# Mac OS Xploitation

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

# Why talk about Mac exploits?

- Macs are becoming more prevalent, especially in consumer laptops
  - Macs were 20% of laptops sold in the U.S. during July and August
- Memory corruption vulnerabilities enable system compromise, worms, spyware, and other malware
- Mac OS X is significantly lacking in memory corruption defense features compared to other current operating systems like Windows Vista and Linux
  - ASLR, Non-eXecutable memory, stack and heap memory protections
- Difference between Safety and Security
  - Level of Risk = Threats * Vulnerability * Attack Likelihood
  - Threats and Attack Likelihood are currently low, Vulnerability is still high

# Memory Corruption

# Memory Corruption Vulnerabilities

- Many types of vulnerabilities that can lead to remote arbitrary 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 memory 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

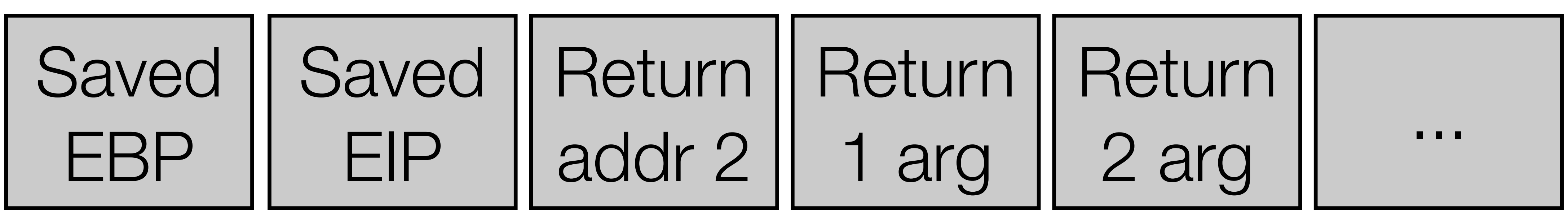
```
mapping EX  112MB 0x90000000 -> 0x9708E000
mapping RW    8MB 0xA0000000 -> 0xA083E000
mapping EX  660KB 0xA0A00000 -> 0xA0AA5000
mapping RO    5MB 0x9708E000 -> 0x97630000
/System/Library/Frameworks/ApplicationServices.framework/Versions/A/Frameworks/C
olorSync.framework/Versions/A/ColorSync
                  __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**

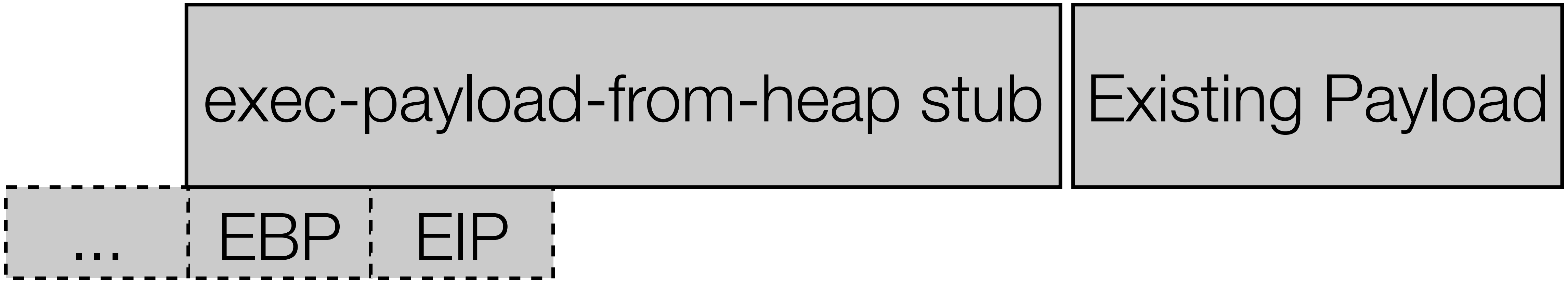
# Library Randomization and NX Stack Bypass

- 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



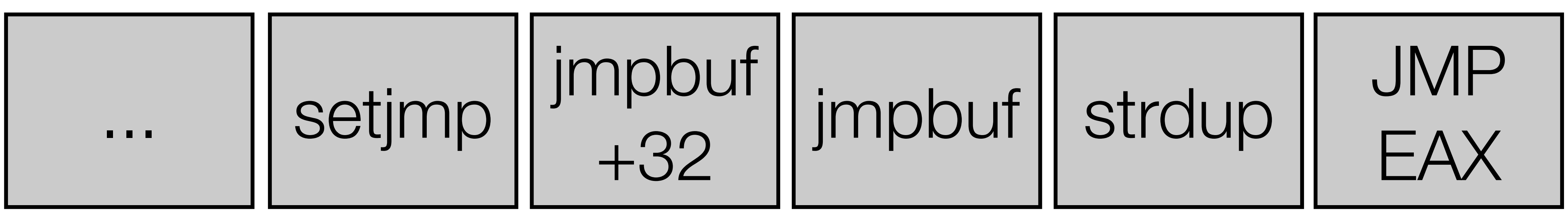
# Execute Payload From Heap Stub

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

| | | | | |
|---|---|---|---|---|
| POPA; RET | setjmp | jmpbuf +24 | jmpbuf | 12 bytes x86 code |

# 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()



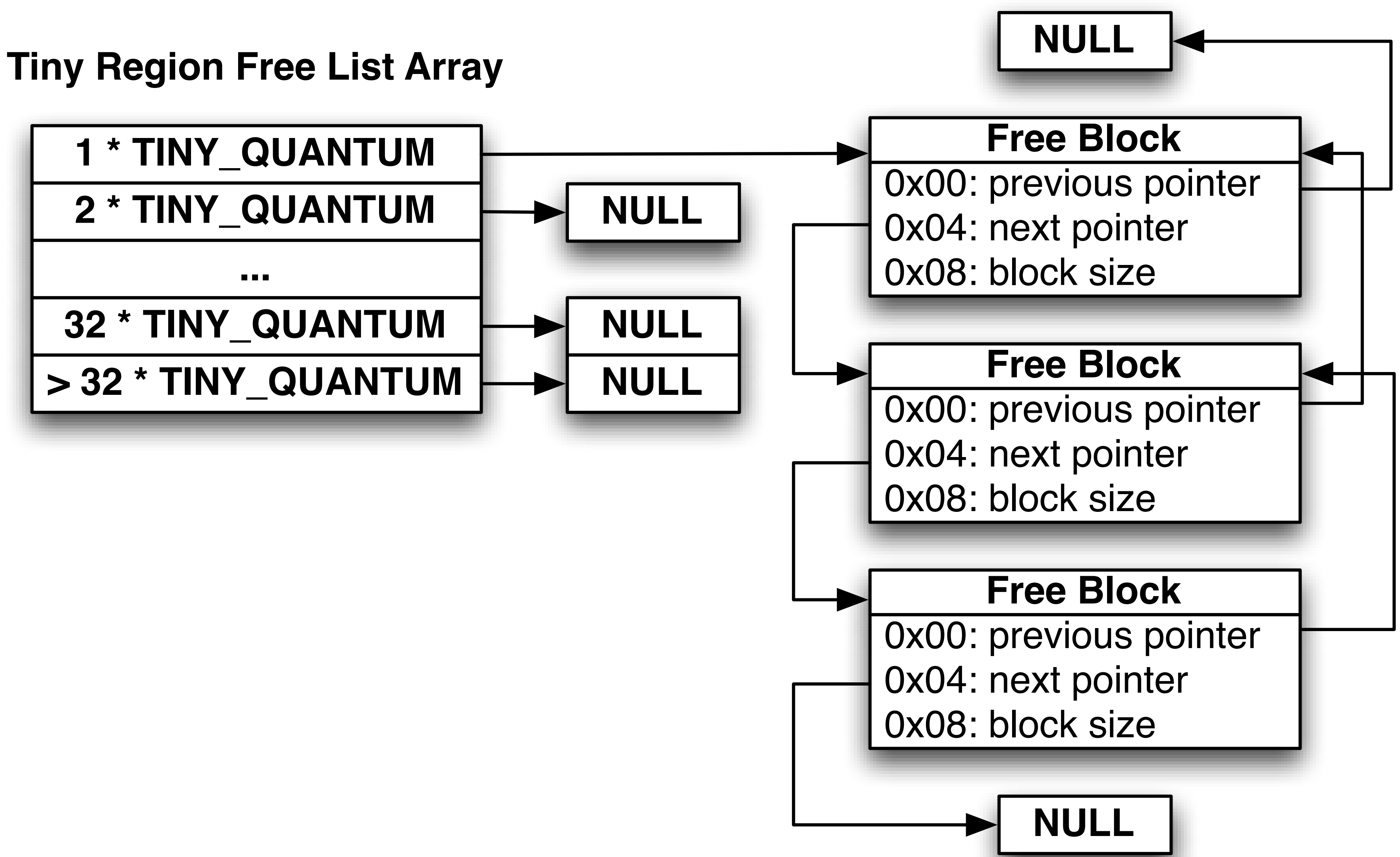
# GCC Stack Protector

- 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**
  - QuickTime is an exception now
  - Started using stack protection in an update after Leopard was released

# Scalable Zone Heap Allocator

- Scalable Zone Heap's security is very 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 regions quantum size
- Free blocks are stored in arrays of 32 free lists, indexed by size in quanta

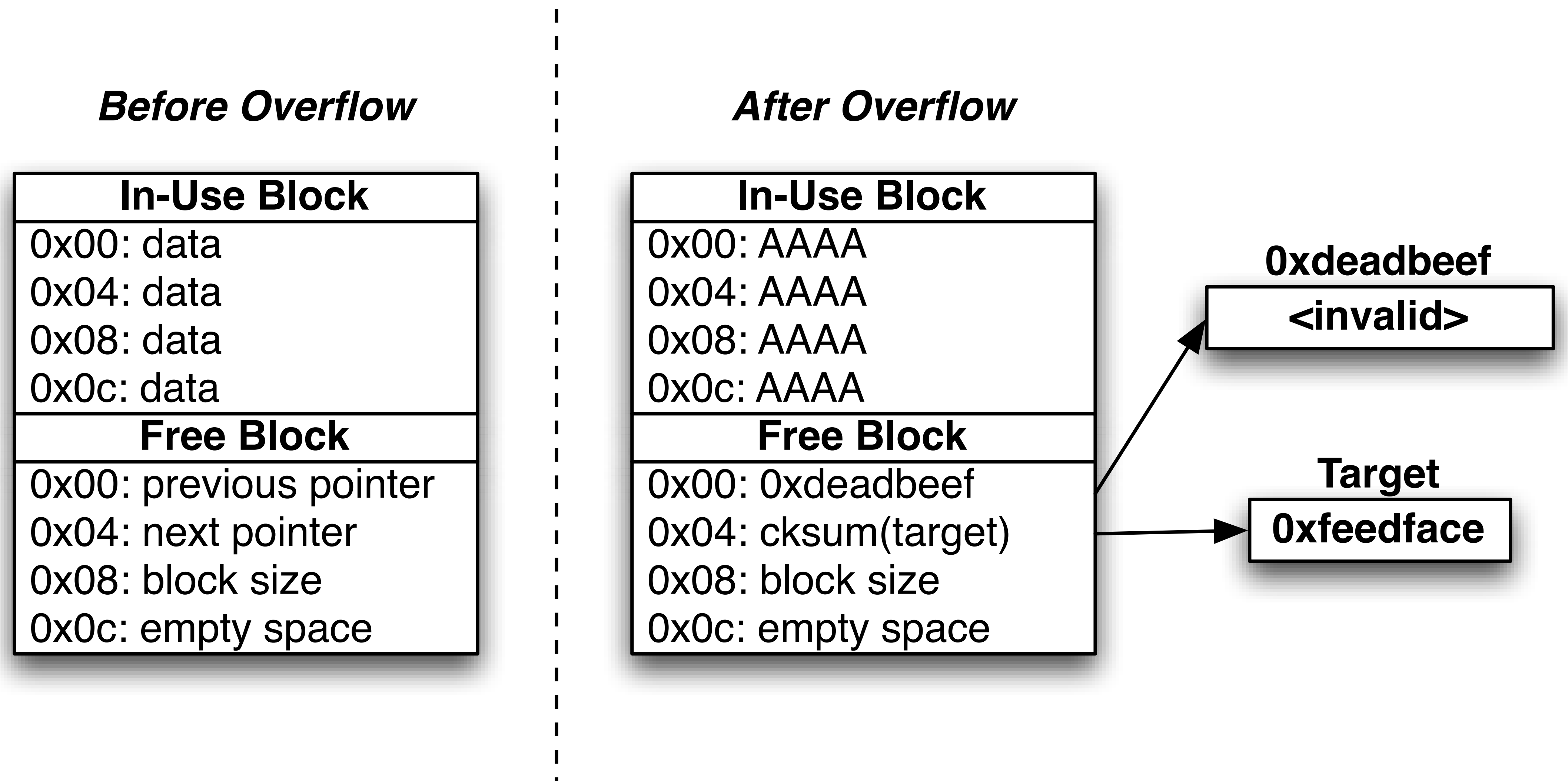
# Free List Arrays



# 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 cause memory allocation/deallocation and control sizes

# Heap Metadata Overwrite



# 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

# Heap Metadata Write4

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

  1. A = malloc(N);
  2. B = malloc(M);
  3. free(B)
  4. // overflow A -> B, overwrite B->prev, B->next
  5. C = malloc(M); // 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(N);
2. B = malloc(M);
3. free(B)
4. // overflow A -> B, overwrite B->prev, B->next
5. C = malloc(M); // B removed from free list, *(uncksum(B->next)) = B->prev
6. D = malloc(M); // D == B->next
7. // 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 the allocation of a target object falls right after a heap block allocation that can be overflown
- Technique used by Charlie Miller (speaking tomorrow) to exploit Safari PCRE vulnerability and win PWN2OWN at CanSecWest 2008

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 from Last Stage of Delirium’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 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

# 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
  - Lookup and call NSLinkModule() in dyld
  - Lookup and call run(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 still 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

- 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];
}
```

# Metasploit Modules To Be Released Soon

- Exploits
  - mDNSResponder UPnP Location Header Overflow (10.4.0,10.4.8 x86/ppc)
    - Was on by default, through firewall, remote root on Tiger
  - 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)
    - Vulnerability used to win PWN2OWN at CanSecWest 2007
- Payloads
  - Staged Mach-O Bundle Injection
  - iSight photo capture payload

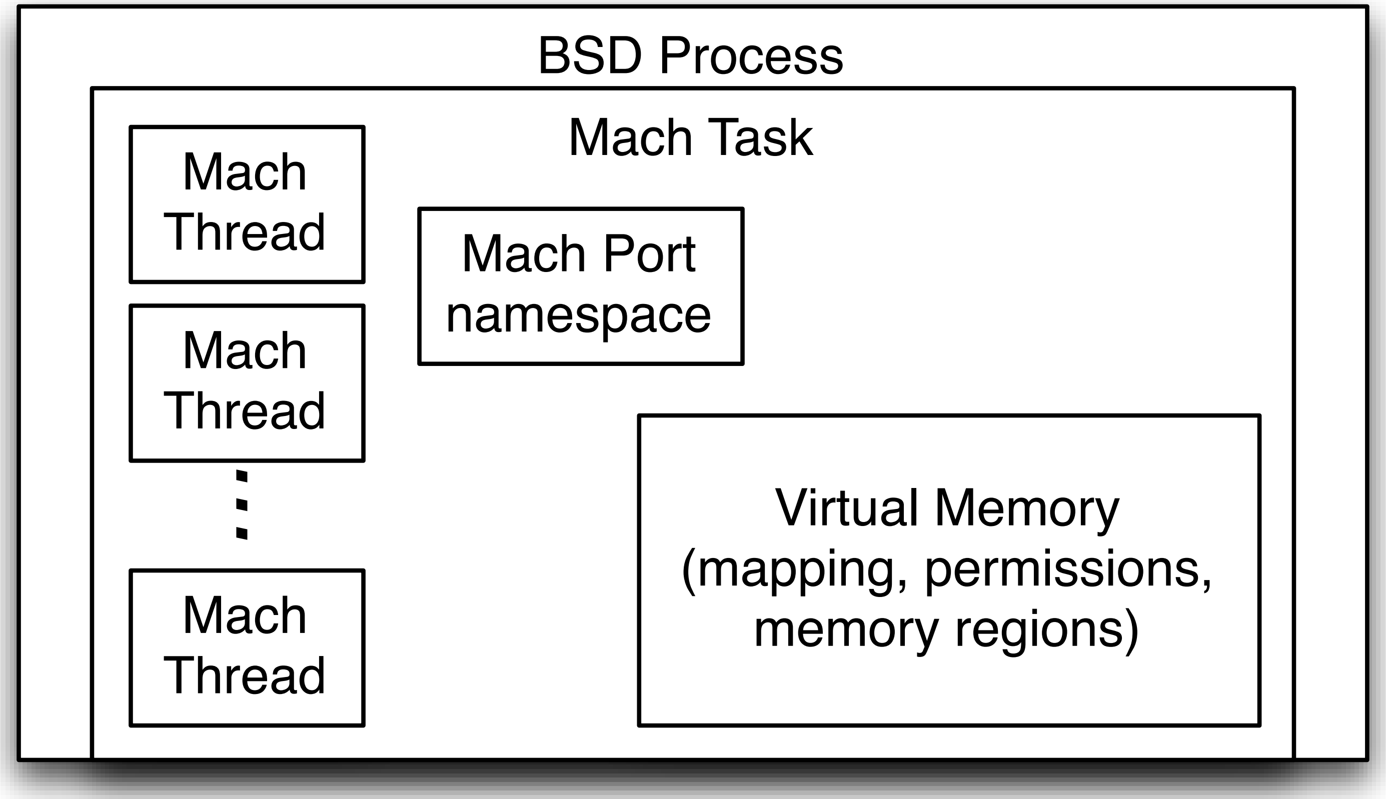
Mach Thread and Bundle Injection

# 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

Final Remarks

# 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
  - Must instead return to code fragments to load registers before returning into next function
- Exploiting 64-bit processes requires one-off tricks, not general techniques
- **Very few security-sensitive processes are 64-bit on Leopard (except**

# 10.6 Snow Leopard

- Security and Stability update to Leopard
- 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?**

# Conclusion

- MacOS X is vulnerable to the same type of malware attacks as Windows
- Significantly lags behind Windows and Linux in memory corruption defenses
  - ASLR, NX, Stack and Heap protection
- Writing exploits for Vista is *hard work*, writing exploits for Mac is *fun*.

Questions?