Relic: ObjC Introspection

An invisible approach

The Basics

Infiltrating opaque mechanisms allows the modifier of the process complete discretion over the control flow of the executable.

NOTE 5/28/2026: FULL CODE AT BOTTOM OF DOCUMENT

On iOS platforms many applications leverage the powerful objective C language. In the code you might see statements like this:

```
[ApplicationViewController userClickedButton:button]
```

This simply calls a method from the ApplicationViewController class called userClickedButton. The button variable could be an anonymous object, often called an “id” object, or something more specific which likely inherits from the base id “object” or some NSObject. But under the hood it looks more like this.

This snippet corresponds to:

```
[[UIApplication sharedApplication] sendAction:suspend to:(x26)
forEvent:NULL]
```

Traditionally, in order to hook certain methods the objective C runtime provides functions to remap the actual address where the destination function resides, such as “extern IMP method_setImplementation(Method m, IMP imp)”.

This means that in order to process and manage any given method call, the IMP needs to be swapped with a different function to preserve original functionality and process the event.

But all the method calls and variable accesses happen through the obj_msgSend function!

The Complicateds

Assume we have a jailbroken device where CoreTrust is a little more lax (this will come in later). So what if we want to intervene in all method calls within our process space? We need to target the core messaging apparatus. This is a high performance function written in assembly, being called more than thousands of times per second in some applications. The technical and performance costs are nothing compared to the power and flexibility. The main advantages to modifying a function like objc_msgSend are:

- Preventing the execution of certain functions
- Modifying the return value of certain method calls
    - Even doing this dependent on the callee argos
- Executing a replacement method instead of the callee's intention
- Responding to methods and classes that don't actually exist

And then you can do all this by matching method or class names with a pattern. So for example, any classes that start in "UIView*" can be targeted and logged for any activity.

This functionality is not unlike the powerful Substrate/Substitute hooking library's MSHookMessageEx() function.

When we compile a binary we generate a short function that performs an address relative load and then fetches the linked address of the messaging function.

```
1000ef428  int64_t _objc_msgSend()

1000ef428  300300b0   adrp    x16, 0x100154000
1000ef42c  106646f9   ldr     x16, [x16, #0xcc8]  {_objc_msgSend}
1000ef430  00021fd6   br      x16
```

This means that we can quickly and simply overwrite this pointer to take full control of the function. But once we have control we need to handle a plethora of edge cases, as any discrepancy in the way that messages are handled will *surely* mess up the control flow.

It is totally possible to do. We can use a HashMap for larger lists of targeted methods, or a simple linear search for <20 targets. If we want nothing to do with a particular method, we pass it on down msgSend and let the default path deal with it.

So it seems simple. Use Facebook's Fishhook library (great stuff btw) to remap the address to our well written handler function for minimal overhead and easy introspection.

Another issue arises. What if, for example, a handler for a button or something is calling something in the main executable from the UIKit part of the shared cache binary blob? Well that wouldn't use the __DATA address anyways. We want to handle all Objective C executions.

The good news is that Apple writes the objc_msgSend function in assembly, so it doesn't change a lot and it's open source. It uses a hashmap style O(1) mapping and various optimizations to pull special information about where methods are and how to handle them, it seems.

So our technique becomes modifying the shared cache (don't worry, it's copy-on-write). In order to redirect control flow to other codepages, we have to generate and patch instructions on the fly.

Here's some real lightweight meat-and-potatoes type instruction generation.

```
//encodes movz (64 bit)
uint32_t inst_generate_movz(uint8_t reg, uint16_t imm, uint8_t shift){
    uint32_t fmt = 0b11010010100000000000000000000000;
    uint32_t hw = (uint32_t)(shift >> 4) << 21;
    uint32_t imm16 = (uint32_t)imm << 5;
    uint32_t rd = reg & 0x1f;
    return fmt | hw | imm16 | rd;
}
//encodes movk (64 bit)
uint32_t inst_generate_movk(uint8_t reg, uint16_t imm, uint8_t shift){
    uint32_t fmt = 0b11110010100000000000000000000000;
    uint32_t hw = (uint32_t)(shift >> 4) << 21;
    uint32_t imm16 = (uint32_t)imm << 5;
    uint32_t rd = reg & 0x1f;
    return fmt | hw | imm16 | rd;
}
//encodes br
uint32_t inst_generate_br(uint8_t reg){
    uint32_t fmt = 0b11010110000111110000000000000000;
```

```
    uint32_t rn = ((uint32_t)reg & 0x1F) << 5;
    return fmt | rn;
}
```

These are super useful, with only 4 instructions we can jump to an arbitrary 48 bit address (sooo, assuming no tagged/signed pointers). We will copy the original address plus the offset of where it will continue after the patched code into our handler routine. We also need to consider that the null pointer check involves a jump even further into the original function, at an offset encoded by a b instruction.

```
****SEE BOTTOM
```

```
    Original objc_msgSend function:
    ═══════════════
    ENTRY:
    cmp x0, #0
    b.le #0xd0
    ldr x14, [x0]
    and x16, x14, ISA_MASK
    ═══════════════


    Replaced code:
    ═══════════════
    ENTRY:
    movz x14, #0x1234, lsl #32
    movk x14, #0x5678, lsl #16
    movk x14, #0x9abc, lsl #0
    br x14
    ═══════════════
```

There's honestly so much going on here. In the first few instructions of the victim function (the messaging apparatus) certain actions are performed like calculating the ISA_MASK and performing a null check. We need to extract the jump offset from the b instruction in the victim and use that offset to jump back into it if a null value occurs. We need to calculate the mask and leave that in x16.

BIG IDEA: By moving part of the functionality of the original into our handler, we can both redirect control flow and properly handle null parameters, sending execution to the right part of the victim function with x16 containing the ISA value in cases where no hooking occurs. When we want to intercept a function we simply check the class and method arguments against a list of functions that are registered for

interception. We can send control right back whenever we want or choose to omit it entirely.

Heres a snippet from a more primitive version of relic that shows this in action. In a new handler for objc_msgSend we push all of the relevant registers and enter a hookmanager function that determines if a function should be allowed to pass, and if it should be intercepted and executed or just intercepted and it’s return value replaced.

```
// arm64(e) witchcraft

struct OrigAndReturn {
  uintptr_t orig;
  uintptr_t ret;
};

struct OrigAndReturn hookmanager(id self, SEL _cmd, uint64_t arg2,
uint64_t arg3, uint64_t arg4, uint64_t arg5, uint64_t arg6)
asm("hookman");

typedef id (*relic_callback)(id self, SEL _cmd, uint64_t arg2, uint64_t
arg3, uint64_t arg4, uint64_t arg5, uint64_t arg6);

struct OrigAndReturn hookmanager(id self, SEL _cmd, uint64_t arg2,
uint64_t arg3, uint64_t arg4, uint64_t arg5, uint64_t arg6) {
    void * hook = getHook(self,_cmd);
    if(hook){
        relic_callback replacement = (relic_callback)hook;
        return (struct OrigAndReturn) {0, (uintptr_t)replacement(self,
_cmd, arg2, arg3, arg4, arg5, arg6)};
    }
    return (struct OrigAndReturn)
{reinterpret_cast<uintptr_t>(orig_objc_msgSend), 0}; //execute, keep
ret //reinterpret_cast<uintptr_t>(*orig_objc_msgSend) for 0 # 1
}

__attribute__((__naked__)) static void replacementObjc_msgSend() {
  __asm__ volatile (
                    "stp q6, q7, [sp, #-32]!\n"
                    "stp q4, q5, [sp, #-32]!\n"
                    "stp q2, q3, [sp, #-32]!\n"
                    "stp q0, q1, [sp, #-32]!\n"
                    "stp x8, lr, [sp, #-16]!\n"
                    "stp x6, x7, [sp, #-16]!\n"
                    "stp x4, x5, [sp, #-16]!\n"
                    "stp x2, x3, [sp, #-16]!\n"
                    "stp x0, x1, [sp, #-16]!\n"
```

```
                    "bl hookman\n"
                    "bic x9, x0, #0xFFFFFF8000000000\n"
                    "mov x10, x1\n"
                    "ldp x0, x1, [sp], #16\n"
                    "ldp x2, x3, [sp], #16\n"
                    "ldp x4, x5, [sp], #16\n"
                    "ldp x6, x7, [sp], #16\n"
                    "ldp x8, lr, [sp], #16\n"
                    "ldp q0, q1, [sp], #32\n"
                    "ldp q2, q3, [sp], #32\n"
                    "ldp q4, q5, [sp], #32\n"
                    "ldp q6, q7, [sp], #32\n"
                    "cbz x9, Lnocall\n"
                    "br x9\n"
                    "Lnocall:\n"
                    "mov x0, x10\n"
                    "ret\n"
    );
}

__attribute__((constructor)) static void inject() {
    MSHookFunction((void*)&objc_msgSend, (void
*)&replacementObjc_msgSend, (void **)&orig_objc_msgSend);
    hookMap = HMCreate(&pointerEquality, &pointerHash);
}
```

Conclusion

Hooking objc_msgSend is a volatile yet powerful way to manipulate objective C code. By directly checking the methods and classes we can selectively handle different calls. This was used successfully to evade detection by secure applications such as private social medias, game anticheats, and banking apps that limit system modification. *Infiltrating opaque mechanisms allows the modifier of the process complete discretion over the control flow of the executable.*

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

#include <mach-o/dyld.h>
#include <stdio.h>
#include <mach-o/dyld.h>
#include <dlfcn.h>
```

```
#include <objc/runtime.h>
#include <fcntl.h>
#include <unistd.h>
#import <AudioToolbox/AudioToolbox.h>

#include <sys/mman.h>

#include <mach/vm_prot.h>
#include <mach/vm_types.h>
#include <mach/mach.h>
#include <stdint.h>
#include <ptrauth.h>

#include <mach/arm/kern_return.h>

#include "inst.h"
#include "artifact.h"

//only mess with MAX_HOOKS
#define BLOCK_EXECUTION NULL
#define MAX_HOOKS 10


//https://github.com/opensource-apple/objc4/blob/cd5e62a5597ea7a31dccef089317abb3a661c154/runtime/Messengers.subproj/objc-msg-arm.s#L110


//memory protection from litehook
__attribute__((noinline, naked)) volatile kern_return_t
syscall_vm_protect(mach_port_name_t target, mach_vm_address_t address,
mach_vm_size_t size, boolean_t set_maximum, vm_prot_t new_protection){
   __asm("mov x16, #-14");
   __asm("svc 0x80");
   __asm("ret");
}

//Checks which executable we are injected into
int selectiveInjection(const char *binaryName){
   char path[1024];
```

```
    uint32_t s_path = sizeof(path);
    if(_NSGetExecutablePath(path, &s_path) == 0)
        if(strstr(path, binaryName) != NULL)
            return 1;
    return 0;
}

//hook storage structure
int num_hooks = 0;
Class hooked_classes[MAX_HOOKS];
SEL hooked_selectors[MAX_HOOKS];
void *hooks[MAX_HOOKS];

//DONT TOUCH - offsets to continue execution
uint64_t LNilOrTagged = 0xd0;
uint64_t LGetIsaDone = 0x10;


void ArtifactHookMessage(const char * class_name, const char*
selector_name, void * replacement, void * original){
    Class _Nullable cls = objc_getClass(class_name);
    SEL cmd = sel_registerName(selector_name);

    if(!cmd || !cls || !replacement) return;

    if(num_hooks < MAX_HOOKS){
        hooks[num_hooks] = replacement;
        hooked_classes[num_hooks] = cls;
        hooked_selectors[num_hooks] = cmd;
    }
    num_hooks = num_hooks + 1;

    if(original != NULL) *(IMP *) original =
class_getMethodImplementation(cls, cmd);
}

volatile void *execution_intervention(id __unsafe_unretained self, SEL
_cmd) asm("____exint");
volatile void *execution_intervention(id __unsafe_unretained self, SEL
```

```
_cmd){
    if((int64_t)self > 0){
        Class _Nullable cls = NULL;
        for(int i = 0; i < num_hooks; i++){
            if(_cmd == hooked_selectors[i]){
                if(cls == NULL) cls = object_getClass(self);
                if(cls == hooked_classes[i]){
                    //maybe check if hooks[i] is a tagged pointer?
shouldnt be
                    return (void*)hooks[i];
                }
            }
        }
    }
    return BLOCK_EXECUTION;
}

__attribute__((noinline, naked)) void patched_handler(){
    //store shit
    __asm(
        "stp q6, q7, [sp, #-32]!\n"
        "stp q4, q5, [sp, #-32]!\n"
        "stp q2, q3, [sp, #-32]!\n"
        "stp q0, q1, [sp, #-32]!\n"
        "stp x8, lr, [sp, #-16]!\n"
        "stp x6, x7, [sp, #-16]!\n"
        "stp x4, x5, [sp, #-16]!\n"
        "stp x2, x3, [sp, #-16]!\n"
        "stp x0, x1, [sp, #-16]!\n"
    );
    //perform handler in c code
    __asm("bl ____exint");
    //store in x13
    __asm("mov x13, x0");

    //load control flow values
    __asm("mov x11, %x[value]" : : [value] "r" (LNilOrTagged));
    __asm("mov x10, %x[value]" : : [value] "r" (LGetIsaDone));
    //restore register
```

```
    __asm(
        "ldp x0, x1, [sp], #16\n"
        "ldp x2, x3, [sp], #16\n"
        "ldp x4, x5, [sp], #16\n"
        "ldp x6, x7, [sp], #16\n"
        "ldp x8, lr, [sp], #16\n"
        "ldp q0, q1, [sp], #32\n"
        "ldp q2, q3, [sp], #32\n"
        "ldp q4, q5, [sp], #32\n"
        "ldp q6, q7, [sp], #32\n"
    );
    //logic to handle a call
    __asm(
        "cbz x13, Lpass_call\n"
        "br x13\n"
        "Lpass_call:\n"
    );

    //handle control flow
    __asm("cmp x0, #0");
    __asm("b.gt #8");
    __asm("br x11");
    //not tagged or nil
    __asm("ldr x14, [x0]");
    __asm("and x16, x14, #0x7ffffffffffff8");
    __asm("br x10");
}

//patches the messaging apparatus to manipulate calls, thx opa334
kern_return_t ArtifactBeginHooking(){

    void *source = dlsym(RTLD_DEFAULT, "objc_msgSend");
    void *target = patched_handler;

    if(source == NULL) return KERN_FAILURE;

    /* DONT MODIFY THIS SHIT
    Original function:
    ==================
```

```
    cmp x0, #0
    b.le #0xd0
    ldr x14, [x0]
    and x16, x14, ISA_MASK
    ==================

    Replaced code:
    ==================
    movz x14, #0x1234, lsl #32
    movk x14, #0x5678, lsl #16
    movk x14, #0x9abc, lsl #0
    br x14
    ==================
    */

    kern_return_t kr = KERN_SUCCESS;
    size_t patch_size;

    uint32_t *victim = (uint32_t*)ptrauth_strip(source,
ptrauth_key_process_independent_code);
    uint64_t handler = (uint64_t)ptrauth_strip(target,
ptrauth_key_process_independent_code);

    LNilOrTagged += (uint64_t)victim;
    LGetIsaDone += (uint64_t)victim;

    //APPLY PATCH FOR VICTIM FUNCTION
    patch_size = 4 * 4;

    kr = syscall_vm_protect(mach_task_self(), (vm_address_t)victim,
patch_size, false, VM_PROT_READ | VM_PROT_WRITE | VM_PROT_COPY);
    if(kr != KERN_SUCCESS) return kr;

    uint8_t sacrificial_reg = 10;

    victim[0] = inst_generate_movz(sacrificial_reg, (handler >> 32) &
0xFFFF, 32);
    victim[1] = inst_generate_movk(sacrificial_reg, (handler >> 16) &
0xFFFF, 16);
```

```
    victim[2] = inst_generate_movk(sacrificial_reg, (handler >> 0) &
0xFFFF, 0);
    victim[3] = inst_generate_br(sacrificial_reg);


    kr = syscall_vm_protect(mach_task_self(), (vm_address_t)victim,
patch_size, false, VM_PROT_READ | VM_PROT_EXECUTE);
    if(kr != KERN_SUCCESS) return kr;

    return kr;
}
```