedListToOtherDevices] + 272
  10: IMDaemonCore           0x1d679dc24 -[IMDNicknameController _removeFromTransitionedList:] + 724
  11: IMDaemonCore           0x1d679f064 -[IMDNicknameController addNicknameToPendingUpdates:] + 616
  12: IMDaemonCore           0x1d679e43c -[IMDNicknameController saveNicknameForRecordID:handleID:userNickname:] + 288
  13: IMDaemonCore           0x1d6793310 sub_1D6793210 + 256
  14: IMDaemonCore           0x1d67922ac sub_1D6792118 + 404
  15: IMTransferServices     0x1fb1c9794 sub_1FB1C952C + 616
```

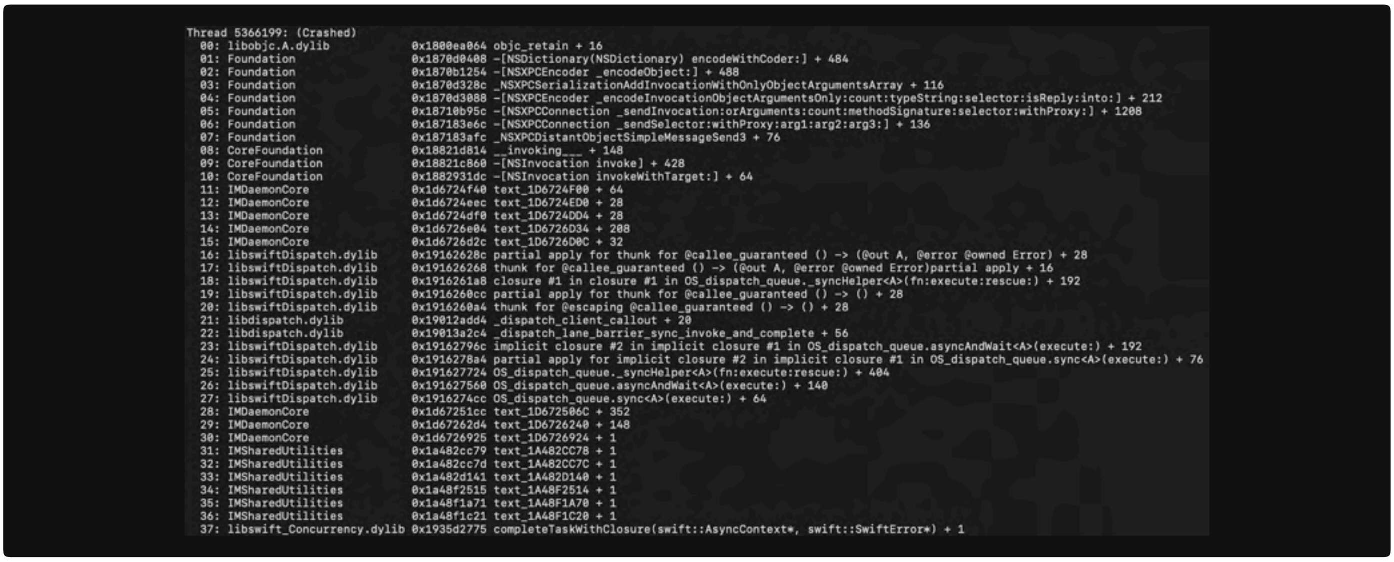
Figure 11: example `imagent` crash from iOS version 17.6.1 after resolving unslid memory addresses to relevant Objective-C methods

The process can be repeated for the `imagent` crashes we have on hand, both for their main threads and the specific threads responsible for crash conditions.



```
Thread 1669: queue: com.apple.main-thread
  00: libobjc.A.dylib    0x180101c24 list_array_tt<unsigned long, protocol_list_t, RawPtr>::iteratorImpl<false>::skipEmptyLists() + 48
  01: libobjc.A.dylib    0x180101a2c list_array_tt<unsigned long, protocol_list_t, RawPtr>::iteratorImpl<false>::operator++() + 316
  02: libobjc.A.dylib    0x1801016f4 class_conformsToProtocol + 416
  03: libobjc.A.dylib    0x180100f90 +[NSObject conformsToProtocol:] + 48
  04: Foundation         0x1870bd330 -[NSCoder validateClassSupportsSecureCoding:] + 52
  05: Foundation         0x1870bc7bc _encodeObject + 356
  06: Foundation         0x1870bc494 -[NSKeyedArchiver _encodeArrayOfObjects:forKey:] + 460
  07: Foundation         0x1870cfd30 -[NSSet(NSSet) encodeWithCoder:] + 392
  08: Foundation         0x1870bcaf4 _encodeObject + 1180
  09: Foundation         0x18714c338 +[NSKeyedArchiver archivedDataWithRootObject:requiringSecureCoding:error:] + 92
  10: IMDaemonCore       0x1d67c5a18 -[IMDNicknameController _removeFromTransitionedList:] + 200
  11: IMDaemonCore       0x1d67c7064 -[IMDNicknameController addNicknameToPendingUpdates:] + 616
  12: IMDaemonCore       0x1d67c643c -[IMDNicknameController saveNicknameForRecordID:handleID:userNickname:] + 288
  13: IMDaemonCore       0x1d67bb310 text_1D67BB210 + 256
  14: IMDaemonCore       0x1d67ba2ac text_1D67BA118 + 404
  15: IMTransferServices 0x1fb1ea794 text_1FB1EA52C + 616
  16: libxpc.dylib       0x1e4d37f30 _xpc_connection_reply_callout + 116
  17: libxpc.dylib       0x1e4d2a6e0 _xpc_connection_call_reply_async + 80
  18: libdispatch.dylib  0x19012ae54 _dispatch_client_callout3 + 20
  19: libdispatch.dylib  0x190148740 _dispatch_mach_msg_async_reply_invoke + 344
  20: libdispatch.dylib  0x1901394b4 _dispatch_main_queue_drain + 748
  21: libdispatch.dylib  0x1901391b8 _dispatch_main_queue_callback_4CF + 44
  22: CoreFoundation     0x188253710 __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ + 16
  23: CoreFoundation     0x188250914 __CFRunLoopRun + 1996
  24: CoreFoundation     0x18824fcd8 CFRunLoopRunSpecific + 608
  25: Foundation         0x187170b5c -[NSRunLoop(NSRunLoop) runMode:beforeDate:] + 212
  26: Foundation         0x187170a30 -[NSRunLoop(NSRunLoop) run] + 64
  27: imagent            0xfdf8b56c
  28: imagent            0xfdf8c390 main + 16
  29: dyld               0x1aba2f154 start + 2356
```

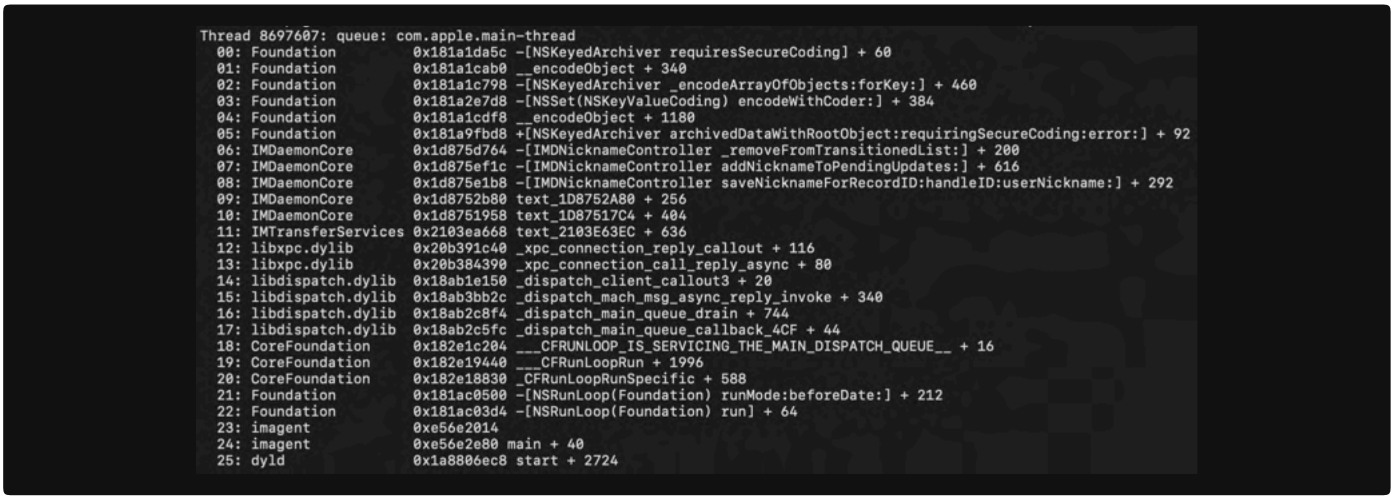
Figure 12: symbolicated `imagent` crash log (thread 0), device with iOS version 17.7



```
Thread 5366199: (Crashed)
  00: libobjc.A.dylib            0x1800ea064 objc_retain + 16
  01: Foundation                 0x1870d0408 -[NSDictionary(NSDictionary) encodeWithCoder:] + 484
  02: Foundation                 0x1870b1254 -[NSXPCEncoder _encodeObject:] + 488
  03: Foundation                 0x1870d328c _NSXPCSerializationAddInvocationWithOnlyObjectArgumentsArray + 116
  04: Foundation                 0x1870d3088 -[NSXPCEncoder _encodeInvocationObjectArgumentsOnly:count:typeString:selector:isReply:into:] + 212
  05: Foundation                 0x18710b95c -[NSXPCConnection _sendInvocation:orArguments:count:methodSignature:selector:withProxy:] + 1208
  06: Foundation                 0x187183e6c -[NSXPCConnection _sendSelector:withProxy:arg1:arg2:arg3:] + 136
  07: Foundation                 0x187183afc _NSXPCDistantObjectSimpleMessageSend3 + 76
  08: CoreFoundation             0x18821d814 __invoking___ + 148
  09: CoreFoundation             0x18821c860 -[NSInvocation invoke] + 428
  10: CoreFoundation             0x1882931dc -[NSInvocation invokeWithTarget:] + 64
  11: IMDaemonCore               0x1d6724f40 text_1D6724F00 + 64
  12: IMDaemonCore               0x1d6724eec text_1D6724ED0 + 28
  13: IMDaemonCore               0x1d6724df0 text_1D6724DD4 + 28
  14: IMDaemonCore               0x1d6726e04 text_1D6726D34 + 208
  15: IMDaemonCore               0x1d6726d2c text_1D6726D0C + 32
  16: libswiftDispatch.dylib     0x19162628c partial apply for thunk for @callee_guaranteed () -> (@out A, @error @owned Error) + 28
  17: libswiftDispatch.dylib     0x191626268 thunk for @callee_guaranteed () -> (@out A, @error @owned Error)partial apply + 16
  18: libswiftDispatch.dylib     0x1916261a8 closure #1 in closure #1 in OS_dispatch_queue._syncHelper<A>(fn:execute:rescue:) + 192
  19: libswiftDispatch.dylib     0x1916260cc partial apply for thunk for @callee_guaranteed () -> () + 28
  20: libswiftDispatch.dylib     0x1916260a4 thunk for @escaping @callee_guaranteed () -> () + 28
  21: libdispatch.dylib          0x19012add4 _dispatch_client_callout + 20
  22: libdispatch.dylib          0x19013a2c4 _dispatch_lane_barrier_sync_invoke_and_complete + 56
  23: libswiftDispatch.dylib     0x19162796c implicit closure #2 in implicit closure #1 in OS_dispatch_queue.asyncAndWait<A>(execute:) + 192
  24: libswiftDispatch.dylib     0x1916278a4 partial apply for implicit closure #2 in implicit closure #1 in OS_dispatch_queue.sync<A>(execute:) + 76
  25: libswiftDispatch.dylib     0x191627724 OS_dispatch_queue._syncHelper<A>(fn:execute:rescue:) + 404
  26: libswiftDispatch.dylib     0x191627560 OS_dispatch_queue.asyncAndWait<A>(execute:) + 140
  27: libswiftDispatch.dylib     0x1916274cc OS_dispatch_queue.sync<A>(execute:) + 64
  28: IMDaemonCore               0x1d67251cc text_1D672506C + 352
  29: IMDaemonCore               0x1d67262d4 text_1D6726240 + 148
  30: IMDaemonCore               0x1d6726925 text_1D6726924 + 1
  31: IMSharedUtilities          0x1a482cc79 text_1A482CC78 + 1
  32: IMSharedUtilities          0x1a482cc7d text_1A482CC7C + 1
  33: IMSharedUtilities          0x1a482d141 text_1A482D140 + 1
  34: IMSharedUtilities          0x1a48f2515 text_1A48F2514 + 1
  35: IMSharedUtilities          0x1a48f1a71 text_1A48F1A70 + 1
  36: IMSharedUtilities          0x1a48f1c21 text_1A48F1C20 + 1
  37: libswift_Concurrency.dylib 0x1935d2775 completeTaskWithClosure(swift::AsyncContext*, swift::SwiftError*) + 1
```

Figure 13: symbolicated `imagent` crash log (thread 3), device with iOS version 17.7

So far the iVerify team was able to identify numerous crashes impacting various versions of iOS 17. Here we'll highlight an additional crash impacting iOS major version 18.



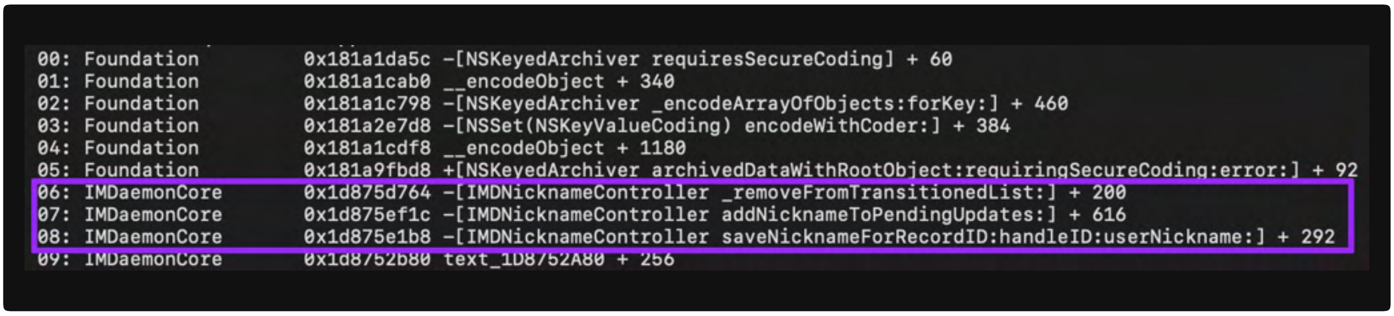
```
Thread 8697607: queue: com.apple.main-thread
  00: Foundation          0x181a1da5c -[NSKeyedArchiver requiresSecureCoding] + 60
  01: Foundation          0x181a1cab0 __encodeObject + 340
  02: Foundation          0x181a1c798 -[NSKeyedArchiver _encodeArrayOfObjects:forKey:] + 460
  03: Foundation          0x181a2e7d8 -[NSSet(NSKeyValueCoding) encodeWithCoder:] + 384
  04: Foundation          0x181a1cdf8 __encodeObject + 1180
  05: Foundation          0x181a9fbd8 +[NSKeyedArchiver archivedDataWithRootObject:requiringSecureCoding:error:] + 92
  06: IMDaemonCore        0x1d875d764 -[IMDNicknameController _removeFromTransitionedList:] + 200
  07: IMDaemonCore        0x1d875ef1c -[IMDNicknameController addNicknameToPendingUpdates:] + 616
  08: IMDaemonCore        0x1d875e1b8 -[IMDNicknameController saveNicknameForRecordID:handleID:userNickname:] + 292
  09: IMDaemonCore        0x1d8752b80 text_1D8752A80 + 256
  10: IMDaemonCore        0x1d8751958 text_1D87517C4 + 404
  11: IMTransferServices 0x2103ea668 text_2103E63EC + 636
  12: libxpc.dylib        0x20b391c40 _xpc_connection_reply_callout + 116
  13: libxpc.dylib        0x20b384390 _xpc_connection_call_reply_async + 80
  14: libdispatch.dylib   0x18ab1e150 _dispatch_client_callout3 + 20
  15: libdispatch.dylib   0x18ab3bb2c _dispatch_mach_msg_async_reply_invoke + 340
  16: libdispatch.dylib   0x18ab2c8f4 _dispatch_main_queue_drain + 744
  17: libdispatch.dylib   0x18ab2c5fc _dispatch_main_queue_callback_4CF + 44
  18: CoreFoundation      0x182e1c204 __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ + 16
  19: CoreFoundation      0x182e19440 __CFRunLoopRun + 1996
  20: CoreFoundation      0x182e18830 _CFRunLoopRunSpecific + 588
  21: Foundation          0x181ac0500 -[NSRunLoop(Foundation) runMode:beforeDate:] + 212
  22: Foundation          0x181ac03d4 -[NSRunLoop(Foundation) run] + 64
  23: imagent             0xe56e2014
  24: imagent             0xe56e2e80 main + 40
  25: dyld                0x1a8806ec8 start + 2724
```

Figure 14: symbolicated imagent crash log (thread 0), device with iOS version 18.1.1

```
Thread 9472249: (Crashed)
  00: libobjc.A.dylib            0x180105c78 _objc_retain + 16
  01: Foundation                 0x181a2eeb0 -[NSDictionary(Foundation) encodeWithCoder:] + 476
  02: Foundation                 0x181a113d0 -[NSXPCEncoder _encodeObject:] + 488
  03: Foundation                 0x181a31ea0 __NSXPCSerializationAddInvocationWithOnlyObjectArgumentsArray + 116
  04: Foundation                 0x181a31c9c -[NSXPCEncoder _encodeInvocationObjectArgumentsOnly:count:typeString:selector:isReply:into:] + 212
  05: Foundation                 0x181a6675c -[NSXPCConnection _sendInvocation:orArguments:count:methodSignature:selector:withProxy:] + 1208
  06: Foundation                 0x181acdfe4 -[NSXPCConnection _sendSelector:withProxy:arg1:arg2:arg3:] + 136
  07: Foundation                 0x181acdc74 __NSXPCDistantObjectSimpleMessageSend3 + 76
  08: CoreFoundation             0x182de6374 ___invoking___ + 148
  09: CoreFoundation             0x182de53c4 -[NSInvocation invoke] + 428
  10: CoreFoundation             0x182e5acb8 -[NSInvocation invokeWithTarget:] + 64
  11: IMDaemonCore               0x1d867a8e8 text_1D867A8A8 + 64
  12: IMDaemonCore               0x1d867a894 text_1D867A878 + 28
  13: IMDaemonCore               0x1d867a798 text_1D867A77C + 28
  14: IMDaemonCore               0x1d88e18a0 text_1D88E17E0 + 192
  15: IMSharedUtilities          0x1a0564fe4 text_1A0564F78 + 108
  16: libswiftDispatch.dylib     0x18c13aad8 _$sxs5Error_pIgrzo_xsAA_pIegrzo_lTRTA + 28
  17: libswiftDispatch.dylib     0x18c13aab4 _$sxs5Error_pIgrzo_xsAA_pIegrzo_lTRTA.78 + 16
  18: libswiftDispatch.dylib     0x18c13aa1c _$sSo17OS_dispatch_queueC8DispatchE11_syncHelper33_F417D752D2C4E9330E1C700411CE0C6ALL2fn7execute6rescuexyyyXEXE_xyKXExs5Error_pKXEtKlFyxyKcXEfU_yyXEfU_ + 192
  19: libswiftDispatch.dylib     0x18c13a940 _$sIg_Ieg_TRTA + 28
  20: libswiftDispatch.dylib     0x18c13a918 _$sIeg_IyB_TR_18C13A8FC + 28
  21: libdispatch.dylib          0x18ab1e0d0 _dispatch_client_callout + 20
  22: libdispatch.dylib          0x18ab2d750 _dispatch_lane_barrier_sync_invoke_and_complete + 56
  23: libswiftDispatch.dylib     0x18c13c1c0 _$sSo17OS_dispatch_queueC8DispatchE12asyncAndWait7executexxyKXE_tKlFyyyXEcABcfu_yyyXEcfu0_Tm + 192
  24: libswiftDispatch.dylib     0x18c13c0f8 _$sSo17OS_dispatch_queueC8DispatchE4sync7executexxyKXE_tKlFyyyXEcABcfu_yyyXEcfu0_TA + 76
  25: libswiftDispatch.dylib     0x18c13bf78 _$sSo17OS_dispatch_queueC8DispatchE11_syncHelper33_F417D752D2C4E9330E1C700411CE0C6ALL2fn7execute6rescuexyyyXEXE_xyKXExs5Error_pKXEtKlF + 404
  26: libswiftDispatch.dylib     0x18c13bdb4 _$sSo17OS_dispatch_queueC8DispatchE12asyncAndWait7executexxyKXE_tKlFTm + 140
  27: libswiftDispatch.dylib     0x18c13bd20 _$sSo17OS_dispatch_queueC8DispatchE4sync7executexxyKXE_tKlF + 64
  28: IMSharedUtilities          0x1a05648c0 _$s17IMSharedUtilities15ProtectiveQueueC4sync7executeqd__qd__xzXE_tlF + 72
  29: IMDaemonCore               0x1d867ab6c text_1D867AA14 + 344
  30: IMDaemonCore               0x1d88d47b8 text_1D88D4730 + 136
  31: IMDaemonCore               0x1d867baf5 text_1D867BAF4 + 1
  32: IMSharedUtilities          0x1a046c6e1 text_1A046C6E0 + 1
  33: IMSharedUtilities          0x1a046c6e1 text_1A046C6E0 + 1
  34: IMSharedUtilities          0x1a0555d5d text_1A0555D5C + 1
  35: IMSharedUtilities          0x1a046c6e1 text_1A046C6E0 + 1
  36: IMSharedUtilities          0x1a057b80d text_1A057B80C + 1
  37: IMSharedUtilities          0x1a046cc11 text_1A046CC10 + 1
  38: libswift_Concurrency.dylib 0x18e52be39 __ZL23completeTaskWithClosurePN5swift12AsyncContextEPNS_10SwiftErrorE + 1
```

Figure 15: symbolicated imagent crash log (thread 4), device with iOS version 18.1.1

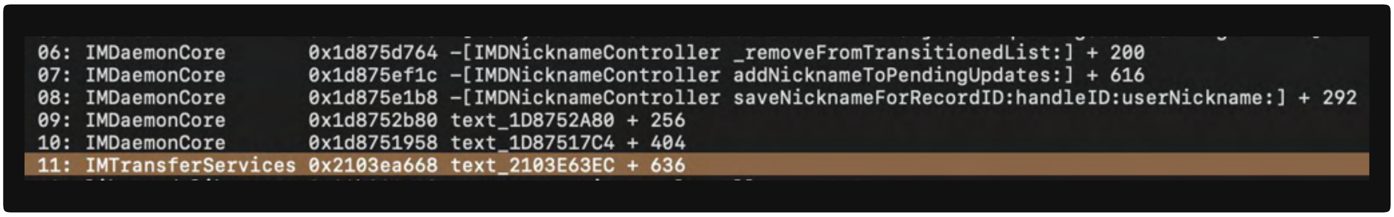
There was another common thread between these anomalous crash logs, most noticeable after supplementing their contents with symbols: all of the crash logs involved iMessage “Nicknames” and, more specifically, “Nickname Updates.”



```
00: Foundation         0x181a1da5c -[NSKeyedArchiver requiresSecureCoding] + 60
01: Foundation         0x181a1cab0 __encodeObject + 340
02: Foundation         0x181a1c798 -[NSKeyedArchiver _encodeArrayOfObjects:forKey:] + 460
03: Foundation         0x181a2e7d8 -[NSSet(NSKeyValueCoding) encodeWithCoder:] + 384
04: Foundation         0x181a1cdf8 __encodeObject + 1180
05: Foundation         0x181a9fbd8 +[NSKeyedArchiver archivedDataWithRootObject:requiringSecureCoding:error:] + 92
06: IMDaemonCore       0x1d875d764 -[IMDNicknameController _removeFromTransitionedList:] + 200
07: IMDaemonCore       0x1d875ef1c -[IMDNicknameController addNicknameToPendingUpdates:] + 616
08: IMDaemonCore       0x1d875e1b8 -[IMDNicknameController saveNicknameForRecordID:handleID:userNickname:] + 292
09: IMDaemonCore       0x1d8752b80 text_1D8752A80 + 256
```

Figure 16: symbolicated crash log indicating the use of IMDNicknameController

And in frame 11, we can clearly identify the peculiar transition from `IMTransferServices`, a service dedicated to managing the transfer of media and attachments within iMessage on iOS, including handling uploads and downloads to and from iCloud. This service also coordinates file transfers, ensures reliability, and manages background operations to continue transfers even when the iMessage app is inactive.



```
06: IMDaemonCore       0x1d875d764 -[IMDNicknameController _removeFromTransitionedList:] + 200
07: IMDaemonCore       0x1d875ef1c -[IMDNicknameController addNicknameToPendingUpdates:] + 616
08: IMDaemonCore       0x1d875e1b8 -[IMDNicknameController saveNicknameForRecordID:handleID:userNickname:] + 292
09: IMDaemonCore       0x1d8752b80 text_1D8752A80 + 256
10: IMDaemonCore       0x1d8751958 text_1D87517C4 + 404
11: IMTransferServices 0x2103ea668 text_2103E63EC + 636
```

Figure 17: symbolicated crash log, frame transition from IMTransferAgent

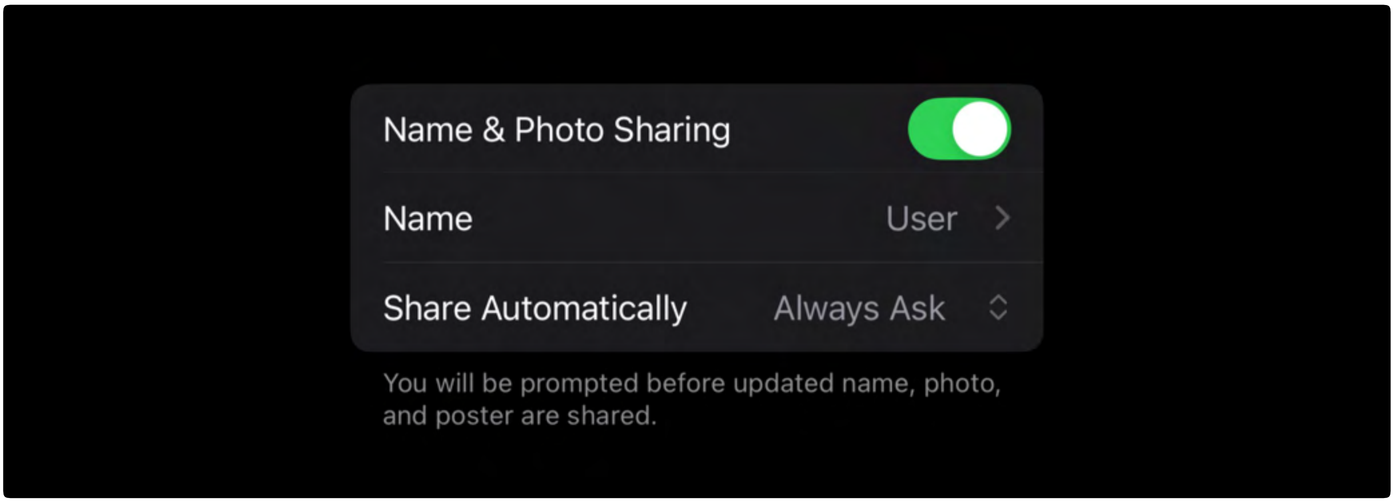
Interestingly, all of these crashes are fundamentally similar in this way - that is, terminating later as a result of some kind of memory corruption (e.g. UAF) after some form of hand-off from iMessage to the `IMDaemonCore`. We decided to dive into the Nickname feature and unearth the bug at hand.

What’s in a Nickname Update?

In recent versions of iOS, it is possible for a user to set a nickname, avatar, and wallpaper for their contact card. It can be shared with other contacts over iMessage. At a high level, Nickname Updates are sent from `imagent` to `IMTransferAgent` which will download the associated data from iCloud. Once the download is successful, the content is passed through BlastDoor. `imagent` will eventually receive this Nickname Update. To adequately understand the data flow of a Nickname Update and identify how these Nickname Updates might trigger the crashes, we instrumented the `Share Name and Photo` feature for testing.

Our test bed consisted of an iOS device running iOS version 18.3 and an M1 Macbook Air on macOS 15.1.1. We disabled System Integrity Protection (SIP) on the Macbook so we could attach ourselves to and effectively debug system processes related to iMessage. Both macOS and iOS share much of the same code for iMessage, so there should be minimal differences between the two. For the reader’s awareness, we will note that the information below may not be entirely accurate for iOS depending on the version.

We started our tests by enabling the `Share Name and Photo` feature on the iOS device. We navigated to `Settings > Apps > Messages > Share Name and Photo` and enabled `Name & Photo Sharing`, shown below:



Name & Photo Sharing ●

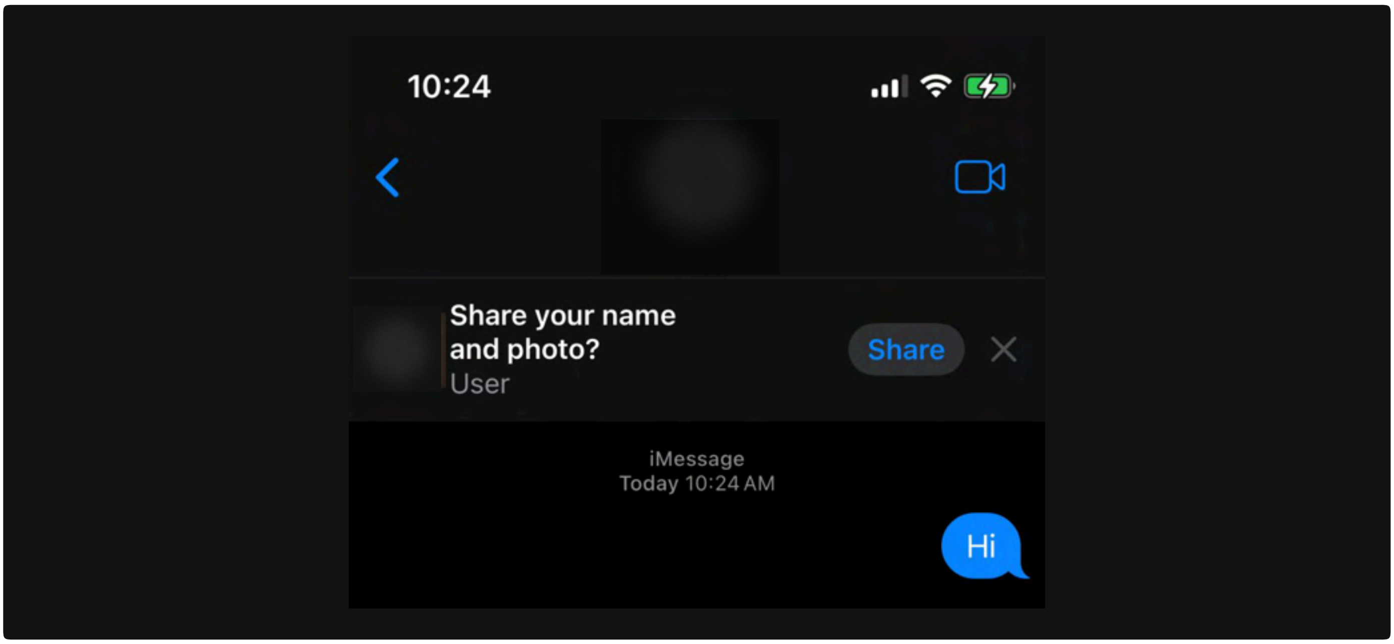
Name: User >

Share Automatically: Always Ask

You will be prompted before updated name, photo, and poster are shared.

Figure 18: Name & Photo Sharing feature in Settings

The `Share Automatically` field can be set to either `Always Ask` or `Contacts Only`. If set to `Contacts Only`, then the Nickname Updates will be sent whenever the user sends a message to a contact. If set to `Always Ask`, the user will see the following prompt in the Messages app whenever they make changes to their Name & Photo for iMessage:



Share your name and photo?

User

iMessage

Today 10:24 AM

Hi

Figure 19: An example of the Always Ask prompt

We verified that Nickname Updates are processed by the receiver even if the sender is unknown. In this circumstance, the sender would have to send an initial message from the Messages app to be able to send Nickname Updates from the UI, but this likely is not required by iMessage itself which would make this a potentially suitable attack surface for zero-click attacks.

How Did a Nickname Update Trigger the Crash?

When we sent a Nickname Update, it resulted in the method call:

```
`-[IMDNicknameController
service:account:incomingTopLevelMessage:fromID:messageContext:]’
```

This method decrypts and unpacks the message (represented as a dictionary); in a subsequent call to:

```
`-[IMDNicknameController
getNicknameWithRecordID:decryptionKey:wallpaperDataTag:wallpaperLowResDataTag:wallpa
perMetadata Tag:isKnownSender:shouldDecodeImageFields:completionBlock:]`
```

a message is sent to `IMTransferAgent` to retrieve the nickname data and define a block callback after the download is complete.

Once the data is downloaded from iCloud, the data is passed as a dictionary to `MessagesBlastDoorService`. In our review of this code path, it looked like the data returned from Blastdoor varies depending on whether the sender is unknown or known to the receiver. If the sender is unknown, the image data will not be processed.

The data from BlastDoor is transformed into an `IMNickname` object and is passed to the completion block for the original download request. The newly transformed `IMNickname` object traverses a couple of other block callbacks before calling the following methods, in sequence:

```
`- [IMDNicknameController saveNicknameForRecordID:handleID:userNickname:]`
```

```
`- [IMDNicknameController addNicknameToPendingUpdates:]`
```

```
`- [IMDNicknameController addNicknameToPendingUpdates:]`
```

So what happens after the call to `- [IMDNicknameController addNicknameToPendingUpdates:]` ?

There is a method call to the selector

```
`updatePendingNicknameUpdates:handledNicknames:archivedNicknames:`
```

but it is difficult to discern what exactly implements that method. In the shared cache, we identified at least two classes which implement that particular selector:

```
`IntentsClientBroadcaster`
```

```
`_IMLegacyDaemonListener`
```

In this iteration of the test bed, we failed to hit either of those methods. This is where the differences between iOS and macOS may have complicated matters. Instead, the object which ultimately received the message was an `IMInvocationCapturingProxy` which, once again, resulted in a relative dead end - `xrefs` in IDA pointed to Swift async methods from the `ClientConnectionManager` class (which can also be found inside of the crash logs), but this bore no fruit.

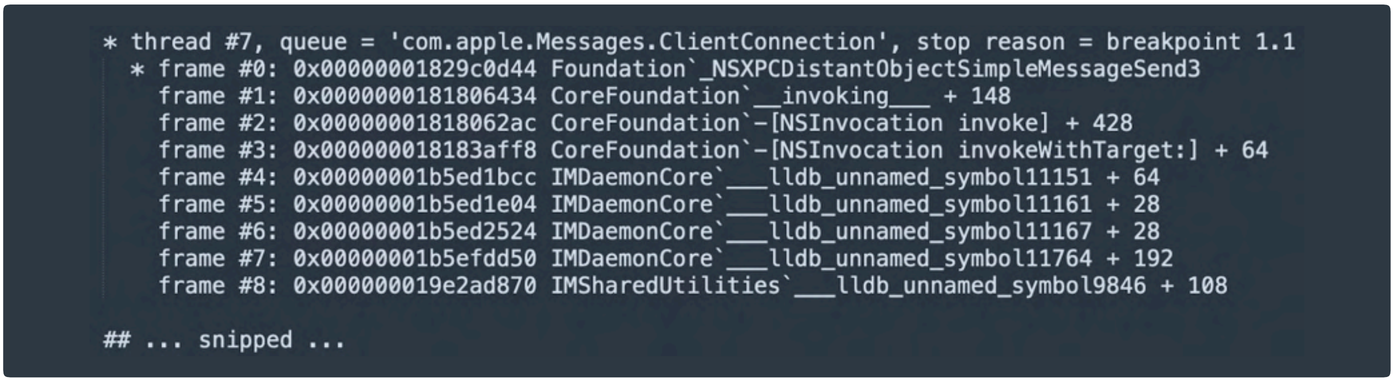
Ultimately, we were able to instrument Frida on our macOS test device to effectively trace all methods with the selector:

```
`updatePendingNicknameUpdates:handledNicknames:archivedNicknames:`
```

We identified the method:

```
`- [__NSXPCInterfaceProxy_IMDaemonListenerProtocol
updatePendingNicknameUpdates:handledNicknames:archivedNicknames:]`
```

After placing a breakpoint on this particular method, we finally generated a familiar backtrace (reference figure 2, frame 7).

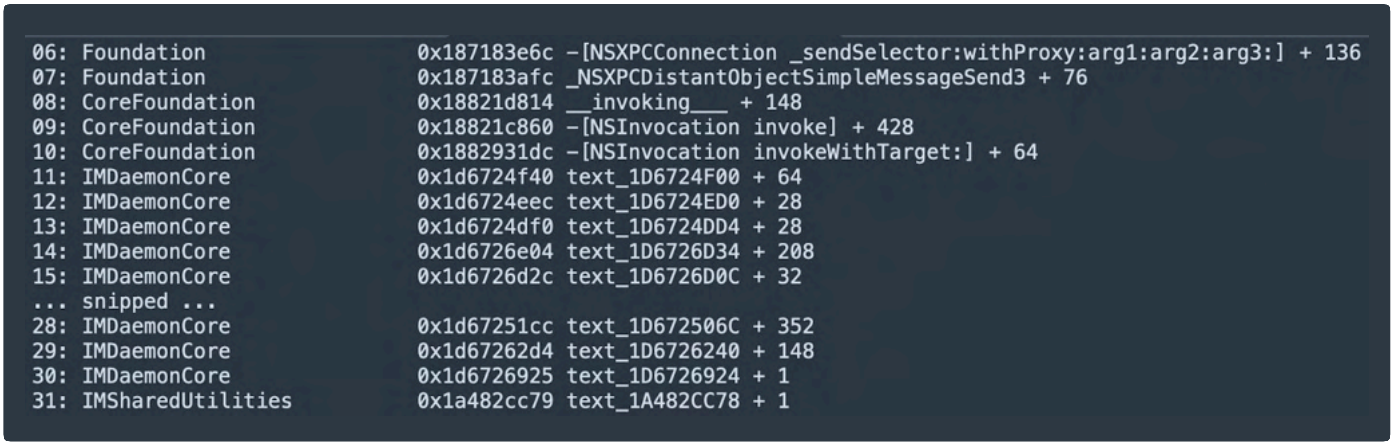


```
* thread #7, queue = 'com.apple.Messages.ClientConnection', stop reason = breakpoint 1.1
  * frame #0: 0x00000001829c0d44 Foundation`_NSXPCDistantObjectSimpleMessageSend3
    frame #1: 0x0000000181806434 CoreFoundation`__invoking___ + 148
    frame #2: 0x00000001818062ac CoreFoundation`-[NSInvocation invoke] + 428
    frame #3: 0x000000018183aff8 CoreFoundation`-[NSInvocation invokeWithTarget:] + 64
    frame #4: 0x00000001b5ed1bcc IMDaemonCore`___lldb_unnamed_symbol11151 + 64
    frame #5: 0x00000001b5ed1e04 IMDaemonCore`___lldb_unnamed_symbol11161 + 28
    frame #6: 0x00000001b5ed2524 IMDaemonCore`___lldb_unnamed_symbol11167 + 28
    frame #7: 0x00000001b5efdd50 IMDaemonCore`___lldb_unnamed_symbol11764 + 192
    frame #8: 0x000000019e2ad870 IMSharedUtilities`___lldb_unnamed_symbol9846 + 108

## ... snipped ...
```

Figure 20: hitting the breakpoint on the macOS and generating a backtrace

We can see that it’s almost identical to the crash log referenced in figure 2 in which the `_NSXPCDistantObjectSimpleMessageSend3` method is called on frame 7.



```
06: Foundation                 0x187183e6c -[NSXPCConnection _sendSelector:withProxy:arg1:arg2:arg3:] + 136
07: Foundation                 0x187183afc _NSXPCDistantObjectSimpleMessageSend3 + 76
08: CoreFoundation             0x18821d814 __invoking___ + 148
09: CoreFoundation             0x18821c860 -[NSInvocation invoke] + 428
10: CoreFoundation             0x1882931dc -[NSInvocation invokeWithTarget:] + 64
11: IMDaemonCore               0x1d6724f40 text_1D6724F00 + 64
12: IMDaemonCore               0x1d6724eec text_1D6724ED0 + 28
13: IMDaemonCore               0x1d6724df0 text_1D6724DD4 + 28
14: IMDaemonCore               0x1d6726e04 text_1D6726D34 + 208
15: IMDaemonCore               0x1d6726d2c text_1D6726D0C + 32
... snipped ...
28: IMDaemonCore               0x1d67251cc text_1D672506C + 352
29: IMDaemonCore               0x1d67262d4 text_1D6726240 + 148
30: IMDaemonCore               0x1d6726925 text_1D6726924 + 1
31: IMSharedUtilities          0x1a482cc79 text_1A482CC78 + 1
```

Figure 21: a nearly identical backtrace, as seen in our first crash log (iOS 17.7)

Here we can see that `_NSXPCDistantObjectSimpleMessageSend3()` retrieves an object from offset `0x8` of `x0`; it calls the method `sendSelector:withProxy:arg1:arg2:arg3:` with this object in `x0`. If we examine this object, we can see that it is an `NSXPCConnection` object:

```
<NSXPCConnection: 0x63276300> connection from pid 1000 on mach service named
com.apple.imagent.desktop.auth
```

Here we can see that `_NSXPCDistantObjectSimpleMessageSend3()` retrieves an object from offset `0x8` of `x0`; it calls the method `sendSelector:withProxy:arg1:arg2:arg3:` with this object in `x0`. If we examine this object, we can see that it is an `NSXPCConnection` object:

Diffing Out a Path Forward

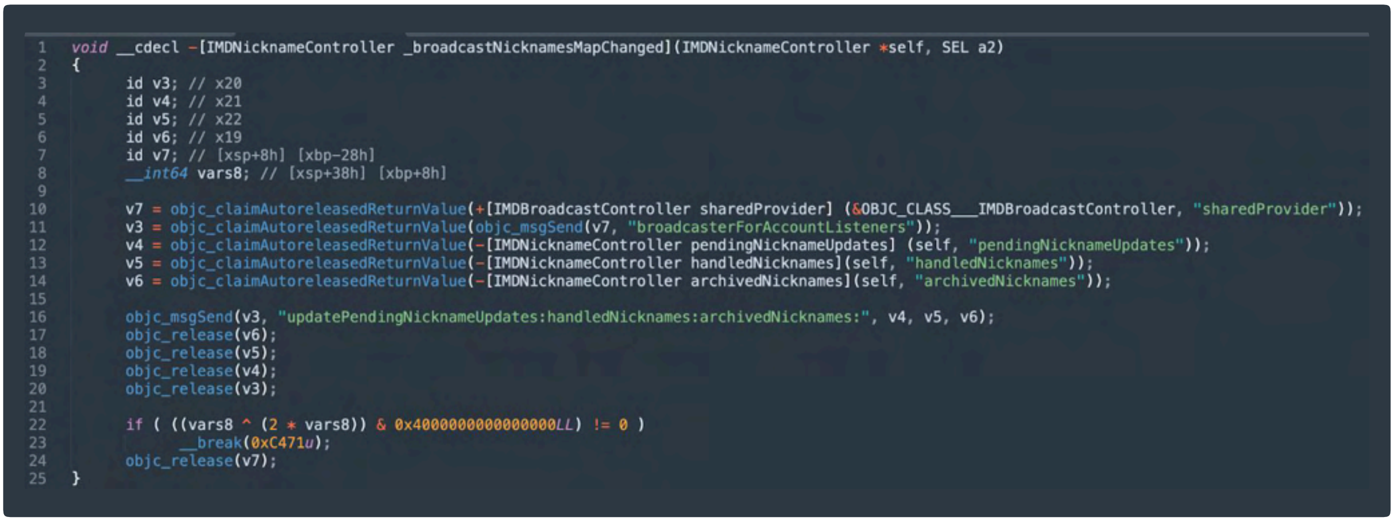
After successfully identifying a viable code path responsible for directing our newly generated “Nickname Updates”, we still needed to identify a potential vulnerability that could “trigger” the crash. We additionally wanted to discern whether that potential vulnerability could already be fixed. The latest crash derived from our telemetry occurred on iOS 18.1.1, but the current iOS version, at the time of our analysis, was 18.3.

If we could concretely identify code changes in 18.3 related to Nickname Updates which successfully address this vulnerability, we might reveal the specific building blocks needed to instrument, trigger, and reproduce the crash in earlier versions of iOS. If we found no changes to the relevant code paths in iOS 18.3, there was a chance we could assist Apple in patching this vulnerability.

Our team spent several days analyzing code changes between iOS versions. iOS versions 18.2 and 18.3 specifically changed one method relevant to the Nickname Update call sequence we have so far analyzed.

The method `-[IMDNicknameController _broadcastNicknamesMapChanged]` is called after various Nickname operations are completed. The `IMDNicknameController` class has three `NSMutableDictionary` instance variables: `pendingNicknameUpdates`, `handledNicknames`, and `archivedNicknames`.

In iOS 18.2, `-[IMDNicknameController _broadcastNicknamesMapChanged]` looks like this:



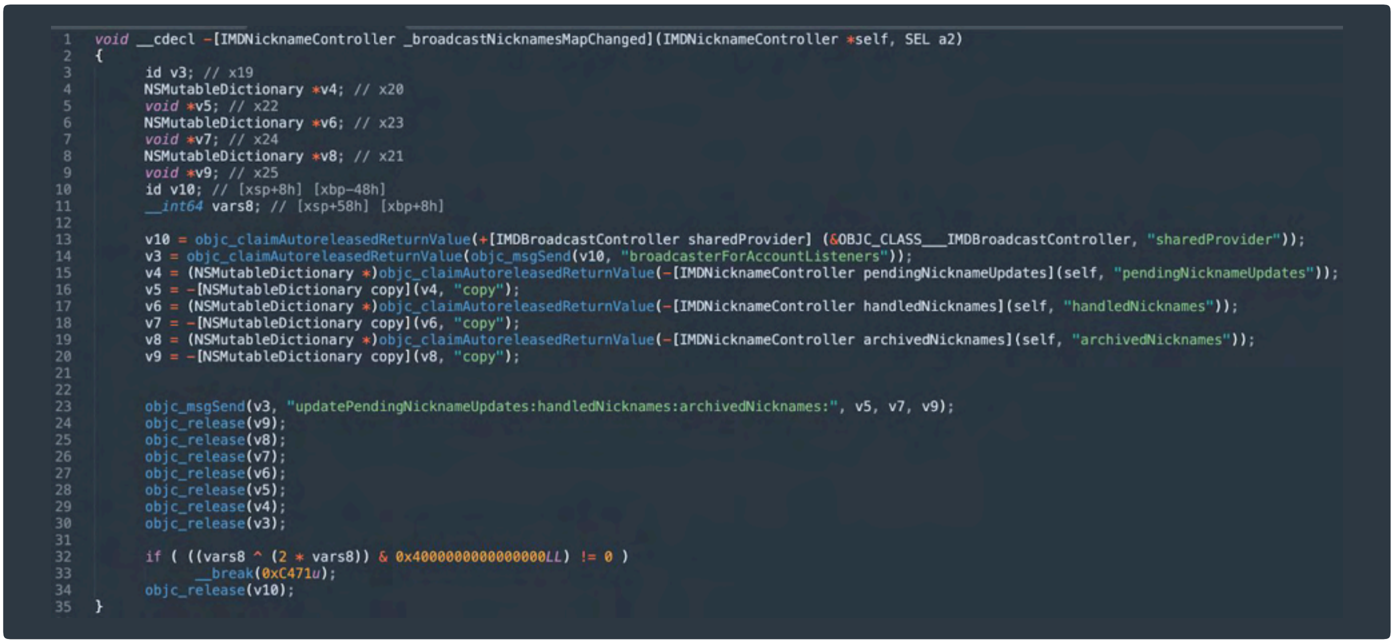
```
void __cdecl -[IMDNicknameController _broadcastNicknamesMapChanged](IMDNicknameController *self, SEL a2)
{
        id v3; // x20
        id v4; // x21
        id v5; // x22
        id v6; // x19
        id v7; // [xsp+8h] [xbp-28h]
        __int64 vars8; // [xsp+38h] [xbp+8h]

        v7 = objc_claimAutoreleasedReturnValue(+[IMDBroadcastController sharedProvider] (&OBJC_CLASS___IMDBroadcastController, "sharedProvider"));
        v3 = objc_claimAutoreleasedReturnValue(objc_msgSend(v7, "broadcasterForAccountListeners"));
        v4 = objc_claimAutoreleasedReturnValue(-[IMDNicknameController pendingNicknameUpdates] (self, "pendingNicknameUpdates"));
        v5 = objc_claimAutoreleasedReturnValue(-[IMDNicknameController handledNicknames](self, "handledNicknames"));
        v6 = objc_claimAutoreleasedReturnValue(-[IMDNicknameController archivedNicknames](self, "archivedNicknames"));

        objc_msgSend(v3, "updatePendingNicknameUpdates:handledNicknames:archivedNicknames:", v4, v5, v6);
        objc_release(v6);
        objc_release(v5);
        objc_release(v4);
        objc_release(v3);

        if ( ((vars8 ^ (2 * vars8)) & 0x4000000000000000LL) != 0 )
            __break(0xC471u);
        objc_release(v7);
}
```

Figure 22: original implementation of the method, pre-patch



```
void __cdecl -[IMDNicknameController _broadcastNicknamesMapChanged](IMDNicknameController *self, SEL a2)
{
        id v3; // x19
        NSMutableDictionary *v4; // x20
        void *v5; // x22
        NSMutableDictionary *v6; // x23
        void *v7; // x24
        NSMutableDictionary *v8; // x21
        void *v9; // x25
        id v10; // [xsp+8h] [xbp-48h]
        __int64 vars8; // [xsp+58h] [xbp+8h]

        v10 = objc_claimAutoreleasedReturnValue(+[IMDBroadcastController sharedProvider] (&OBJC_CLASS___IMDBroadcastController, "sharedProvider"));
        v3 = objc_claimAutoreleasedReturnValue(objc_msgSend(v10, "broadcasterForAccountListeners"));
        v4 = (NSMutableDictionary *)objc_claimAutoreleasedReturnValue(-[IMDNicknameController pendingNicknameUpdates](self, "pendingNicknameUpdates"));
        v5 = -[NSMutableDictionary copy](v4, "copy");
        v6 = (NSMutableDictionary *)objc_claimAutoreleasedReturnValue(-[IMDNicknameController handledNicknames](self, "handledNicknames"));
        v7 = -[NSMutableDictionary copy](v6, "copy");
        v8 = (NSMutableDictionary *)objc_claimAutoreleasedReturnValue(-[IMDNicknameController archivedNicknames](self, "archivedNicknames"));
        v9 = -[NSMutableDictionary copy](v8, "copy");


        objc_msgSend(v3, "updatePendingNicknameUpdates:handledNicknames:archivedNicknames:", v5, v7, v9);
        objc_release(v9);
        objc_release(v8);
        objc_release(v7);
        objc_release(v6);
        objc_release(v5);
        objc_release(v4);
        objc_release(v3);

        if ( ((vars8 ^ (2 * vars8)) & 0x4000000000000000LL) != 0 )
                __break(0xC471u);
        objc_release(v10);
}
```

Figure 23: post-patch, with NSDictionary objects being copied into immutable objects

We can see that a call to `-[NSObject copy]` has been added for each of the three `NSMutableDictionary` variables. The documentation for this method indicates that it "returns the object from `copyWithZone:`."

`copyWithZone:` is further described in the following way:

> The returned object is implicitly retained by the sender, who is responsible for releasing it. **The copy returned is immutable if the consideration "immutable vs. mutable" applies to the receiving object**; otherwise the exact nature of the copy is determined by the class.

The patch diff revealed the `NSMutableDictionary` instance variables as likely candidates for some form of memory corruption. At a high level, the original workflow for broadcasting Nickname Updates in iOS version 18.2 works similar to the following:

- Retrieve a pointer to the underlying data structure (a dictionary).
- Start a new thread.
- Use the thread to conduct remote procedure calls asynchronously to broadcast Nickname Updates to all registered listeners.

This is likely what was happening during the crashes when attempting to serialize an `NSDictionary` for the XPC message: a previous Nickname Update was in the process of being serialized in the `com.apple.Messages.ClientConnection` thread (explored in the section above) while a new Nickname Update was being processed on the main thread. Said another way, one thread was responsible for mutating the dictionary data structure while the other was still operating on its contents, now malformed or otherwise corrupted in some unexpected way.

With this new insight, and in reviewing both of the crash logs presented within this report, we explicitly note that both crashed threads (frame 0) ended at `objc_retain()` while serializing an `NSDictionary` object used as an argument for an `NSXPC` remote invocation. There exists a possible scenario where multiple Nickname Updates, in rapid succession, mutate the targeted dictionary to corrupt memory.

The patch diff shows iOS version 18.3 circumventing this issue by leveraging immutable copies of the underlying data structures. The main thread would continue to make updates to the mutable data structure as needed, depending on when it received Nickname Updates, while other threads would operate on copies of the data.

Characterizing the Imagent Crashes

We queried our telemetry to describe the significance of these `imagent` crashes and whether they could be categorized as normal behavior. Table 1 contains metrics which describe the relevant subsets of data.

Type of ".ips" file	Percentage of ".ips" file
".ips" files related to imagent	0.9%
imagent crashes	0.0042%
Crashes related to Nickname Updates	0.0016%
Crashes indicating Nickname Updates triggered memory corruption	0.0016%

Table 1: description of imagent telemetry data as a % of total ".ips" files collected during the relevant time period (April 2024 - January 2025)

As seen in Table 1, imagent ".ips" files are generally not a rare occurrence, constituting around 1% of all the crash-related logs we collected during the relevant time period of April 2024 to January 2025. The specific crashes related to memory corruption, however, are indeed exceedingly rare – less than .001%. It is also worth noting that the devices on which we saw memory-corruption related crashes only belonged to individuals who are significantly more likely to be targeted by foreign threat actors, all of which described historic, highly credible targeted attacks against their members. At least one unnamed individual within this table specifically reported observing physical surveillance and anomalous device behavior. And these same highly anomalous crashes were observed on at least one device belonging to a senior government official in the European Union approximately thirty days prior to receiving Apple Threat Notifications.

For these findings to be meaningful, however, iVerify's internal telemetry database would need to be sufficiently large and representative of iPhone behavior; otherwise, sample bias becomes a concern. We can think about capturing diversity across several axes: different iOS versions, device models, regions, and user behavior patterns. If we conservatively estimate ~10-20 common variants in each dimension, then getting reasonable coverage would require at least tens of combinations. If we assume partial overlap and some redundancy, a sample size of only ~1000 unique devices should suffice to observe repeated patterns across subgroups and generalize internal behaviors with some confidence. The sample used in this investigation contained closer to 50,000 unique devices, each collecting crash log data relevant to this investigation. And we have no reason to believe the sample is overly biased towards high-risk individuals either, as the overwhelming majority of iVerify's end users constitute rank-and-file individuals, including consumers and employees at various financial institutions, technology companies, non-profit organizations, and government agencies around the world.

Forensic Investigation

The relative rarity of these `imagent` crash files - specifically those triggered by Nickname Updates - prompted a more comprehensive look at available forensic data. The team wanted to identify data artifacts highlighting the malicious nature of these crashes, possibly related to exploitation.

We were able to retrieve a Sysdiagnose and an Encrypted Backup from one of the devices. After reviewing the data, we specifically identified anomalous modification to several SMS Attachments directories approximately 20 seconds after the crash. These directories were later revealed to be empty. This sort of behavior has been observed in the past when threat actors “cleaned up” residual traces of exploitation [1,4,5].

Additionally, we were able to identify two directories related to MessagesMetadata similarly modified and void of any content. These directories generally contain chat information. In other cases, these directories contained a “GroupPhotoImage”. The same directory also contained a folder, denoted simply “NicknameCache”. The “NicknameCache” directory did not contain any data for the date of the crash.

On the same device, we noted a second window of time which marked similar activity, once again occurring after the `imagent` crash was initially observed. For the sake of protecting the identity of the device owner, we have rebased this second window of time to be 80 days after the `imagent` crash.

During the second window of time, we noted similar activity; the SMS Attachment directories once again showed indications of modification, but remained empty. We observed one instance of the “com.apple.CrashReporter.plist” file containing one urgent submission on day 81 after the `imagent` crash.

The use of `mvt` enabled us to review additional records contained within `sms.db` that were modified at day 0, and we found that no existing SMS records were related to the modified directories in question.

TimeStamp rebased to Date of Crash	MVT Module	Event	Details
T+ 00 00:00:00	Manifest	M-CB	DiagnosticReports/imagent-0000-00-00-000000.ips - SysSharedContainerDomain-systemgroup.com.apple.osanalytics
T+ 00 00:00:20	Manifest	M-C-	Library/SMS/Attachments/7e/14 - MediaDomain

T+ 00 00:00:21	Manifest	M-C-	Library/MessagesMetaData/0c/12/iMessage;+;chat-A - HomeDomain
T+ 00 00:00:28	Manifest	M-C-	Library/SMS/Attachments/80/00 - MediaDomain
T+ 00 00:00:29	Manifest	M-C-	Library/SMS/Attachments/6d/13 - MediaDomain
T+ 00 00:00:29	Manifest	M-C-	Library/SMS/Attachments/28/08 - MediaDomain
T+ 00 00:00:29	Manifest	M-C-	Library/SMS/Attachments/90/00 - MediaDomain
T+ 00 00:00:30	Manifest	M-C-	Library/SMS/Attachments/c3/03 - MediaDomain
T+ 00 00:00:33	Manifest	M-C-	Library/SMS/Attachments/c4/04 - MediaDomain
T+ 00 00:00:33	Manifest	M-C-	Library/SMS/Attachments/a2/02 - MediaDomain
T+ 00 00:00:33	Manifest	M-C-	Library/SMS/Attachments/e6/06 - MediaDomain
T+ 00 00:00:34	Manifest	M-C-	Library/SMS/Attachments/a0/00 - MediaDomain
T+ 00 00:00:34	Manifest	M-C-	Library/SMS/Attachments/2a/10 - MediaDomain
T+ 00 00:00:36	Manifest	M-C-	Library/SMS/Attachments/7f/15 - MediaDomain
T+ 00 00:00:36	Manifest	M-C-	Library/SMS/Attachments/29/09 - MediaDomain
T+ 00 00:00:36	Manifest	M-C-	Library/SMS/Attachments/5c/12 - MediaDomain
T+ 00 00:00:38	Manifest	M-C-	Library/SMS/Attachments/6e/14 - MediaDomain
T+ 00 00:00:39	Manifest	M-C-	Library/SMS/Attachments/39/09 - MediaDomain
T+ 00 00:00:39	Manifest	M-C-	Library/MessagesMetaData/84/04/iMessage;+;chat-B - HomeDomain
T+ 80 00:00:00	Manifest	M-C-	Library/SMS/Attachments/ed/13 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/c7/07 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/0d/13 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/87/07 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/60/00 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/f0/00 - MediaDomain
T+ 80 00:00:01	Manifest	M-C-	Library/SMS/Attachments/a7/07 - MediaDomain
T+ 81 00:00:00	CrashReporter		Urgent Submission Count: 1(com.apple.CrashReporter.plist)
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/eb/11 - MediaDomain

T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/cb/11 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/73/03 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/ab/11 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/f9/09 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/9b/11 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/52/02 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/fc/12 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/ca/10 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/96/06 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/ce/14 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/65/05 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/f8/08 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/01/01 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/de/14 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/5a/10 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/50/00 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/18/08 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/ae/14 - MediaDomain
T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/b4/04 - MediaDomain

Table 2: summary of modified directories

We were not able to determine with full certainty if these filesystem metadata artifacts were somehow related to Nickname Updates as facilitated by iMessage. When we attempted to investigate other databases that could have assisted in the analysis, there were no traces of the modified directories. This behavior appeared to us to be relatively unique and highly correlated to the specific type of imagent crash described above, and bears strong similarity to cleanup behavior observed in other confirmed spyware attacks [1,4,5].

T+ 81 00:00:00	Manifest	M-C-	Library/SMS/Attachments/cb/11 - MediaDomain

Discussion & Analytical Assertions

iVerify maintains moderate confidence that these crash logs, as described above, could be related to exploitation attempts against select individuals.

Our analysis reveals various iOS versions crashing as a result of memory corruption and the misuse of mutable `NSDictionary` objects implemented as part of the Nickname Updates subsystem within `imagent`. iVerify maintains high confidence that a Nickame Update, transmitted through iMessage at an irregular frequency, could feasibly mutate Objective-C objects and shape memory into a viable primitive for additional exploitation against core iOS services. Crashes occurred within a specific subset of iOS versions, not exceeding iOS version 18.1.1, and these crashes were derived from various organizations. We identified code changes indicating that the `NSDictionary` objects have effectively been rendered “immutable” and Nickname Updates now operate on copies of the dictionaries, successfully addressing this UAF condition.

iVerify further maintains moderate confidence that these crash logs can only be generated in unique circumstances, e.g. submitting Nickname Updates in rapid succession. This would explain the relative rarity of such an event, which is part of what initially appeared to us as suspicious. Specific test cases are still outstanding, to include the instrumentation of select forms of fuzzing, the results of which would likely boost our confidence. It should be noted that Nickname Updates are not guaranteed to generate any form of UI notifications, and it is likely that a motivated adversary could instrument and productize the rapid transmission of iMessage payloads necessary for exploiting this vulnerability. In our testing, we were successful in transmitting a NicknameUpdate only after a sender and recipient had previously been in contact; how this security mechanism may be bypassed is worth researching.

It could be argued that our findings are a small portion of a larger exploit chain. Assuming the memory corruption described above is viable, to successfully exploit it would likely necessitate prerequisite knowledge derived from other components of the exploit chain: an ASLR defeat, or heap information leak to specifically target the dictionary and mutate it into some form of primitive.

It is altogether possible that these crashes are the byproduct of an entirely separate vulnerability but still related to Nickname Updates. This vulnerability might require the attacker to send frequent messages to the device. An excessive number of messages, if sent too quickly to the intended target, could trigger the specific race condition for the UAF. In this case, the crashes we have identified are unintentionally triggered by the attacker. If we assume an actor was successful in compromising the target device and had cleaned up known indicators of compromise and exploitation residue, then the lingering `imagent` crash logs are possibly indicative of the actor targeting a similar (but distinct) code path that may somehow cause corruption of the mutable dictionaries which handle Nickname Updates. It is worth recalling the structure of a Nickname Update: it is a serialized composition of images, avatars, etc., represented as a dictionary, and likely handles other objects, too. We are not currently positioned to say specifically what requirements must be in place in order to trigger these crashes.

As part of our technical analysis, and in recognizing the significance of these crashes, we conducted forensic investigations on these devices. Unfortunately, the time between notification and data acquisition prevented us from gathering qualitative data that might comprehensively reveal artifacts pointing to other components of an exploit chain; however, forensic examination of one device did reveal suspicious behavior closely attributed to iMessage exploitation. On this device, we observed indications that iMessage attachments, directories, and metadata were modified and no longer contained legitimate attachments. The modification timestamps were temporally significant, occurring within 20 seconds of the original `imagent` crash we analyzed. We identified similar filesystem metadata modifications related to NicknameUpdate attachment directories that were modified and altogether empty. While it's possible this behavior was caused by the user deleting messages in bulk, we have also observed this behavior associated with past exploitation of the iMessage remote attack surface. The circumstances under which the attachments were deleted – exactly 20 seconds after a highly anomalous crash, on two occasions, well outside of normal business hours – make it difficult to rule out the possibility the behavior may be residue from clean-up behavior after a malicious implant successfully received execution on the device.

More data is needed to comprehensively describe the threat at hand. There is, however, a body of telemetry that cannot be ignored:

These crashes are exceedingly rare considering the scale of our telemetry;
The crashes were found only on devices associated with organizations with a heightened risk profile and evidence of previous attacks by advanced actors;
This bug occurs in `imagent`, a core service that is expected to operate with higher standards of integrity and security, and which has been been exploited in the past;
At least one device on which we observed these crashes also received an Apple Threat Notification after the crash occurred; and
There is some evidence of potential clean-up activity from one of the devices we forensically analyzed.

In conclusion, we have moderate confidence that this was an attempt to exploit this subset of devices. We have moderate confidence that at least one device was successfully compromised. And we were unable to verify whether Lockdown Mode impacts the functionality of NICKNAME in one way or another.

Scope & Incidence

All crash logs were generated by devices with iOS versions between 17.2.1 to 18.1.1, inclusive.

References

1. https://iverify.io/blog/clipping-wings-our-analysis-of-a-pegasus-spyware-sample
2. https://googleprojectzero.blogspot.com/2025/03/blasting-past-webp.html
3. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html
4. https://citizenlab.ca/2023/04/nso-groups-pegasus-spyware-returns-in-2022/
5. https://securelist.com/operation-triangulation/109842/
6. https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/
7. https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-how-to-catch-nso-groups-pegasus/
8. https://www.amnesty.org/en/latest/research/2021/07/forensic-methodology-report-appendix-d/
9. https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
10. https://developer.apple.com/documentation/foundation/nsinvocation
11. https://developer.apple.com/documentation/foundation/nsxpcconnection
12. https://developer.apple.com/documentation/foundation/nscopying/1410311-copywithzone
13. https://developer.apple.com/documentation/xcode/sigbus
14. https://developer.apple.com/documentation/xcode/acquiring-crash-reports-and-diagnostic-logs
15. https://libimobiledevice.org/
16. https://github.com/blacktop/ipsw

iVerify.