Reverse Engineering/Mac OS X

Apple Computer's Mac OS X is the standard Operating System used on Apple Macintosh computers. Other operating systems, primarily Linux, have been ported onto Mac Hardware, and there has been some effort to port OS X onto non-Mac Intel-based hardware, but neither of these efforts has attained the kind of popularity that the "standard bundle" has attained.

Mac OS X has been critically acclaimed by many people in the computer world as being both beautiful and easy to use. OS X is built on a BSD and Mach core but has a certain amount of software that is Mac-specific.

Try hard to keep this on the subject of general reverse engineering for Mac OS X, and not on 'cracking', or reversing only for security purposes. I have created special sections for these subjects, and all material focused on them should be kept there. Thanks! --Macpunk 04:17, 9 July 2007 (UTC)

Contents

Hardware Architecture

Historically before OS X Macs ran the Mac OS (http://en.wikipedia.org/wiki/Macos) operating system on the Motorola 68000 (http://en.wikipedia.org/wiki/68000) through the 68040 (http://en.wikipedia.org/wiki/Motorola_68040) and PowerPC (http://en.wikipedia.org/wiki/Powerpc) the architectures. Steve Jobs would later leave Apple to create NeXT (http://en.wikipedia.org/wiki/NeXT). After Apple had completed its hardware migration to the PowerPC platform it looked to a new kernel that could take advantage of this new hardware architecture. Many projects were started and failed and this and other factors led to the decline of Apple. In a move to capitalize on the new architecture it turned to Be Inc. (http://en.wikipedia.org/wiki/Be_Inc.) to purchase its new BeOS (http://en.wikipedia.org/wiki/Beos), this would later fall through as Be Inc. wanted too much money. Apple then turned to NeXT and acquired not only the NeXT OS but Steve Jobs. Steve Jobs would quickly take control of Apple and place the NeXT architecture as the replacement for Apple's aging Mac OS. The replacement product was 1st known as Rhapsody (http://en.wikipedia.org/wiki/Rhapsody_%28operating_system%29) which had the older Mac OS feel to it. Steve Jobs felt the interface did not do it justice so his team of ex-NeXT engineers developed Aqua (http://en.wikipedia.org/wiki/Aqua_%28user_interface%29) and Mac OS X was born.

Mac OS X 10.0 "Cheetah" (http://en.wikipedia.org/wiki/Mac_OS_X_v10.0) through 10.4.3 "Tiger" (http://en.wikipedia.org/wiki/Mac_OS_X_v10.4) would only run on the 4th and 5th generation of the PowerPC architecture. It became clear to Apple that IBM was having trouble with the 5th generation of the PowerPC known as the PowerPC G5 (http://en.wikipedia.org/wiki/PowerPC_G5) both in Development and Manufacturing. In addition IBM had yet to release a laptop version of the G5 process a year after it promised Apple it would. Apple then decided to migrate away from the PowerPC architecture and to an

Intel based one. Apple chose the Intel 32-bit Core Duo (http://en.wikipedia.org/wiki/Core_duo) architecture. Apple's second generation of Intel products appeared less than a year later running the Intel 64-bit Core 2 Duo (http://en.wikipedia.org/wiki/Intel_Core_2) architecture.

Apple originally included a Trusted Platform Module (http://en.wikipedia.org/wiki/Trusted_Platform_Module) (TPM) to help curb pirating of Mac OS X. Later, Apple would turn to a simple AES encryption system where the encryption keys were stored in a kernel device driver. This led to the ability to decrypt and even encrypt Mac OS X executable binary files. The new TPM system is no longer present in any modern Mac. The new encryption system is only available to the Intel based Macs and yields all sorts of errors if attempted on the PowerPC platform.

Apple has committed to supporting both PowerPC and Intel platforms for the next few years. Every Mac OS X system today ships with its binary files in a Universal binary (http://en.wikipedia.org/wiki/Universal_binary) format which can be ran in both PowerPC and Intel based Macs. The Universal binary is simply the source files compiled multiple times, (once for each architecture), and then glued together afterwards. When the OS reads this universal binary it will then select the proper version of that compiled code and execute it. Since not all binary files are Universal, Apple released for the Intel platforms a software component called Rosetta (http://en.wikipedia.org/wiki/Rosetta_%28software%29) which would dynamically translate PowerPC system calls to Intel system calls allowing the PowerPC binary to be executed on an Intel Mac.

Software Architecture

All builds of Mac OS X (OS X) are built on top of an XNU (http://en.wikipedia.org/wiki/XNU) kernel and Mach-O (http://en.wikipedia.org/wiki/Mach-o) file format. The XNU kernel is a Hybrid (http://en.wikipedia.org/wiki/Hybrid_kernel) kernel. The kernel is divided into 4 sections.

Kernel Sections

1. The Hardware Platform Expert
2. The Mach 3.0 Subsystem (OSFMK)
3. The BSD 4.4 Subsystem
4. The IOKit Subsystem and Framework

While the traditional Mach (http://en.wikipedia.org/wiki/Mach_%28kernel%29) kernel is a Microkernel (http://en.wikipedia.org/wiki/Microkernel), Apple has instead implemented its variation of Mach 3.0 with a Monolithic (http://en.wikipedia.org/wiki/Monolithic_kernel) design. The Mach subsystem is only a partial implementation of the Mach 3.0 kernel that was designed by Carnegie Mellon University (http://en.wikipedia.org/wiki/Carnegie_Mellon_University). This partial implementation consists of the Mach Messaging system, Mach Virtual Memory System and Mach Process Manager.

The BSD 4.4 Subsystem is a micro implementation of the FreeBSD 4.x kernel. Over time Apple has been shrinking and reducing the feature set of this kernel subsystem in the hope to eliminate all but the essential pieces to run BSD source code software on the XNU kernel. Originally the subsystem had support for BSD device drivers, which could communicate directly to hardware. Unfortunately the device driver architecture in the BSD Subsystem is only able to support direct main memory access and to interface into the UserMode (http://en.wikipedia.org/wiki/Usermode) part of a running process.

The IOKit (http://en.wikipedia.org/wiki/IOKit) Framework is a subset of the C++ programming language known as Embedded C++ (http://en.wikipedia.org/wiki/Embedded_C%2B%2B). The IOKit Subsystem drives the components written in the IOKit Framework. IOKit's purpose is to unify and simplify the Driver architecture while maintaining some level of compatibility between major and minor OS releases. IOKit has generally been a resounding success as some other BSD operating systems have ported or implemented an IOKit like system to it, (such as DragonFly BSD (http://en.wikipedia.org/wiki/DragonFly_BSD)).

The Hardware Platform Expert deals with the hardware differences of the PowerPC (G3 (http://en.wikipedia.org/wiki/PowerPC_G3), G4 (http://en.wikipedia.org/wiki/PowerPC_G4) and G5), Intel 32-bit, Intel 64-bit, Intel Xeon 64-bit (http://en.wikipedia.org/wiki/Xeon) (Mac Pro (http://en.wikipedia.org/wiki/Mac_Pro) and XServe (http://en.wikipedia.org/wiki/Xserve)) and ARM (http://en.wikipedia.org/wiki/ARM_architecture) (iPhone (http://en.wikipedia.org/wiki/Iphone)) architectures.

Commonly Used Tools

The common tools used to both compile/create the software and to disassemble/debug the software has been titled by Apple as the "Developer Tools". The developer tools can be found both on the Installation DVD for Mac OS X 10.4 and higher as well as the Apple Developer Connection (http://developer.apple.com/) (ADC) site. Joining ADC is free and is highly recommended. The ADC site has up to date documentation, tools and even sample source code. The ADC site should be your 1st place to do you research. A summary of the developer tools can be found at Apple's official XCode And Tools (http://developer.apple.com/tools/) website. The tools common used on the Mac OS X platform for reverse engineering besides the developer documents are found in the list below.

Developer Tools Used

1. gdb (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/gdb.1.html) (GNU Debugger (http://www.gnu.org/software/gdb/))
2. nm (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/nm.1.html) (Object File Symbol Table Viewer)
3. otool (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/otool.1.html) (Object File Display Tool)
4. fs_usage (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/fs_usage.1.html) (File System Monitoring Tool)
5. lsof (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man8/lsof.8.html) (File Descriptor Table Viewer)
6. vmmap (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/vmmap.1.html) (Virtual Memory Regions Viewer)
7. lipo (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/lipo.1.html) (Universal Binary Handler)
8. file (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/file.1.html) (Binary File Format Analyzer)

All of the above tools are installed during the Developer Tools Installation. As of current writting (3 Aug 2008) the current Developer Tools version is 3.1 (Build 2199).

Third party tools:

1. [1] (http://stevenygard.com/projects/class-dump/) class-dump is useful for parsing Objective-C runtime information.

Reversing Basics

Architecture

Since most target binaries that you wish to reverse engineer on the Mac OS X platform are in the Mach-O Universal Binary format you should decide which target binary platform you wish to reverse engineer. To get a list of what formats a specific binary has you would call the "file" program. Example:

A common example using the file "/bin/ls":

```
$ file /bin/ls
/bin/ls: Mach-O universal binary with 2 architectures
/bin/ls (for architecture i386):       Mach-O executable i386
/bin/ls (for architecture ppc7400):    Mach-O executable ppc
```

Another example, this time more of a rare one, using the file "/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices"

```
$ file /System/Library/Frameworks/ApplicationServices.framework/ApplicationServices
/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices: Mach-O universal binary with 4 arc
/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices (for architecture ppc7400):       M
/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices (for architecture ppc64):         M
/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices (for architecture i386):          M
/System/Library/Frameworks/ApplicationServices.framework/ApplicationServices (for architecture x86_64):        M
```

Symbols

Once you have identified the architecture you wish to use as your base for reverse engineering you would then dump the symbol table. This can be handy for the future. Example:

Common symbol table dump from the i386 architecture:

```
$ nm -arch i386 /bin/echo
         U ___error
00001000 A __mh_execute_header
         U _exit
         U _malloc
         U _strerror$UNIX2003
         U _strlen
         U _write$UNIX2003
         U _writev$UNIX2003
```

The above symbols can be broken up into 2 major categories:

Symbol Types

1. External
2. Internal

There is a 3rd category of symbols which are called "hidden" or "stripped" symbols. These symbols do not show up on *nm* and are hard to find out what they are doing and if they exist at all.
Each symbol type has a scope. The scope can either be private or public. In the past you could set the dynamic linker to a "flat namespace" which would convert the private symbols to public for your program only, however it has been reported that this functionality has been disabled on most libraries.
A private symbol is a symbol that is addressable by either the entire program or a section of the program only and can not be addressed by anyone else. A public symbol is one that is commonly known on other platforms as "Exported". The public symbols can be accessed by anything that links to that binary either at compile time or runtime.

Internal Symbols

Internal symbols are symbols that are defined within the program and thus are not imported, (dynamically linked), during runtime. An internal symbol can however be an external symbol that was linked in at compile time (http://en.wikipedia.org/wiki/Compile_time) and the source of that symbol was an object file or a static library. You can identify an internal symbol quickly because the line with the symbol has a hexadecimal (http://en.wikipedia.org/wiki/Hexadecimal) number before the symbol type letter. External symbols have a blank space where the number should be. The number specified in the symbol table denotes at what offset in the file that symbol's code or data starts at. This value is relative and WILL be different at runtime (http://en.wikipedia.org/wiki/Runtime). One way a symbol is located in memory during runtime is to find the relative positions of 2 symbols on the disk, 1st being a well known symbol and the 2nd being an unknown one, then extract the difference. Once you have the difference you can find the 2nd symbol in memory by simply apply the difference to the 1st symbols address. Example:

Example

Find the 1st and 2nd symbols:

```
$ nm /System/Library/Frameworks/QTKit.framework/QTKit
  /System/Library/Frameworks/QTKit.framework/QTKit(single module):
  {...}
  0005a638 T _copyBitmapRepToGWorld
  0008b017 t _createDisplayList
  {...}
```

In the above example our 1st symbol is "_copyBitmapRepToGWorld" which in a program is known as "copyBitmapRepToGWorld". Our 2nd symbol is "_createDisplayList" which in a program is unknown since its a private symbol, (See private symbols). Once the function definition for the symbol "_createDisplayList" can be determined then it becomes important to define that symbol for your program's use. To do this lets assume that "_createDisplayList" C function prototype would be:

```
void * createDisplayList(void);
```

The above prototype would be defined in the source code for the QTKit which is our target. That unfortunately doesn't help us since both the function prototype and the symbol name is unknown to our program. To resolve this problem we simple compute the difference from the above symbols, (the difference is 0x309DF), and define our function prototype as this:

```
void * (* createDisplayList)(void);
```

Then you would assign that function its address by having another function, (such as main), execute this command before you use that function for the 1st time:

```
createDisplayList = copyBitmapRepToGWorld + 0x309DF;
```

Some programs can get away with doing the above in 1 command outside of a function, I would NOT recommend this as the Mac OS X dynamic linker dyld (http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/dyld.1.html) sometimes will change the value of the symbol address before you enter your main function but after the variable's initial values have been defined.

External Symbols

External symbols are symbols that are defined elsewhere like in a library, (see library below). To read an external symbol you simply strip the leading "_" off. If the symbol has a "$" in its name then everything past the 1st "$" is a hint to the dynamic linker that this symbol is an explicate external symbol and should be matched with that exact version of the symbol in the external library. An explicit symbol is very helpful for a program creator since it allows him/her to make it difficult to override the symbol or to have a runtime link mismatch error. The letter to the left of the symbol name, (in the above example "U"), denotes the type of symbol such as function or data structure.

PowerPC

Basic instructions include li (load immediate) and mr (move register).

The Stack

The PowerPC stack works exactly as any other stack would. It's a LIFO structure, and it grows downwards(towards lower memory addresses). The most important detail to remember when reversing PowerPC binaries is that the PowerPC chip has no built in implementation of a stack. There's no register designated to keep track of where the bottom of the stack is, and there's no instructions to push and pop data off of the stack. Everything is done via a general purpose register, and various arithmetic instructions.

(This section will contain PowerPC specific information like how PowerPC function calls are executed, how arguments are passed to functions, the stack format, et cetera.)--Macpunk 06:19, 8 July 2007 (UTC)

Intel

(This section will contain Intel specific information like how Intel function calls are executed, how arguments are passed to functions, the stack format, et cetera.)--Macpunk 06:19, 8 July 2007 (UTC)

This page or section of the Reverse Engineering book is a stub. If you have information on this topic, write about it here.

Reversing for security

This page or section of the Reverse Engineering book is a stub. If you have information on this topic, write about it here.

Reversing for 'cracking'

This page or section of the Reverse Engineering book is a stub. If you have information on this topic, write about it here.

Further Reading

- Wikibooks: PowerPC Assembly
- A Brief Tutorial on Reverse Engineering OS X [2] (http://osnews.com/story.php/10366/A-Brief-Tutorial-on-Reverse-Engineering-OS-X)
- Cocoa Reverse Engineering [3] (http://www.woodmann.com/0xf001/osx/CocoaReverseEngineering.html)
- KellogS' Intro to OS X Reversing [4] (http://kellogsosx.blogspot.com/2007/05/essay-0-intro-to-os-x-reversing.html)
- A Non Practical & Non Real World Intro to Kracking for Mac OS X [5] (http://www.woodmann.com/0xf001/osx/kracking_osx.txt)
- What is Mac OS X?[6] (http://www.kernelthread.com/mac/osx/)

Special Notes

A large section of this document has been prepared and written by JosephC7, while this information has been granted for use by Wikibooks for free publication it should be noted that the author only asks that if you republish this information that you provide the author's user name and link to his user page on wikibooks.org. No fees is required or requested for this information and it is expected that if this information is republished that it too be given away freely with out compensation. This document is a work in progress and should be completed by the end of Aug 2008.