Mac OS Xploitation

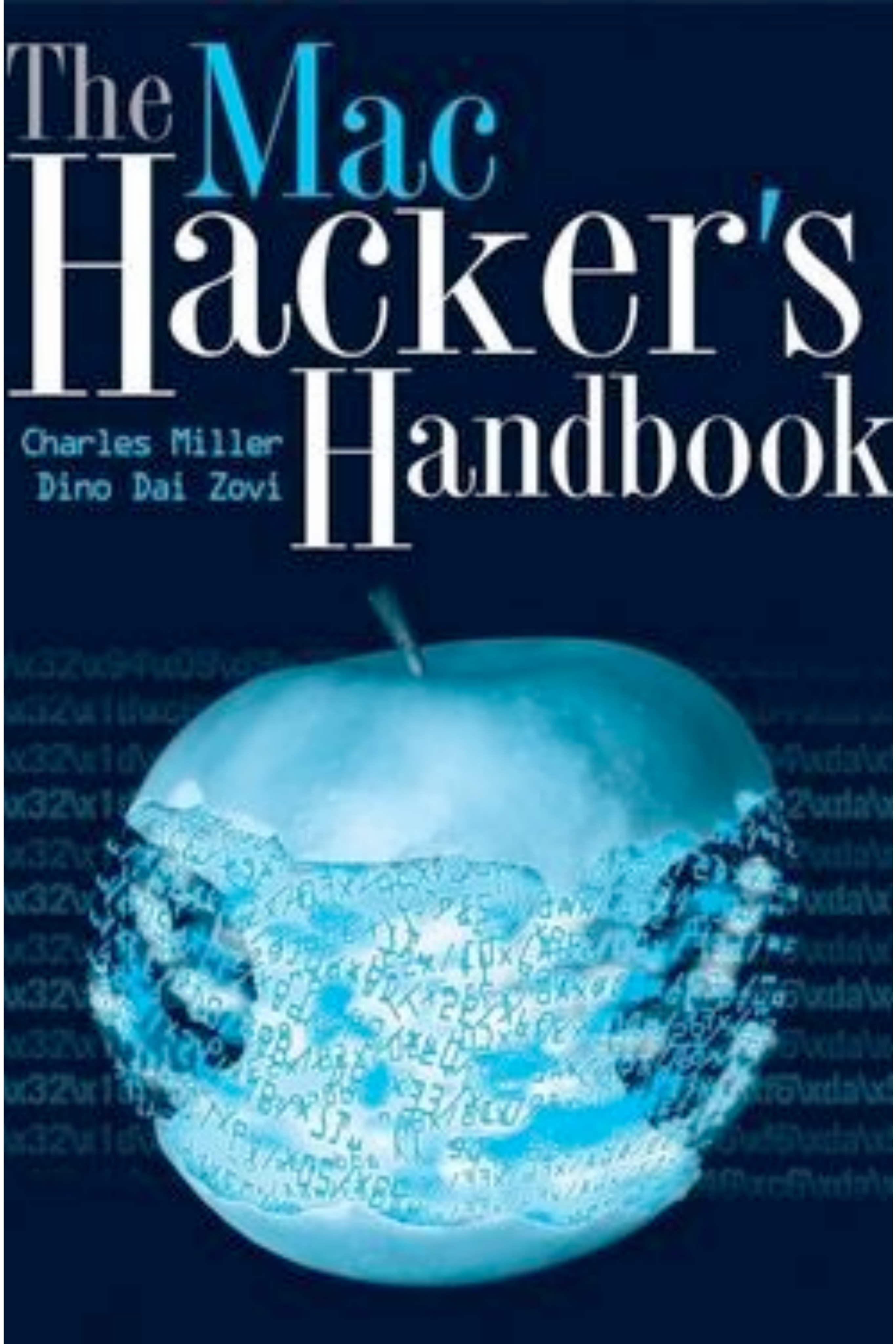
Dino A. Dai Zovi
Offensive Security Researcher
ddz@theta44.org
http://trailofbits.com
http://theta44.org

Overview

- Shameless plug
- Safety vs. Security
- Exploiting memory corruption vulnerabilities
  - Bypassing Leopard’s library randomization and non-executable stack
  - Scalable Zone heap allocator and heap metadata overwrites
  - Heap Feng Shei
- Exploit payloads
  - Mach-O symbol resolver
  - Dynamic bundle injection
  - “Take a Pic of the Vic”

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 I mention that there's free 0day in it?



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 two years in a row (a three-peat is very likely ;))
  - 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

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

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

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

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.

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
```

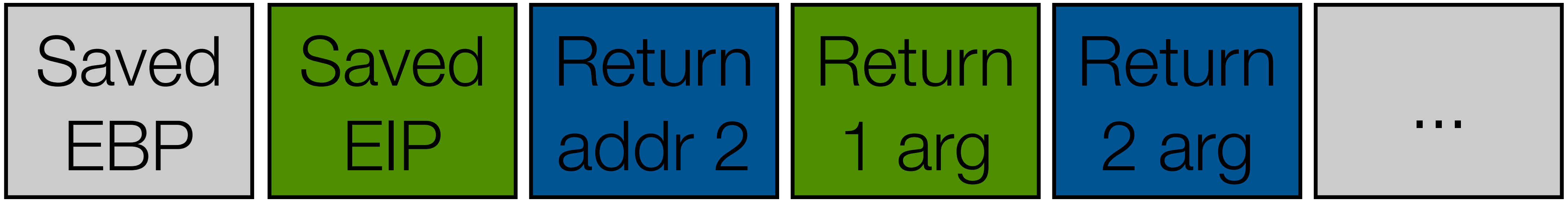
Non-eXecutable Memory

- 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**

Stack Corruption

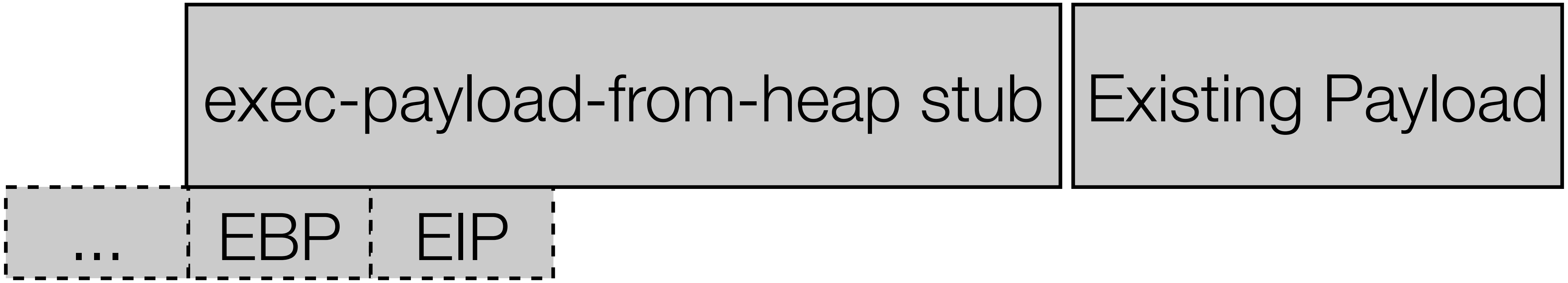
Library Randomization and NX Stack Bypass

- 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

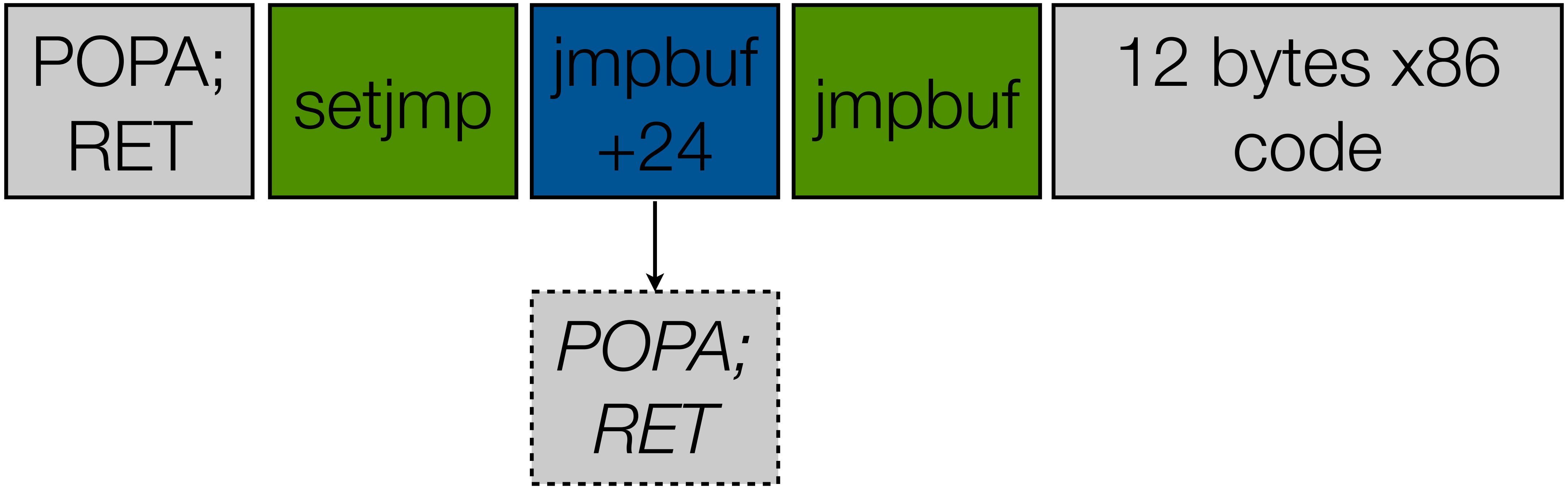
...

EBP

EIP

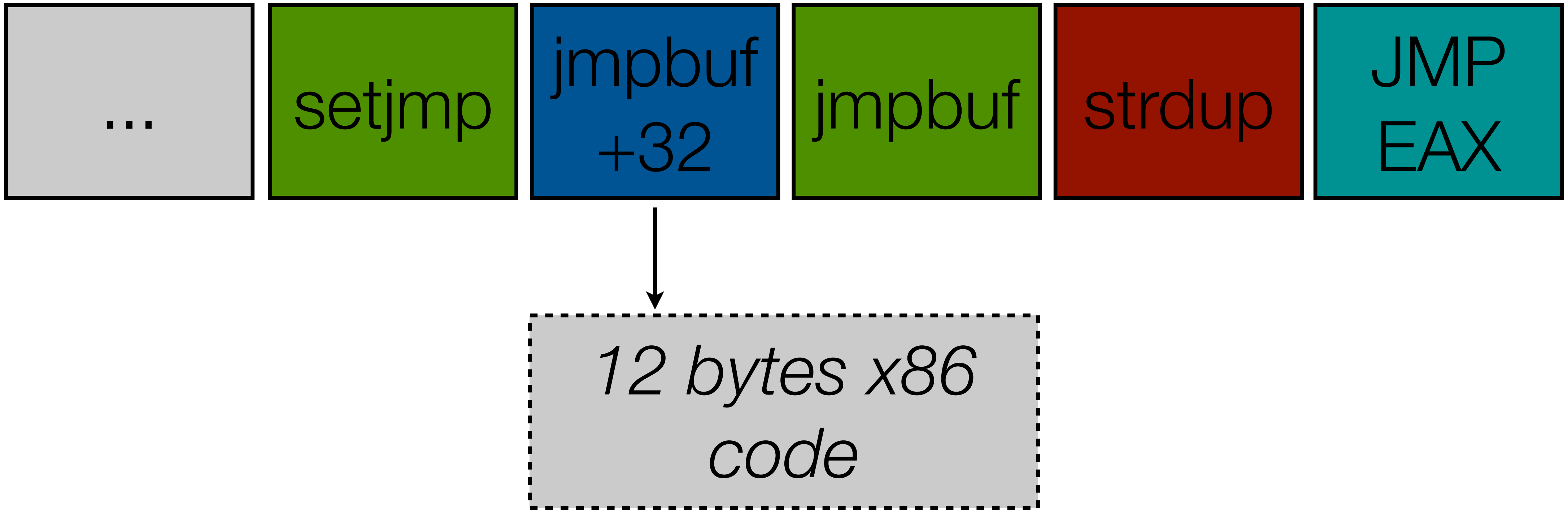
Execute Payload From Heap Stub

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()



GCC Stack Protector

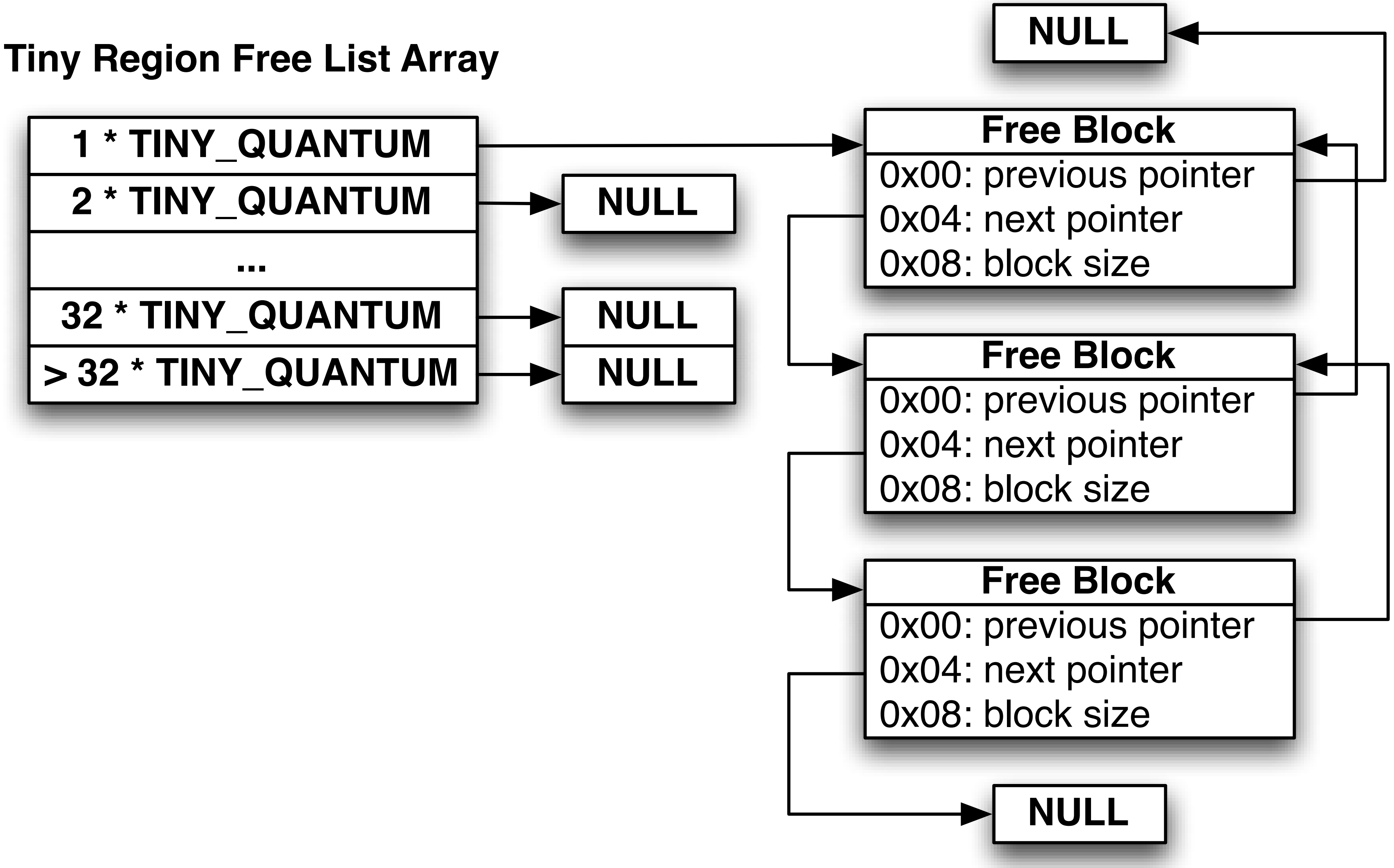
- Adds a guard variable to stack frames potentially vulnerable to stack buffer overflows
- Guard variable (aka “canary”) is verified before returning from function
  - ___stack_chk_guard() function
- Effectively stops exploitation of most stack buffer overflows
  - Potentially ineffective against some 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

Heap Corruption

Scalable Zone Heap Allocator

- Scalable Zone Heap’s security is so 1999
  - /* 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



Tiny Region Free List Array

- 1 * TINY_QUANTUM
- 2 * TINY_QUANTUM
- ...
- 32 * TINY_QUANTUM
- > 32 * TINY_QUANTUM

NULL

NULL

NULL

Free Block

0x00: previous pointer

0x04: next pointer

0x08: block size

Free Block

0x00: previous pointer

0x04: next pointer

0x08: block size

Free Block

0x00: previous pointer

0x04: next pointer

0x08: block size

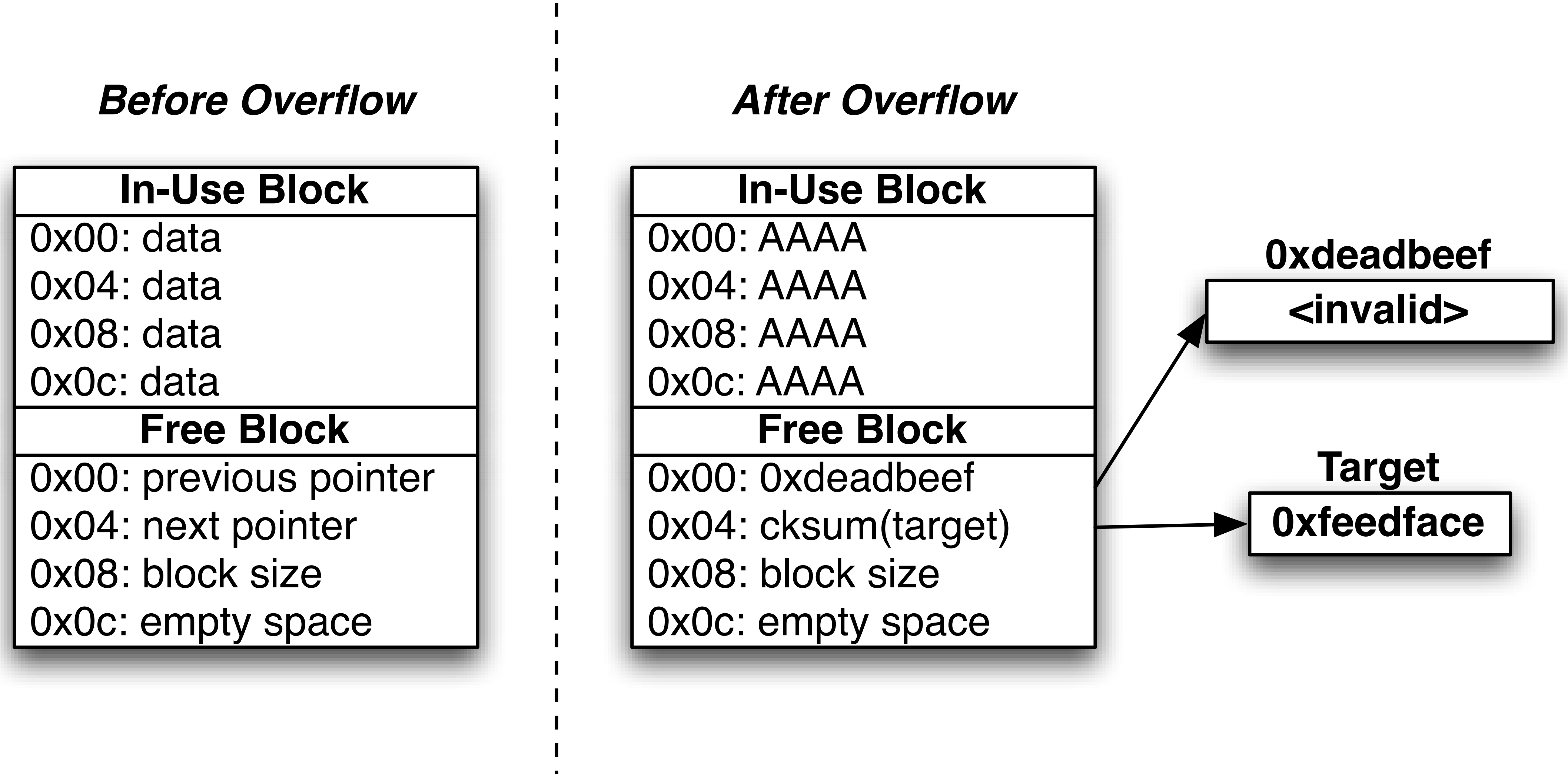
NULL

NULL

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

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

In-Use Block
0x00: AAAA
0x04: AAAA
0x08: AAAA
0x0c: AAAA

Free Block
0x00: 0xdeadbeef
0x04: cksum(target)
0x08: block size
0x0c: empty space

0xdeadbeef
<invalid>

Target
0xfeedface

Heap Pointer Checksums

- Free list pointer 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 imported function pointers
  - __OBJC segments with method pointers
  - MALLOC regions

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);`

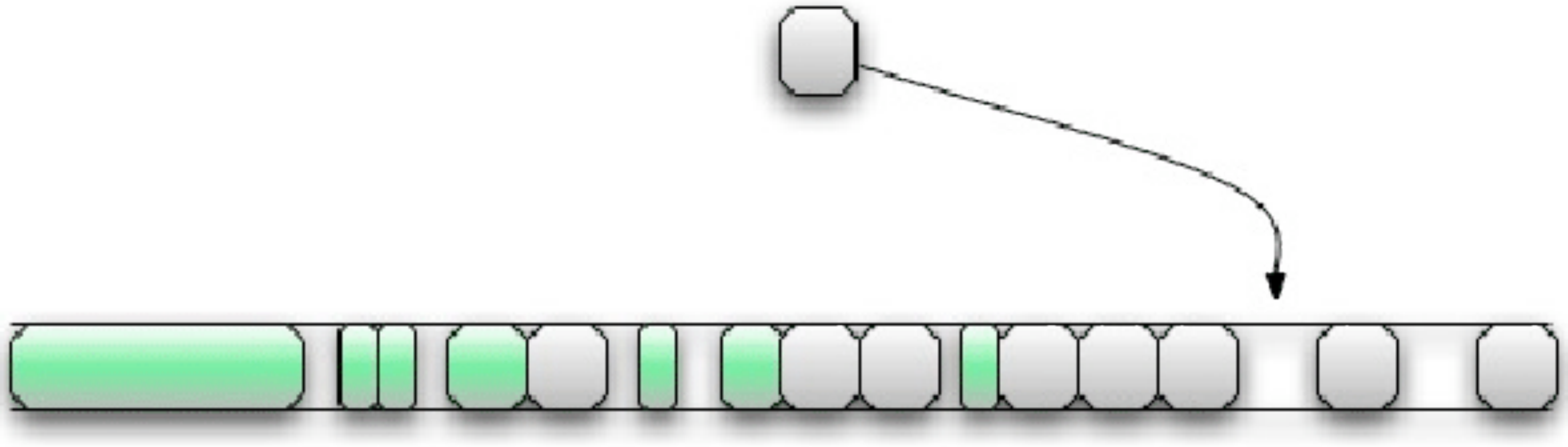
   B removed from free list, *(uncksum(B->next)) = B->prev

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)



Exploit Payloads

Mach-O Function Resolver

- Dyld is always loaded 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
  - Technique similar to LSD’s Win32 Assembly Components
- 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 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

Mach-O Staged Bundle Injection Payload

- Third stage (compiled bundle, can be as large as needed)
  - Does whatever you want
  - Can use C, C++, Objective-C and any Frameworks
  - Must export an `int run(int socket_fd)` function
  - Pure-memory injection, not written to disk
  - Bundles are relatively compact; a “hello world” bundle is ~12 KB

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 CocoaSequenceGrabber from Amit Singh’s MacFUSE procfs:

```
(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];
}
```

Demo

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
  - iSight photo capture payload
  - More to follow soon...

Final Remarks

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*.

Questions?