Tales From The Crash Mines Issue #1

Landon Fuller

February 6, 2014

Introduction

In our work on PLCrashReporter, we get to see a variety of interesting crashes in applications and libraries. One of the support services we offer is helping application developers track down truly insidious bugs; we'll often perform deep source and instruction-level analysis of crash reports, the customer's application code, and Apple's system frameworks in the process.

This is the first issue of "Tales From The Crash Mines" – what we intend to be an intermittent series on the *interesting* bugs we've tracked down on iOS and Mac OS X. To us, interesting bugs are ones that reveal themselves in non-obvious or novel ways, and that developers are likely to see in their own crash reports. We hope that we can provide some concrete guidance as to how to avoid these issues in your own code base, as well as how to recognize when you're seeing a similar issue in your crash reports, and maybe some insight into how we analyze failures.

For our first bug, we've got a fun crasher from our friends at Unibox, who were gracious enough to allow us to use their issue as an example in our series. Unibox is a Mac OS X application, but our analysis applies equally to iOS – we'll be sure to explain any minor differences as we go.

If you're looking for a fantastic e-mail client, be sure to give Unibox a try.

Ladies and gentlemen: the story you are about to hear is true. Only the code has been changed to protect Unibox's intellectual property rights. Any issues you find in the example source code listings, class and method names, and crash reports are entirely *our* fault, because we wrote all of the examples.

A Butterfly Flaps Its Wings

I'm sure everyone is familiar with the pop-sci definition of chaos theory – a butterfly flaps its wings in Brazil, and sets off a tornado in Texas.

The idea isn't that the butterfly powers the tornado, but rather, that a tiny change

in the initial conditions of a deterministic nonlinear system can result in wildly different large-scale outcomes.

The execution of software is *nowhere near* the complexity of weather systems (a good thing for us!), but the basic idea still applies; there can exist a significant gap between the bug that triggers a failure, and the actual visible outcome of that failure. What's more, the actual *type* of visible failure (and whether a failure even occurs at all) is highly dependent on the initial conditions –including what threads were running, how far had they progressed, when did a network server respond, and the initial data with which the code was initialized.

This was the case with Unibox's bug, and one of the reasons I found it interesting. The folks at Unibox simply could not reproduce the bug locally. It only happened semi-regularly. The actual point at which the software crashed varied, and it rarely corresponded to an obvious line of source code. The *types* of crashes varied. The only commonalities between the crash reports were:

- The crashes occured in the same area of the networking code.
- The failures all pointed to an object over-release, despite the use of ARC.

Given a bug that can't be reproduced locally, crashes at different places, in different ways, and may not even refer to any of your own code – how do you debug it?

This is where we turn back to our pop-sci chaos theory. Given an observed *outcome* (crash reports), and a *deterministic system* (the processor and execution environment), we need to figure out what initial conditions could lead to all the failures we saw.

To do so, our usual approach at Plausible Labs is to load up the binary in a disassembler and work backwards from the crash, determining the full set of initial conditions that could result in the failure. This is a simple process of elimination, where we discard possible initial conditions by proving that they could *not* have triggered the crash.

In most cases, enough data exists in the collected crash reports to narrow the possibilities to a single set of initial conditions that could cause it – and thus, the bug –without ever needing to actually run the code or reproduce the issue.

If we can't narrow it down to a single bug, we instead perform more directed data gathering to help eliminate whatever possibilities remain. The more data we gather, the more we can narrow the number of initial conditions until we find the root cause.

This approach is the fastest way I've found to triage crash reports – and sometimes the only way – as it requires no actual reproduction case, just a crash report. Think of it as a brain-powered git bisect, leveraging your long-term memory and pattern recognition to quickly rule out what initial conditions you can, and then evaluating what's left over.

I also want to take a moment to emphasize the following point: for most bugs, you *don't need to be able to read assembly* to perform this type of crash analysis. While being

able to perform static analysis at the assembly level makes it easier to more accurately eliminate impossible conditions, it's often possible to reach the same conclusions by evaluating everything at the source code level. Sometimes you might need a bit more data, but this still remains the best approach I know of to quickly reason about failure.

Now, with our pop-sci chaos theory in hand, let's take a look at one of the crash reports that the Unibox team provided, and determine what possible initial states could result in the exhibited behavior. There are some fun twists and turns ahead.

The Crash Report – SIGBUS at 0x0

For the purposes of this article, we'll be walking through a complete step-by-step root cause analysis of a single crash report – a `SIGBUS` signal triggered in `objc_autorelease()`, with a faulting address of `0x0`:

Unibox Crash Report

```
...
Exception Type:  SIGBUS
Exception Codes: BUS_ADRERR at 0x0
...
Thread 17 Crashed:
0   libobjc.A.dylib                     0x00007fff896392d2
    objc_autorelease + 18
1   ExampleApp                          0x0000000103fbf0f9
    -[EXNetConnection execute:timeout:completionBlock:]
    (EXNetConnection.m:89)
2   ExampleApp                          0x0000000103fbeff7
    -[EXNetConnection execute:completionBlock:] (EXNetConnection.m:77)
3   ExampleApp                          0x0000000103fbfba7
    -[EXNetConnection selectItem:] (EXNetConnection.m:163)
...
Thread 17 crashed with X86-64 Thread State:
    rip: 0x00007fff896392d2    rbp: 0x0000000115846590    rsp:
        0x0000000115846568    rax: 0xbadd30ac3ceabead
    rbx: 0x0000600002a2ef20    rcx: 0x0000000000000001    rdx:
        0x0000000115846468    rdi: 0x0000608001c4b0a0
    rsi: 0x0000000000001a7a     r8: 0x0000000000001fff     r9:
        0xffff9fffffb0267f    r10: 0x000000010f8e47a0
    r11: 0x0000000000000201    r12: 0x00007fff89622080    r13:
        0x000060000263d040    r14: 0x0000000000000000
    r15: 0x0000608002c692c0 rflags: 0x0000000000010246     cs:
        0x000000000000002b     fs: 0x0000000000000000
    gs: 0x0000000011190000
```

Initial Hypothesis

At a glance, we can make the following hypothesis about the initial state of the process at the time `objc_autorelease()` was called:

- A SIGBUS with an address of 0x0 *probably* means we dereferenced a NULL pointer (see Exploring iOS Crash Reports).
- The `id objc_autorelease (id obj)` function is equivalent to calling `[obj autorelease]`, and like `-autorelease`, it ignores nil object values.[1] If we're crashing due to a NULL pointer in `objc_autorelease`, it's *probably* because the object's pointer points to real memory, but the data is not actually a valid object.

This gives us a place to start digging — a `frame 0` hypothesis.

Examining Frame 0: Crash in objc_autorelease()

```
Thread 17 Crashed:
0   libobjc.A.dylib    0x7fff896392d2 objc_autorelease + 18
...
```

We think we know what the crash was — a NULL dereference in `objc_autorelease()` — but we haven't proven anything. To actually prove that the crash *was* a NULL dereference, we need to examine the implementation of `objc_autorelease()`, eliminating any impossible initial states until we're left with (ideally) one state that could be valid – such as our hypothesized NULL dereference.

We'll start by examining the implementation of `objc_autorelease()` from Apple's published objc4-551.1 runtime sources. In reviewing the source code below, be aware that Objective-C objects are implemented as C++ structs in the modern Objective-C runtime; statements such as `obj->isTaggedPointer()` are C++ method calls, not calls to C function pointers:

[1] The `id objc_autorelease (id obj)` function was added with the introduction to ARC. Refer to the clang documentation for details.

```
id objc_autorelease (id obj) {
    if (!obj || obj->isTaggedPointer())
        goto out_slow;

    if (((Class)obj->isa)->hasCustomRR())
        return [obj autorelease];

    return bypass_msgSend_autorelease(obj);

 out_slow:
    return obj;
}
```

It appears that `objc_autorelease()` will only crash dereferencing a NULL pointer if `obj->isa` is NULL; if instead the `obj` argument is NULL, the function immediately terminates, and `obj->isa` looks like the only other pointer dereferenced in the function.

However, "likely" isn't good enough. To provide proof, we need to look at the actual `objc_autorelease()` implementation, and specifically, the actual crashing instruction, at `0x7fff896392d2 objc_autorelease + 18`.

Dissassembling Frame 0

The x86-64 implementation of `objc_autorelease` follows; we'll step through the assembly listing in detail, so don't worry if your x86-64 is rusty.[2]

```
; if (!obj || obj->isTaggedPointer())
;     goto out_slow;
0x7fff896392c0    test    rdi, rdi
0x7fff896392c3    je      loc_out_slow
0x7fff896392c5    test    dil, 1
0x7fff896392c9    je      do_autorelease
; out_slow:
;    return obj;
0x7fff896392cb loc_out_slow:
0x7fff896392cb    mov     rax, rdi
0x7fff896392ce    ret
0x7fff896392cf do_autorelease:
...
```

The first pair of instructions test whether the `obj` argument is equal to nil, and if

[2]To get started with assembly-level analysis, I suggest Mike Ash's articles on Object File Tools and The Hopper Disassembler, as well as Gwynne Raskind's Disassembling the Assembly, Parts 1, 2, and 3.

so, we jump to `loc_out_slow`.[3]

The second pair of instructions test whether the `obj` argument is a tagged pointer[4]. This is the inlined implementation of `obj->isTaggedPointer`. If `obj` **is not** a tagged pointer, we jump to `do_autorelease` below. Otherwise, execution continues at the next instruction (which just so happens to be `loc_out_slow`). If you're wondering about the odd `dil` register, on x86-64, the `dil` register is just an alias for the lower 8 bits of the `rdi` register.

Lastly, `loc_out_slow` copies the `obj` argument (`rdi`) into the return address register (`rax`), and returns it to the caller. If we arrived here from the nil or a tagged pointer test, the original `obj` argument value will be directly returned to the caller, and `objc_autorelease()` will terminate. Since our crash occurred later in the function, we've confirmed the `obj` argument was neither nil, nor a tagged pointer.

At this point, nothing has dereferenced the `obj` pointer, much less `obj`, and we haven't yet reached the instruction that crashed. Let's move on.

```
; if (((Class)obj->isa)->hasCustomRR())
0x7fff896392cf    mov     rax, [rdi]
0x7fff896392d2    mov     rax, [rax+32] ; <-- We crashed loading 8 ↙
bytes from rax+32
... [elided remainder of function]
```

Here we arrive at the meat. The first mov instruction fetches the `obj->isa` pointer[5], successfully storing the value in the `rax` register.

The second `mov` instruction is the inlined implementation of `hasCustomRR`, and is where we actually crashed attempting to load 8 bytes from `rax+32`.

From our reading of the code, it's clear the `obj` argument to `objc_autorelease()` *must* have pointed to mapped, readable memory; otherwise, we never would have been able to read `obj->isa`. The actual crash occurred one instruction later, while dereferencing `isa->data` (ie, `[rax+32]`) at `0x7fff896392d2`.

Given this proven order of events, we now know the memory pointed to by `obj` *was* mapped and readable, but *did not* contain a valid `isa` pointer at the time `objc_autorelease()` was called — or it became invalid during execution of the function — as the crash occurred in attempting to read 8 bytes from `isa->data`.

Thus, we've proven *part* of our hypothesis:

[3] On Mac OS X x86-64 the `rdi` register is used for the first function argument. For more details, refer to Apple's Mac OS X and iOS calling convention documentation.

[4] Tagged pointers are covered in depth by Bavarious's Tagged Pointers in Lion and Mike Ash's Lets Build Tagged Pointers

[5] See Intro to the Objective-C Runtime for an explanation of the Objective-C `isa` pointer

- The object passed to `objc_autorelease()` was a non-NULL pointer to mapped memory; otherwise, execution would have never reached the point at which it crashed.
- At the time of the crash, the object pointer *did not point to a valid Objective-C object*, as `objc_autorelease()` crashed dereferencing a corrupt `isa` pointer value within the object.

This leaves us with one remaining unproven item from our original hypothesis: that the dereferenced `isa` *was* NULL, as was implied by `Exception Codes: BUS_ADRERR at 0x0` in our crash report.

There's just one problem; take a look at the crashing instruction again:

```
0x7fff896392d2    mov     rax, [rax+32] ; <-- We crashed loading 8 ↙
bytes from rax+32
```

According to our crash report, we received a `SIGBUS`, with a faulting memory address of `0x0`. There's *no way* the faulting address was `0x0`, as the code *always* dereferences the value of `rax`, **plus 32**. Compounding matters, if we look at the value of `rax` as reported in the crash report, it's nowhere *near* `0x0`:

```
rax: 0xbadd30ac3ceabead
```

What gives? Did the crash reporter generate an invalid crash report?

Invalid Crash Report? No, just an invalid assumption

The answer is no – the crash report is fine. The `si_addr` value reported by the kernel *was* 0x0, the actual faulting address *was not* 0x0, and there's a good reason for all of that: x86-64 canonical form addresses. On all current implementations of the x86-64 instruction set, only 48-bits of the full 64-bit address range is available for use. Additionally, the AMD64 and Intel specifications require that addresses be expressed in canonical form: bits 48-63 of an address must be copies of bit 47. If bit 47 is 0, so must be bits 48-63, and vise versa. If you're curious about the details, refer to section 3.3.7.1 of Intel's 64 and IA-32 Architectures Software Developer's Manual: Volume 1.

Take another look at `rax`'s value. `0xbadd30ac3ceabead` is *not* in canonical form. A memory reference to a non-canonical address triggers a general protection fault, and on x86, a general protection fault *does not supply the faulting address*. If you want to know the actual faulting address, you have to examine register state and the machine

code to determine what the software was attempting to load — which is exactly what we just did, manually. The kernel *could* try to do this automatically to provide a valid `si_addr` , but it doesn't, and that's something POSIX explicitly permits::

> For some implementations, the value of si_addr may be inaccurate.

Performing these sorts of advanced instruction-level heuristics automatically is on the PLCrashReporter roadmap, but in the mean time, how would you even *find* the answer for a discrepancy like this if you weren't already familiar with kernel-level x86-64 fault handling?

Digging Deeper

If you find something confounding in an area you're unfamiliar – like a discrepancy between what the crash report contains, and what the code says – then start digging to figure out how it could possibly happen. It's generally a good idea to work downwards from high-level to low-level, first by validating your own assumptions, and then the assumptions of the implementation you rely on, and so on down the technology stack until you find your answer:

1. Consult the system documentation, such as the man pages for `signal(3)` and `sigaction(2)` , to determine what behavioral guarantees are actually made by the system. Your assumptions regarding a specific behavior may not match what is actually documented.
2. Review the relevant standards, such as the POSIX specification, to determine what behavioral guarantees are expected to exist, and to help resolve any ambiguities in the system documentation.
3. Review the actual implementation to determine how those behavioral guarantees are met. In a case like this, books such as Amit Singh's http://osxbook.com/ provide a great starting point. Apple continues to provide the source code for the Mac OS X kernel at `http://www.opensource.apple.com/`, and the Mac OS X implementation tends to match the iOS kernel.
4. Consult the architecture reference manuals for the underlying platform(s) – freely available from both ARM and Intel – to confirm the behavior that you're seeing exhibited.

In this case, I actually resolved this issue exactly as described above. I was already familiar with the canonical addressing rules, but I didn't make the connection until I worked my way through the full stack and arrived at the Intel architecture manual.

Lastly, if you've dug deeply to no avail, *ask*. Seriously. Resources like the PLCrashReporter mailing list and IRC channel are here to help.

Refining our Hypothesis

We've proven two of the initial statements in our hypothesis, and we've discarded the third - that the isa was NULL, triggering a NULL dereference. That's OK — it wasn't imperative to our analysis, and we now know for certain:

- The object pointer passed to `objc_autorelease()` was a valid pointer to readable memory.
- At the time of the crash, the object pointer did not point to a valid Objective-C object.

Now that we know what the `objc_autorelease()` failure *was* — it was passed a pointer to mapped memory that *wasn't* an object — let's figure out *how that happened*.

Examining Frame 1

```
Thread 17 Crashed:
0   libobjc.A.dylib     0x7fff896392d2 objc_autorelease + 18
1   ExampleApp          0x000103fbf0f9 -[EXNetConnection ↙
execute:timeout:completionBlock:] (EXNetConnection.m:89)
...
```

We've established that a bad object pointer was passed to `objc_autorelease()`. Since our backtrace shows that the next frame is `-[EXNetConnection execute:timeout:completionBlock:]`, it's *probably* the function that called `objc_autorelease()` with an invalid argument. We say "probably" quite intentionally: A crash report isn't a time machine; it's a snapshot of the process as it was when the crash occurred, from which the backtrace is reconstructed. We can't take any unproven inferences at face value.

By now, some readers have likely thought of running the code with `NSZombie` enabled to track down the bad object; remember that there is no reliable reproduction case, and enabling `NSZombie` was tried long before the bug landed on our desks.

Instead, let's take a look at the `-[EXNetConnection execute:timeout:completionBlock:]` implementation and see if anything stands out:

```
- (EXResponse *) execute: (NSString *) command timeout: ↙
(NSTimeInterval) timeout completionBlock: (EXResponseBlock)
{
      return [self execute: command timeout: timeout completionBlock: ↙
      block updateCallback: nil]; // <-- Stack trace claims that we ↙
      called objc_autorelease() here, at line #89
}
```

Unfortunately, this is too ambiguous to be useful — the call to `objc_autorelease()` was inserted by ARC, but we don't know where, or even what object ARC was attempting to autorelease. We could guess — knowing what we do about ARC, it almost certainly inserted autorelease code for the return value. However, we can't prove anything with this level of ambiguity, and you might be surprised what we find out when we start digging. Let's look at the actual assembly code for `-[EXNetConnection execute:timeout:completionBlock]`:

```
0x103fbf0ba -[EXNetConnection execute:timeout:completionBlock:] proc ↙
near
0x103fbf0ba var_20          = qword ptr -20h
0x103fbf0ba    push    rbp
0x103fbf0bb    mov     rbp, rsp
0x103fbf0be    push    r15
0x103fbf0c0    push    r14
0x103fbf0c2    push    rbx
0x103fbf0c3    push    rax
0x103fbf0c4    mov     r14, rcx
0x103fbf0c7    movsd   [rbp+var_20], xmm0
0x103fbf0cc    mov     r15, rdi
0x103fbf0cf    mov     rdi, rdx
0x103fbf0d2    call    cs:_objc_retain_ptr
0x103fbf0d8    mov     rbx, rax
0x103fbf0db    mov     rsi, ↙
cs:selRef_execute_timeout_completionBlock_updateCallback_
0x103fbf0e2    mov     rdi, r15
0x103fbf0e5    mov     rdx, rbx
0x103fbf0e8    movsd   xmm0, [rbp+var_20]
0x103fbf0ed    mov     rcx, r14
0x103fbf0f0    xor     r8d, r8d
; This is the instruction that supposedly called objc_autorelease(),
; but it's actually a call to [self ↙
execute:timeout:completionBlock:updateCallback:]
0x103fbf0f3    call    cs:_objc_msgSend_ptr
; The return address in our crash report
0x103fbf0f9    mov     r14, rax
0x103fbf0fc    mov     rdi, rbx
0x103fbf0ff    call    cs:_objc_release_ptr
0x103fbf105    mov     rdi, r14
0x103fbf108    call    _objc_retainAutoreleasedReturnValue
0x103fbf10d    mov     rdi, rax
0x103fbf110    add     rsp, 8
0x103fbf114    pop     rbx
0x103fbf115    pop     r14
0x103fbf117    pop     r15
0x103fbf119    pop     rbp
0x103fbf11a    jmp     _objc_autoreleaseReturnValue
0x103fbf11a -[EXNetConnection execute:timeout:completionBlock:] endp
```

There's no call to `objc_autorelease()` made at address `0x103fbf0f3`, or anywhere else in the method! Instead, the address listed in the crash report corresponds to a call to `[self execute:timeout:completionBlock:updateCallback:]` (via `objc_msgSend`). What's happening here?

To find out, let's take a look at the assembly for the method that *is* being called, `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`. Specifically, we want to look at the function epilogue ARC emitted:

```
...[snipped everything but the final two instructions]...
0x103FBF3A3     pop     rbp
; Tail call to objc_autoreleaseReturnValue()
0x103FBF3A4     jmp     _objc_autoreleaseReturnValue
```

Here's our answer. When `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` returns to it's caller, it does so via what's known as a *tail call* to `objc_autoreleaseReturnValue()`

A tail call is a function (or method) call that is performed as a final action within a function; the function pops its stack frame (or never allocates one), and then branches directly to another function. The tail-calling function's stack frame is *gone* – there is nothing for a crash reporter to include in the backtrace. This use of tail calls is an optimization strategy; by letting `objc_autoreleaseReturnValue` return directly upon completion, we eliminate additional stack cleanup that would otherwise be performed by every ARC-generated function returning an autoreleased value.

By investigating the assembly in question, we were able to determine the actual calling path:

- The only way for `objc_autorelease()` to appear in the backtrace at that location was via a tail-call issued by `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` .
- The only tail call issued by `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` is to `objc_autorelease()` (via `objc_autoreleaseReturnValue()`).
- `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` passes its return value to ‘ `objc_autoreleaseReturnValue()` .

This is why we couldn't reason accurately about this bug from just the source code or the crash report; there were two intermediate calls invisible in the backtrace:

```
Thread 17 Crashed:
0   libobjc.A.dylib     0x7fff896392d2 objc_autorelease + 18
*   libobjc.A.dylib     0x7fff896252a9 objc_autoreleaseReturnValue + 47
*   ExampleApp          0x000103fbf11a -[EXNetConnection ↙
execute:timeout:completionBlock:updateCallback:] (NetConnection.m:106)
1   ExampleApp          0x000103fbf0f9 -[EXNetConnection ↙
execute:timeout:completionBlock:] (NetConnection.m:89)
...
```

We this information information in hand, we now know who called `objc_autorelease()`, and with what value. We can add this item to our list of proven initial conditions:

- **New**: The invalid argument passed to `objc_autorelease()` was `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`'s return value.
- The argument passed to `objc_autorelease()` was a valid pointer to mapped memory.
- At the time of the crash, the argument passed to `objc_autorelease()` did not point to a valid Objective-C object.

We're nearly to the end now.

Examining Frame 1.1

```
Thread 17 Crashed:
0   libobjc.A.dylib    0x7fff896392d2 objc_autorelease + 18
*   libobjc.A.dylib    0x7fff896252a9 objc_autoreleaseReturnValue + 47
*   ExampleApp         0x000103fbf11a -[EXNetConnection ↙
execute:timeout:completionBlock:updateCallback:] (NetConnection.m:106)
1   ExampleApp         0x000103fbf0f9 -[EXNetConnection ↙
execute:timeout:completionBlock:] (NetConnection.m:89)
...
```

Now that we know `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` returned an invalid object, causing the crash in `objc_autorelease()`, the only thing left is to figure out *why* the object was invalid.

Let's try taking a look at the Objective-C implementation of `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`:

```
- (EXResponse *) execute: (NSString *) command timeout: ↙
(NSTimeInterval) timeout completionBlock: (EXResponseBlock) block ↙
updateCallback: (EXUpdateBlock) updateCallback
{
    EXRequest *request = [[EXRequest alloc] initWithCommand: command ↙
    timeout: timeout completionBlock: block updateCallback: ↙
    updateCallback];
    EXRequestHandler *requestHandler = [[EXRequestHandler alloc] ↙
    initWithRequest: request];
    [self.requestExecutor executeRequest: requestHandler]; // ↙
    Executes synchronously, on a background thread.
    return requestHandler.response;
}
```

Now we're cooking with gas.

We already know that our crash is caused by an invalid return value, and `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` returns in only one place, at the end of the method:

```
return requestHandler.response;
```

This code fetches the `requestHandler.response` property value and returns it.

ARC rules guarantee that the `requestHandler` instance itself is valid: we created it, we hold a live reference to it, and it *should* remain valid. For this code to fail in the way we're seeing, there exist two likely possibilities:

A) The `EXResponse` property value returned by `requestHandler.response` was invalid at the time it was accessed.

B) The `EXResponse` value returned by `requestHandler.response` somehow concurrently *became* invalid after it was fetched, but before ARC passed it to `objc_autoreleaseRetu` `()` on return.

Let's take a look at the assembly for `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` and try to narrow down this list to one.

Disassembling Frame 1

The x86-64 implementation of `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` follows; once again, we'll step through the assembly listing in

detail. We'll also skip unrelated instructions, focusing only on the relevant lifetime of the `requestHandler` and `requestHandler.response` instances:

```
; return requestHandler.response
0x103FBF36D    mov     rsi, cs:selRef_response
0x103FBF374    mov     rdi, r13 ; requestHandler handler instance
0x103FBF377    call    r12 ; _objc_msgSend
0x103FBF37A    mov     r14, rax
...
0x103FBF383    mov     rdi, r14
0x103FBF386    call    _objc_retainAutoreleasedReturnValue
```

The first four instructions fetch the `requestHandler.response` property via `objc_msgSend(requestHandler, @selector(response))` and store the returned `EXResponse *` result in the `r14` register. The fact that this code executed successfully implies that `requestHandler` was valid at the time it was executed.

The last two instructions move the returned `response` value into the first argument register (`rdi`), and then issue a call to `objc_retainAutoreleasedReturnValue(response)` is called. This call will either retain the `response` value, or will skip the retain if `requestHandler.response` transferred its ownership to our method[6] – determining which one of these occurred may allow us to establish *when* the EXResponse value became invalid.

```
; return requestHandler.response
...
0x103FBF383    mov     rdi, r14 ; response is in r14
0x103FBF3A4    jmp     _objc_autoreleaseReturnValue
```

The `jmp` instruction occurs at the very end of the `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` method; this is the tail-call to `objc_autoreleaseReturnValue(response)` that crashed.

The `response` property is fetched, passed to `objc_retainAutoreleasedReturnValue()` , and then finally returned via a call to `objc_autoreleaseReturnValue()` .

[6] ARC applies runtime heuristics on the *caller's* instructions to determine whether it actually needs to autorelease a value prior to return, or instead, can simply hand the current reference to the caller. This is done as an optimization, and if you're interested in why and and how, I suggest starting with Jonathan Rentzsch's ARC's Fast Autorelease, and Mike's Friday Q&A on Automatic Reference Counting.

The Response Object's Lifetime

It is notable that the call to `objc_retainAutoreleasedReturnValue()` *did not crash*, but the later call to `objc_autoreleaseReturnValue` *did*. We determined during our analysis of `frame 0` that the crash was caused by an invalid `isa` pointer, and we can use this information to narrow our current set of hypotheses.

If `objc_retainAutoreleasedReturnValue()` accessed the `response` object's `isa` and did not crash, then we know that the `response` object became invalid concurrently to the execution of the method, *after* it was passed to `objc_retainAutoreleasedReturnV()`, but *before* it was handed to `objc_autoreleaseReturnValue()`. If we can prove this occurred, we'll know that we're *probably* dealing with a threading related race condition.

Let's take a look at the implementation of `objc_retainAutoreleasedReturnValue()` from Apple's published objc4-551.1 runtime sources, and see if we can determine whether it dereferenced the object's `isa`:

```
id objc_retainAutoreleasedReturnValue (id obj) {
#if SUPPORT_RETURN_AUTORELEASE
    if (obj == tls_get_direct(AUTORELEASE_POOL_RECLAIM_KEY)) {
        tls_set_direct(AUTORELEASE_POOL_RECLAIM_KEY, 0);
        return obj;
    }
#endif
    return objc_retain(obj);
}
```

This code is short, but a little tricky. On x86-64 and ARM, where `SUPPORT_RETURN_AUTORELEASE` is enabled, ARC uses thread-local storage and runtime introspection of return addresses to elide calls to `objc_retain()` and `objc_autorelease()`, if ARC can confirm that both the callee and caller support this behavior.[6] This is an optimization strategy that allows code compiled with ARC to return ownership of an object directly to its caller *without* the overhead of using the autorelease pool.

On the caller-side, thread-local storage is used to determine whether the callee returned direct ownership of the object. This is what you're seeing in the implementation of `objc_retainAutoreleasedReturnValue()` – if the per-thread `AUTORELEASE_POOL_RECLAIM_KEY` value is set, and is equal to the returned object's pointer value, then the caller already has a reference, and `objc_retain()` won't be called.

If `AUTORELEASE_POOL_RECLAIM_KEY` was *not* set by the `requestHandler.response` property getter, we'll know that `objc_retain(obj)` was called. If we review the implementation of `objc_retain()`, we can see that it *does* dereference `objc->isa` when passed a non-NULL, non-tagged pointer, as is the case for our

`response` value:

```
id objc_retain(id obj) {
    if (!obj || obj->isTaggedPointer()) {
        goto out_slow;
    }
    if (((Class)obj->isa)->hasCustomRR()) {
        ...
    }
    ...
    return obj;
}
```

Thus, if we can prove that `objc_retain()` was called, we'll have proven that the memory pointed to by the `response` return value *was* a valid object at the time of the call, as otherwise the call would have crashed. If that is true, then we'll have narrowed our two hypotheses down to a single provable initial condition: that the `EXResponse` value returned by `requestHandler.response` somehow concurrently *became* invalid after it was fetched, but before ARC passed it to `objc_autoreleaseReturnValue()` on return.

To prove this, we need to know whether `AUTORELEASE_POOL_RECLAIM_KEY` was set by the `requestHandler.response` property getter. If it was *not* set, then `objc_retainAutoreleasedReturnValue()` successfully called `objc_retain()` on the `response` object. The easiest way to determine this is to look at the the `EXRequestHandler.request` property getter implementation:

```
0x103FC3DF2 -[EXRequestHandler response] proc
; Standard function prologue
; Set up the stack frame
0x103FC3DF2    push    rbp
0x103FC3DF3    mov     rbp, rsp

; Fetch the offset to the response instance variable
0x103FC3DF6    mov     rax, cs:_OBJC_IVAR_$_EXRequestHandler_response

; Load the response instance variable's pointer value
; from self+ivaroffset, and place the result
; in the return address register.
0x103FC3DFD    mov     rax, [rdi+rax]

; Standard function epilogue
; Restore the frame pointer and return
0x103FC3E01    pop     rbp
0x103FC3E02    retn
```

There is no code that sets a thread-local value, and there are no calls out to any other functions that could – therefore, we know for certain that `objc_retainAutoreleasedReturnValue()` successfully called `objc_retain()` on the `response`, and that the `response` value became invalid *after* it was returned by the property getter.

We've also discovered something very interesting: `-[EXRequestHandler response]` directly returns a borrowed reference to its backing instance variable. This is the standard behavior for synthesized `non-atomic` properties, and indeed, if we look at the property declaration, that's exactly what we find:

```
@property (nonatomic, readonly, strong) EXResponse *response;
```

If the `response` property was concurrently set on another thread, the underlying value could be deallocated *prior* to our call to `objc_retain()`. While we've proven that the `response->isa` pointer pointed at readable memory at the time `objc_retain()` was called, we haven't proven that the object was actually alive; if we call `objc_retain()` on a deallocated object, the behavior is undefined, and *not crashing* is just one possible undefined behavior.

This means we now have *two* new items to add to our list of proven initial conditions:

- **New**: The `EXResponse` value returned by `requestHandler.response` was returned as a borrowed reference, and without external locking, a race condition exists between fetching and setting the value.
- **New**: The `requestHandler.response` *became* invalid after it was fetched, but before it was returned by `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`.
- The invalid argument passed to `objc_autorelease()` was `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`'s return value.
- The argument passed to `objc_autorelease()` was a valid pointer to mapped memory.
- At the time of the crash, the argument passed to `objc_autorelease()` did not point to a valid Objective-C object.

We also have a new – and likely final – hypothesis: If no external locking is performed around setting of the `requestHandler.response` value, and the value is set concurrently to being read by `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]`, the `response` object may be deallocated out from under the method, and ultimately trigger a crash in `objc_autorelease()`. Compared to the rest of our analysis, proving this final one is easy.

A quick project text search returned two methods that set the response property, without any locking: one set the property when a server response had been returned, and the other set the property if a connection error occurred. When a valid response was returned, the code in `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]` immediately attempted to read the `response` property.

If it so happened that the server returned a valid response, and then immediately thereafter a connection error occurred, the property would be set twice in succession, resulting in the race condition we hypothesized, with the `response` value being deallocated out from under our `-[EXNetConnection execute:timeout:completionBlock:updateCallback:]]` .

The fix is simple enough, and Unibox has already rolled it out in a beta release. The race condition can be prevented by synchronizing *updates* to the `response` property – if a response has already been set, then a connection error should not modify the property value, and there's no risk of an unexpected deallocation.

Objective-C also supports the `atomic` property flag, which provides atomic get/set semantics: you will always receive a valid reference to the underlying property, even if it's concurrently being set by another thread. While use of an atomic property would have prevented the crash, it would have also masked the underlying bug – having received a valid server response, the connection failure was unimportant, and should not have overwritten the `response` property.

Conclusion

This has been a deep dive, and I hope that we've presented some useful methodologies that you can use to analyze complex or difficult-to-reproduce issues in your own code. Even if you're not fluent in assembly, leveraging this deductive approach allows you to break many complex and confounding crashes into approachable, provable hypotheses.

If you are fluent in assembly, I hope we've demonstrated just how deeply it's possible to dive on a difficult-to-reproduce issue. We performed all of this analysis post-mortem, with only a crash report and no reproduction case – we never even actually ran the application in question.

In future issues, we'll address some of the tools we used in this analysis, including Hopper and IDA Pro. If you have a particular crash-related topic you'd like to see covered, please send it in.

If you're facing your own particularly insidious bug, or require deeper insight into the crash reporting process, we offer commercial support services as part of our work on the open-source PLCrashReporter project. In exchange for our helping to solve your crashes, you help fund ongoing development to make PLCrashReporter even better!