Mach-O Tricks

~qwertyoruiopz@BaijiuCon

Agenda

- What is Mach-O
- Mach-O format
- Mach-O bugs
- Mach-O obfuscation tricks
- Loading Mach-O on exotic platforms

What is MachO?

- Mach object file format
  - Used for executables, dynamic libraries, kernel extensions, core dumps, etc..
- Used in XNU-based OSes (iOS/MacOS X) and by SEPOS
- Describes a portion of address space
  - Permissions, contents, etc..
- 32 as well as 64 bit support, little or small endian

What is MachO?

- Can be dynamically linked
  - Not just libraries, you can link against normal executables as long as they're not stripped
  - Support for lazy symbols
- Can describe so-called __PAGEZERO
  - Essentially used to prevent NULL page mapping
- Fairly easy format to parse and play with
- Two representations: VM and file

Mach-O Format

- All Mach-O files begin with the Mach-O header
  - struct mach_header(_64)

```
/*
 * The 64-bit mach header appears at the very beginning of object files for
 * 64-bit architectures.
 */
struct mach_header_64 {
        uint32_t        magic;          /* mach magic number identifier */
        cpu_type_t      cputype;        /* cpu specifier */
        cpu_subtype_t   cpusubtype;     /* machine specifier */
        uint32_t        filetype;       /* type of file */
        uint32_t        ncmds;          /* number of load commands */
        uint32_t        sizeofcmds;     /* the size of all the load commands */
        uint32_t        flags;          /* flags */
        uint32_t        reserved;       /* reserved */
};
```

Mach-O Format

- Bitness and endianness can be detected from magic value
  - 0xfeedface: 32 bit, native endianness
  - 0xcefaedfe: 32 bit, representing the other endianness
  - 0xfeedfacf: 64 bit, native endianness
  - 0xcffaedfe: 64 bit, representing the other endianness

Mach-O Format

- After Mach-O header, a list of so-called Load Commands is present
  - ncmds and sizeofcmds in header respectively tell you how many load commands and how many bytes are taken up by all the load commands
- Each load command shares a common prefix that tells you the type of load command and size of load command
- To scan load commands you just add the size of current load command to current load command offset

```
struct load_command {
        uint32_t cmd;           /* type of load command */
        uint32_t cmdsize;       /* total size of command in bytes */
};
```

Mach-O Format

- LC_SEGMENT(_64) describes a memory map
- Virtual Address of the map can be set for non-PIE Mach-Os
  - On PIE Mach-Os, the virtual addresses are rebased to a memory area reserved for you
- Permissions of a given map
- Full size of the reserved memory area, incl. uninitialized data (e.g. BSS)
- Size of the mapped memory area and file offset passed to mmap

Mach-O Format

- LC_DYLD_INFO(_ONLY) describes information passed to the dynamic linker
  - Exported symbols
  - Normal imported symbols
  - Lazily-bound imports
  - Weakly-bound imports
  - Rebase information for PIE binaries

Mach-O Format

- A lot of other significant load commands
  - LC_MAIN: entry point
  - LC_UNIXTHREAD: register state at runtime
  - LC_LOAD_DYLINKER: file path of dynamic linker
  - LC_(DY)SYMTAB: symbol tables
  - LC_DYLIB: tells dynamic linker you want to link against some other Mach-O at load time

Mach-O Bugs

- Incomplete Codesign
  - Comex's codesign bypass strategy in iPhoneOS 3
- iOS CS is enforced by AppleMobileFileIntegrity on page fault
  - But only for non-writeable pages
    - Due to W^X, this enforces code sign for all executable mappings
      - Just mark everything as non-executable and use various load commands to take over the program counter and ROP

Mach-O Bugs

- Apple's approach to fixing Comex's incomplete code sign was to enforce mach header as read/write/execute
  - But dynamic linker doesn't actually read the mach header from memory
    - It will pread() it into the stack raw off the file
      - No pagefault even if header marked R-X

Mach-O Bugs

- So after loading the Mach-O, Apple will actually read data from each segment to force a page fault
  - Fundamental TOC-TOU
    - Get code execution before dyld pagefaults
      - Memory corruption
    - Make dyld pagefault ineffective
      - Logic flaw: Overlapping segments

Overlapping Segments

- When LC_SEGMENT(_64) is parsed, a call to mmap() is issued with MAP_FIXED
  - If two segments describe the same virtual address, the second will “cover” the first
    - If we cover the segment containing load commands with a RW- map, dyld will fault in the page but AFMI will skip code sign checks
      - We’re back to using the incomplete code sign tricks

Overlapping Segments

- This class of bugs is easy to mitigate, but Apple failed to do so for 4-5 years
  - Good ‘ol days
- Integer overflows on overlapping checks over and over
  - Pangu and TaiG were huge fans of this :)

Memory Corruption

- Only practical, public memory corruption attack on dynamic library loading was published by me for iOS 8.4.1
  - Overlapped segment checks only applied between a Mach-O's own segments
    - By loading a non-PIE mach-O it was possible to overlap other mappings in the address space
      - However, before mmapping, the dynamic linker would try to reserve address space by calling vm_allocate, which fails on an already existing address
        - For the __PAGEZERO map, it's actually assumed that vm_allocate might fail, so we can skip this check

Memory Corruption

- We need to target some control data that can get us code execution before dyld pagefaults, or we crash (we have an executable segment as required by dyld and it's not covered by another mmap, so we'd crash if we tried faulting)
  - Stack was chosen for this
    - But we do run under ASLR, and stack is randomized
      - However we can just map a very large segment covering every possible stack randomization value
        - Bonus: it is possible to derive main binary ASLR slide by knowing stack ASLR slide

Memory Corruption

- Additionally it is possible to just steal signed pages from valid binaries (e.g. dyld) and map them at fixed vm addresses
- ROP is really easy as dyld statically links against libsystem (libc equivalent)

Mach-O Obfuscation Trick

- Lazy symbols will indirectly branch via global offset table (__got)
  - Initially, they point to dyld_stub_helper
    - Small stub compiled into every dynamically linked Mach-O
    - Will figure out which GOT entry you came from and invoke dyld to solve the symbol, then update GOT to cache result

Mach-O Obfuscation Trick

- You can redefine this in a .s file, and implement your own logic
  - Alter symbol loading logic to e.g. scramble symbol table, so the symbol you're linking against will not be the one that IDA thinks you are linking against
    - Bonus: IDA automatically hides dyld_stub_helper before version 7
  - Implement logic by writing a state machine that performs operations on each invocation, then make up a completely meaningless program that just branches a lot of times
- IDA will not be aware of this, and you can generate pretty cool-looking idbs

Loading Mach-O on PS4

- Context: We just pwned WebKit and FreeBSD and have full kernel r/w from JavaScript
- I want to load C-written code
- I hate ELF and am quite a fan of Mach-O
  - Wanna load Mach-O on PS4
    - Can just easily compile against MacOSX SDK
    - Will pre-convert from file representation to unbound/unrebased VM representation by expanding uninitialized data

Loading Mach-O on PS4

- Janky loader technique:
  - Define magic value in a assembly file
  - Scan for magic value, after magic value add branch to loading routine
  - Implement VM to parse binding opcodes in loading routine
    - Redefine dyld_stub_helper to invoke PS4 dlsym for lazy symbol resolution as per the previous obfuscation technique
    - Non-lazy symbols can be solved at runtime

Loading Mach-O on PS4

- This technique was used for my private PS4 jailbreak toolkit
  - Public toolkits use “normal” shell code and direct dlsym calls
- I also use this on iOS 11 to load Mach-O from WebKit RCE
  - _niklasb from phoenhex published a 11.3.1 webkit exploit recently using this strategy
    - He called it “beautifully implemented”
      - My code is kinda ugly so I am not sure I agree :p