Understanding and Defending Against Reflective Code Loading on macOS

Justin Bui

Introduction

This blogpost will describe the concept of loading executables in-memory on macOS. This technique is mapped to MITRE ATT&CK under ***Reflective Code Loading (T1620)***.

Over the many years I've used Cobalt Strike for red teaming, I've always appreciated the ability to load third-party software such as .NET assemblies and PowerShell scripts, through `execute-assembly` and `powershell/psinject/powerpick`, respectively. The ability to load third-party tooling on the fly greatly increases the flexibility of an agent. I decided to explore this concept on macOS to eventually incorporate into Hermes.

After introducing the concept of reflective code loading on macOS, I present my Swift implementation. We will explore how this technique differs on Big Sur/Monterey and I will explain my methodology behind digging into these differences.

Lastly, we will look into how reflective code loading can be detected, on Monterey, by monitoring for temporary files created during the "in-memory" loading process.

Executing Mach-Os In-Memory

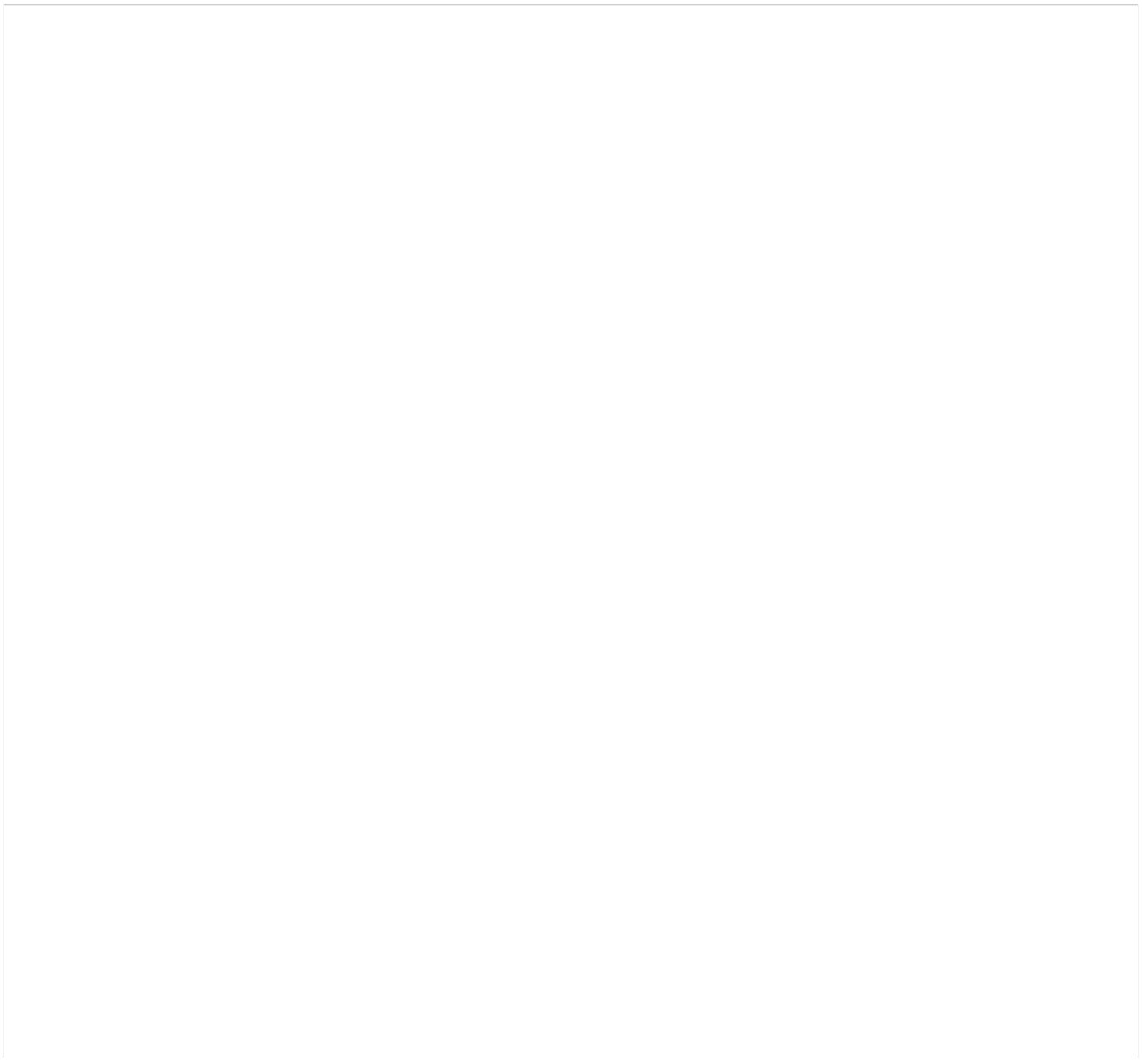
Note: Skip this section if you're familiar with reflective code loading on

macOS and want to dive into differences on how it works on Big Sur vs Monterey.

On macOS we can use the dynamic linker APIs to load two file types in-memory:

- Mach-O Binaries
- Loadable Bundles (.bundle)

The following macOS API calls can be used to execute and clean-up Mach-Os in-memory: `NSCreateObjectFileImageFromMemory()`, `NSLinkModule()`, `NSLookupSymbolInModule()`, `NSAddressOfSymbol()`, `NSUnLinkModule()`, `NSDestroyObjectFileImage()`.



Loading Mach-Os In-Memory with macOS API

Calling a Function

Before we can perform in-memory execution, we need to first load the binary or bundle into a region of memory. This can be done in a variety of ways such as loading the binary from disc, downloading the binary into memory, etc.

If the file being loaded is not a bundle, swap the `filetype` within the Mach-O header from `MH_EXECUTE` to `MH_BUNDLE`. This is necessary because `NSCreateObjectFileImageFromMemory()` checks that the file being loaded is a bundle. If the file being loaded is a bundle, no changes to the header are necessary.

```
NSObjectFileImageReturnCode NSCreateObjectFileImageFromMemory(
  const void*         address,
  size_t              size,
  NSObjectFileImage   *objectFileImage
```

```
);
```

`NSCreateObjectFileImageFromMemory()` takes the address of your binary in memory and will create an `NSObjectFileImage` structure.

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

`NSLinkModule()` will link the `NSObjectFileImage` into the program and return a module handle for it. This will add all shared libraries referenced by the module to the list of libraries searched.

```
NSSymbol NSLookupSymbolInModule(
  NSModule     module,
  const char*  symbolName
);
```

`NSLookupSymbolInModule()` takes the module handle returned from `NSLinkModule()` and searches for a symbol. We pass the function name we want to execute within the binary to `symbolName`.

```
void* NSAddressOfSymbol(
  NSSymbol  symbol
);
```

`NSAddressOfSymbol()` takes the `NSSymbol` returned by

`NSLookupSymbolInModule()` and returns the memory address of the symbol. Finally, we execute the address as a function pointer and pass any arguments to achieve code execution!

Executing the Entry Point of a Binary

Instead of searching for a particular function within a binary using `NSLookupSymbolInModule()` and `NSAddressOfSymbol()`, we can also search for and execute the binary's entry point. This technique was well documented by Stephanie Archibald here.

To find the binary's entry point, we can do the following:

1. Locate the base address of our Mach-O in memory
2. Find the entry point offset and add this to the Mach-O base address

To locate the Mach-O base address, we can use `NSLookupSymbolInModule()` + `NSAddressOfSymbol()` to search for `__mh_execute_header`, the address of the mach header in a Mach-O executable file type. This technique was used by Dwight Hohnstein here.

To locate the entry point offset, we walk the load commands and search for the `LC_MAIN` command.

We execute the address, obtained by adding the entry point offset to the Mach-O base address, as a function pointer to achieve code execution! Additionally, we can pass in command line arguments to our binary using `argc` and `argv`.

Cleaning Up

After executing our binary, we want to clean it up from memory using `NSUnLinkModule()` and `NSDestroyObjectFileImage()`.

```
bool NSUnLinkModule(
  NSModule  module,
  uint32_t  options
);
```

`NSUnLinkModule()` unlinks the specified module handle from the program and unmaps the image, but does not release the `NSObjectFileImage`.

```
bool NSDestroyObjectFileImage(
  NSObjectFileImage  objectFileImage
);
```

`NSDestroyObjectFileImage()` releases the `NSObjectFileImage` and frees the memory.

Further Operationalizing

Handling Universal Binaries

On macOS, there are two types of object files:

- Mach-O files (thin)
- Universal Binaries (fat)

Mach-O files hold object code for a single architecture while Universal Binaries can contain object code for multiple architectures (i386, x86_64, arm, arm64, etc.).

My initial proof-of-concept (POC) worked great for Mach-O files but not for Universal Binaries. John Baek did some awesome work adding Universal Binary header parsing to Stephanie's original POC.

Thin vs Fat Mach-Os

Using John's implementation, we could now load system binaries that already exist on disk such as `/sbin/ping`!

Retrieving stdout/stderr Using Inter-Process Communication (IPC)

Additionally, I wanted to retrieve output without direct access to the program's stdout/stederr, such as through C2. I looked into multiple methods but ultimately decided on named pipes for IPC for ease of use.

I start a named pipe server with a random name in `/tmp`, redirect stdout/stderr using `freopen()`, retrieve the output of the Mach-O loaded in-memory, and clean up the named pipe after.

Preventing In-Memory Mach-O From Exiting

> *Normally, when a Mach-O binary finishes execution, the program exits and returns back to the caller (like your terminal); however, this exit call, when called from your current process, will exit your loading process (https://github.com/djhohnstein/macos_shell_memory)*

To prevent the in-memory Mach-O from exiting, I copied Dwight's `atexit()` routine implementation.

1. Just before calling the program's entry point or function, we save a

`jump_buf` with `setjmp()`. This will allow us to restore the call stack later

2. We then create an `atexit()` routine that will `longjmp()` back to our saved `jump_buf`. This will hook any exit functions and jump back to our saved call stack, preventing the in-memory Mach-O from exiting. We also set a global Boolean to prevent execution of the entry point or function after restoring the call stack
3. Once the call stack is returned, the global Boolean is checked, the in-memory execution is skipped, and normal program execution continues

SwiftInMemoryLoading

Here's my Swift implementation for in-memory loading on macOS, it works on Big Sur as well as Monterey.

GitHub - slyd0g/SwiftInMemoryLoading: Swift implementation of in-memory Mach-O loading on macOS

Swift implementation of in-memory Mach-O loading on macOS - GitHub - slyd0g/SwiftInMemoryLoading: Swift implementation...

Executing `ifconfig` and `SwiftBelt` with Arguments In-Memory

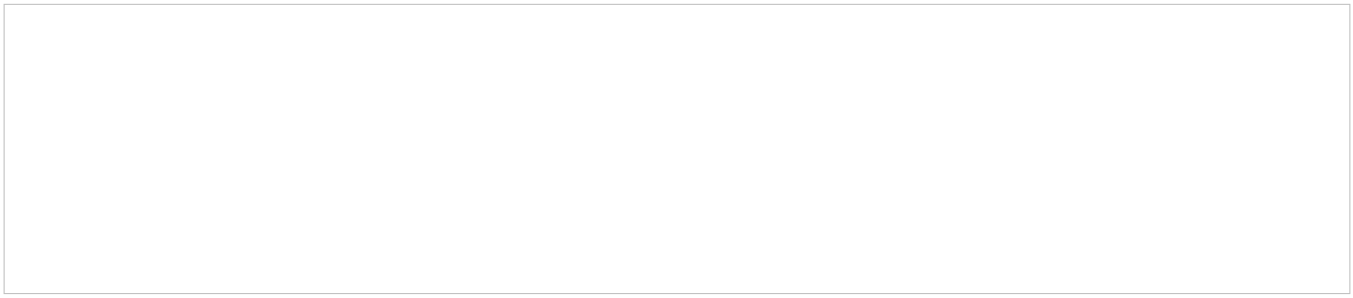
Reflective Code Loading: Big Sur vs Monterey

Segmentation Faults on Monterey

On Monterey, various red teams found that dereferencing the pointer returned by `NSLinkModule()` was causing a segfault. Timo Schmid and Carl Svensson discovered that this was due to a change in the API in more recent versions of `dyld`. They found that `NSLinkModule()` eventually calls `handleFromLoader()` which adds a flag at the end and performs a left shift of one byte on the return value.

In SwiftInMemoryLoading, this is handled by checking the operation system version with `ProcessInfo.processInfo.isOperatingSystemAtLeast()` and counteracting the left shift with a one byte right shift.

Checking OS Version



Right Shift if Monterey is Detected

“In-Memory”?

Attackers generally favor in-memory execution techniques because it can be harder to detect vs executing malicious programs on disc. I had heard that this technique was no longer file-less from multiple sources and wanted to confirm this. Let’s observe the difference on macOS Big Sur (11.6.5) and macOS Monterey (12.3.1).

On Big Sur, we can use `strings` to determine the version of `dyld`. We see that we are running `dyld-852.2`.

Big Sur Using dyld-852

Within the source code for `dyld-852`, particularly under `dyld3` we see that temporary file creation was introduced inside `NSLinkModule()`. Let's see if we can intercept that file creation.

NSLinkModule() Temporarily Writes In-Memory File to Disc

We executed SwiftInMemoryLoading and observe any potential file events with FileMonitor. We notice file events related to opening up the targeted binary and creation of the named pipe in `/private/tmp`, but nothing related to a temp file named `NSCreateObjectFileImageFromMemory-*****` like the source code implies.

No File Creations from FileMonitor on Big Sur

Running the program with a debugger, `lldb`, we see that there are two breakpoints for `NSLinkModule()`. Allowing the program to execute, we hit a breakpoint for `libdyld.dylib`NSLinkModule`.

Hitting Breakpoint in lldb on Big Sur

Since `NSLinkModule()` lives within `/usr/lib/dyld`, we can search for it with Hopper Disassembler. Comparing the recovered psuedocode, this matches the source code for `dyld`.

dyld Loaded Within Hopper

It appears that our binaries are using `NSLinkModule()` from the `dyld` namespace vs the `dyld3` namespace. In my testing, no temporary file creation occurred on Big Sur.

Let's repeat this process on Monterey!

On Monterey, we can use `strings` to determine the version of `dyld`. We see that we are running `dyld-955`.

Monterey Using dyld-955

Again, we executed SwiftInMemoryLoading with `lldb` and set a breakpoint for `NSLinkModule()`.

```
slyd0g-dev@slyd0g-devs-Mac Downloads % lldb SwiftInMemoryLoadi
...
...(lldb) sProcess 631 stopped* thread #1, queue = 'com.apple.
...(lldb) ddyld`dyld4::APIs::NSLinkModule:...
...0x10004b83c <+102>: movq   (%rdi), %rax0x10004b83f <+105>:
...0x10004b8b7 <+225>: leaq   0x3a558(%rip), %rsi       ; "NSC
...0x10004b8f6 <+288>: callq  0x10006a63c               ; pwri
```

Upon hitting the breakpoint, we step through a couple instructions and see that `dyld4::NSLinkModule()` is called! We also see hints of temporary file creation that we also saw in `dyld3::NSLinkModule()` source code.

Let's step through the program with `lldb` and see if we can spot the file creation with File Monitor.

Since we know the prefix to the temp file, we can `grep` for that specifically.

```
slyd0g-dev@slyd0g-devs-Mac Downloads % sudo /Applications/File
```

After a bit of stepping, we see the temp file get created. We confirm that the temp file created has the same hash as the path being passed to SwiftInMemoryLoading. A similar analysis has been done by @roguesys here.

Unfortunately, the technique presented above is no longer truly file-less on macOS Monterey. The file loaded into memory is written to disc during `NSLinkModule()` and then `dlopen()` is used to load it back into memory.

This can also be seen in the latest release of `dyld-940` here.

Detection

On Monterey, we can detect attempts at in-memory Mach-O loading by alerting on file creations with the predictable temporary file name (`NSCreateObjectFileImageFromMemory-*******`). These files will be written either to the value within the environment variable $TMPDIR or /tmp.

Conclusion

To recap, I learned about the necessary API calls to perform in-memory Mach-O loading on macOS and presented my Swift implementation.

I then explored this technique on Big Sur and Monterey and documented the differences. On Monterey, the file loaded in-memory is temporarily written to disc and then loaded with `dlopen()`. This presents us with a detection vector because the temporary file name and location is predictable:

- `/private/tmp/NSCreateObjectFileImageFromMemory-*******`
- `$TMPDIR/NSCreateObjectFileImageFromMemory-*******`

Thanks for taking the time to read this post, I hope you learned a little about macOS API, reflective code loading, and debugging on macOS!

If you see any errors please don't hesitate to let me know by e-mail (jbui006@ucr.edu) or Twitter (@slyd0g).

Credits

I realize this is not a new technique and wanted to explore it for my own knowledge, I am standing on the shoulders of giants and want to give credit where credit is due so big thanks to the following people/resources. Please see their prior work!

- Timo Schmid and Carl Svensson for discovering the changes in NSLinkModule's return value on Monterey
- John Baek for their work on Universal Binary header parsing
- Dwight Hohnstein for their `atexit()` routine concept and alternative method to bruteforcing addresses with `chmod` to find Mach-O headers in memory
- https://objective-see.com/blog/blog_0x51.html
- File Monitor

macOS reflective code loading analysis

Reflective code loading is an interesting attack technique, useful when there's a desire to conceal or protect the code...

Running Executables on macOS From Memory

As a security researcher, I'm always researching new and innovative ways that malware and attackers might exploit...

The Mac Hacker's Handbook

As more and more vulnerabilities are found in the Mac OS X (Leopard) operating system, security researchers are...

cctools/NSModule.3 at master · opensource-apple/cctools

typedef void * NSModule; extern NSModule NSLinkModule(NSObjectFileImage objectFileImage, const char *moduleName...

GitHub - djhohnstein/macos_shell_memory: Execute MachO binaries in memory using CGo

This is a CGo implementation of the initial technique put forward by Stephanie Archibald in her blog, Running...

GitHub - apple-oss-distributions/dyld

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

Parsing Mach-O files

This article describes how to parse Mach-O file and explains its format a little bit. It's not a definitive guide...

GitHub - its-a-feature/macos_execute_from_memory: PoC of macho loading from memory

I wanted to see how hard would it be to recreate in-memory macho execution used by Lazarus...