INDEPENDENT SECURITY EVALUATORS
Expert analysis. Customized solutions.

Hacking Leopard:

Tools and Techniques for Attacking the Newest Mac OS X

Charlie Miller

Jake Honoroff
Independent Security Evaluators

cmiller@securityevaluators.com

jake@securityevaluators.com

June 22, 2007

Abstract

This paper serves as an introduction to the tools and techniques available on the Mac OS X operating system for vulnerability analysis. It is particularly targeted for those security researchers already familiar with tools for Windows and/or Linux. It also reveals tools that are only found on Mac OS X and how they can be used to find security flaws, especially those that can be used in conjunction with fuzzing. Finally, it introduces a few tools recently ported from Windows to Mac OS X, pydbg and PaiMei.

Introduction

According to the Apple website, "Mac OS X delivers the highest level of security through the adoption of industry standards, open software development and wise architectural decisions." Of course, the Month of Apple Bugs and the flurry of activity after the release of Safari for Windows showed that Macs are just as susceptible to vulnerabilities as other operating systems. Arguably, two factors keep the number of announced vulnerabilities on Mac OS X low: the lack of researchers interested in exploring this operating system due to its low market share and the steep learning curve for researchers who are unfamiliar with the platform. The first of these reasons is going away as Apple's market share continues to rise. This paper hopes to address the second reason, that is, to provide researchers already familiar with Windows and Linux the knowledge and tools necessary to search for new security bugs in this operating system. Specifically, this paper will address the new forthcoming release of "Leopard", the newest version of Mac OS X. Happily, there are plenty of bugs and some Mac-only tools that help to find them.

Legal Disclaimer: Due to the fact Apple pushed back the release of Leopard, and the only releases of Leopard are available under NDA, this paper will avoid using screenshots or code directly from Leopard. Most of the information is not specific to Leopard anyway, besides the Dtrace examples, which are illustrated using Solaris 10.

Why Hack Macs?

While the market share of Apple computers running Mac OS X is still small, at roughly 6.5% of all operating systems, it is growing. The market share has gone up approximately 35% in the last year. This increasing market share is attracting more and more attention by computer researchers.

The growing interest in vulnerability analysis of Mac OS X can be seen in some recent developments.

The first major event occurred in January 2007 with the "Month of Apple Bugs". Two computer security researchers, LMH and Kevin Finisterre, released a vulnerability a day in Mac OS X or in applications that run in Mac OS X. The types of bugs reported were in a variety of programs and most appear exploitable. There were at least two exploitable default client-side remote exploits. Additionally, no less than five local exploits were announced that worked against default configurations.

A few months later, in April, at CanSecWest, the "Hack a Mac" contest took place. Attendees were encouraged to try to break into a default, patched MacBook, and if they did, they would win the computer. Eventually, a $10,000 reward was added by TippingPoint's Zero Day Initiative Program. The attack was successfully announced the next day.

Finally, in June, Apple released the Safari web browser for Windows. Within hours, at least 18 critical vulnerabilities were discovered in this application by numerous researchers including Aviv Raff, David Maynor, and Thor Larhom.

So whether you're interested in Apple's increasing market share, or you'd like to jump on the bandwagon, or you'd just like to shut up the local fanboy, this paper will get you

started with some of the basics of vulnerability analysis for the newest Mac OS X operating system, Leopard.

Hacker Friendly Features

Apple's Mac OS X is touted as the most user friendly operating system available. This is also true for security researchers wishing to test the security of the OS. Consider Safari, the default web browser for Mac OS X. Safari will automatically launch the following applications when it finds corresponding files on the Internet:

- Address Book
- Finder
- iChat
- Script Editor
- iTunes
- Dictionary
- Help Viewer
- iCal
- Keynote
- Mail
- iPhoto
- QuickTime Player
- Sherlock
- Terminal
- BOMArchiveHelper
- Preview
- DiskImageMounter

This list was obtained with the *RCDefalutApp* application. Therefore, while these applications can not always be given much user supplied data, a vulnerability in a program such as iPhoto, Preview, or QuickTime Player can be easily extended into a Safari exploit. Additionally, by default, Safari will open files associated with these programs (PDF's, MP3's, WAV's, ZIP's, etc) without prompting or warning the user. If that is not enough, by default, Safari allows pop-ups, Java, and Javascript, all by default.

Another nice thing about Mac OS X is that some of the source code is available, http://developer.apple.com/opensource/index.html. This includes most of the standard Unix open source tools, as well as Webkit, the html parsing engine for Safari - more on this later.

Another friendly feature of Mac OS X is that it allows users to do some system configuration that they normally wouldn't be allowed to do. This includes over fifty setuid root programs on a default install of Mac OS X. Most of these you would have probably never heard of, including:

- Locum
- NetCfgTool
- afpLoad
- TimeZoneSettingTool
- securityFixerTool

while others include familiar files with unfamiliar file permissions,

- netstat
- top
- ps

Most of the files in this last group have not been setuid root on Linux systems in many years.

Gotchas

While Macs have some great development tools, they do lack some traditional tools used in Linux analysis. There are usually equivalent tools available, though, as we'll see.

First, there are differences in naming conventions. In Mac OS X, shared objects, a.k.a. dynamic libraries, have the file extension DYLIB. Device drivers, a.k.a. kernel modules, have the file extension KEXT. Finally, the Mac OS X applications reside in /Applications, not /bin or /usr/bin. Furthermore, the actual binary will have a pathname of something like

```
/Applications/Preview.app/Contents/MacOS/Preview
```

Probably the first thing someone with a Linux background will notice when they start to look at binaries is there is no *ldd* command. The Mac OS X equivalent command is *otool*, see below.

```
$ otool -L /bin/ls

/bin/ls:

    /usr/lib/libncurses.5.4.dylib (compatibility version 5.4.0, current version 5.4.0)

     /usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 1.0.0)

     /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.1.5)
```

Likewise, there is no *strace* or *ltrace*. The command to use is *ktrace* (or *dtrace* in Leopard. More on that in a later section).

```
$ ktrace -tc w

...

$ kdump

       4087 ktrace   RET   ktrace 0

       4087    ktrace           CALL execve(0xbffff3cc,0xbffff990,0xbffff998)
```

Until February 2007, IDA Pro did not support Universal Binaries, the type of file used by Mac OS X applications. This made disassembly very difficult. Luckily, it works now!

Another point to remember is that Mac OS X shellcode has a couple of restrictions. First, you cannot call *execve()* until you call *vfork()*. Also, despite popular belief, it appears you must call *vfork()* instead of *fork()*.

One of the biggest bummers regarding Mac OS X is that *ptrace()* is hopelessly broken. For example, it doesn't support *PTRACE_PEEKUSR*, *PTRACE_GETREGS*, etc. This makes writing a ptrace-based debugger impossible. Instead, use the Mach API, the interface into the Darwin kernel. This is how *gdb* and pydbg both work.

The way the heap is handled is different than in Linux and Windows. This, of course, has a large impact on heap overflows in this operating system. The Mac OS X heap is composed of *zones*, which are variable size portions of virtual memory, and *blocks* which are allocated within these zones, see " OS X Heap Exploitation Techniques". The document is a must read. However, since 10.4.1, the large zone is no longer located at an address smaller than the tiny zone, which makes the overall exploitation method discussed in that paper not work as advertised. So unlike most OS's, there is not a lot of heap management pointers available on the heap for overwriting. Instead, typically, you need to overwrite application specific data.

One last caveat is that on Mac OS X, *LD_PRELOAD* is replaced by *DYLD_INSERT_LIBRARIES*. You'll need this if you want to use sharefuzz or perform something similar to Electric Fence.

Fuzzing Tools

With *ptrace()* broken, what types of fuzzing tools are available for Mac OS X?

While the applications and development tools mentioned throughout this paper were readily available in Mac, there wasn't much in the

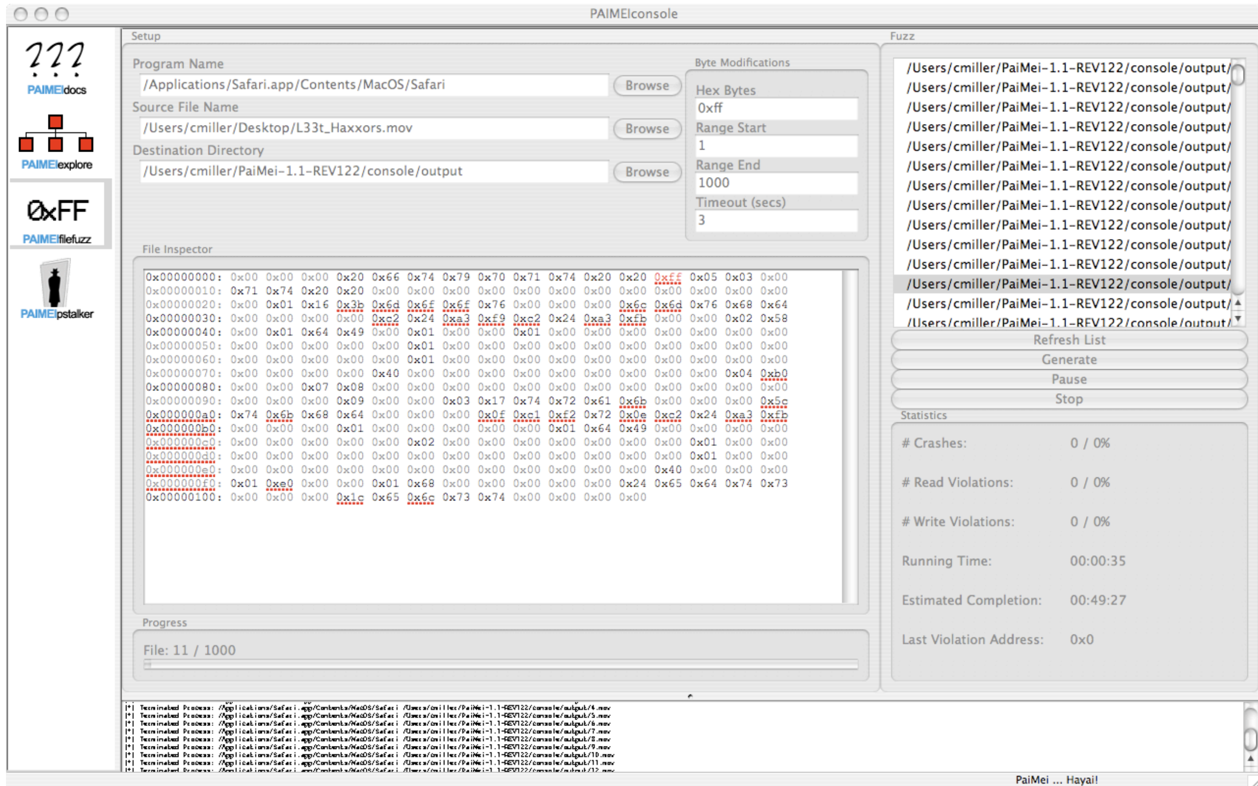


Figure 1: PAIMEIFile fuzz for Mac OS X in action

way of fuzzers. However, this paper introduces two recent Windows tools, pydbg and PaiMei, that are now available from the PaiMei website.

pydbg is an open source tool that enables a researcher to perform a number of useful actions, all from within the Python programing language. For example, pydbg can be used to monitor a program for exceptions or crashes, to take memory snapshots of the process, or as a general purpose debugger. pydbg was written originally for Windows, but this tool is now available for Mac OS X in a beta release. Most pydbg features are supported. Using pydbg, it is relatively easy to write fuzzing and fuzz monitoring tools.

Likewise, PaiMei is a reverse engineering framework, until now only available for Windows. It is built upon pydbg and allows for process tracing and file fuzzing while recording all data in a back end database. Furthermore, it can output data to graphing programs or to IDA Pro. PaiMei is now available for Mac OS X in a beta release from the PaiMei website.

Mac Specific Tools and Techniques

Up to this point, we've discussed various difficulties for security researchers using Mac OS X and how to get around these problems. However, this is only half the story. There are a number of features of Mac OS X that can significantly help researchers discover vulnerabilities, especially when fuzzing. Most of these are debugging facilities provided to help debug end user problems, but security researchers can use them as well.

Debugging Symbols

The first is the ability to use debugging symbols from the Mac OS X frameworks. These libraries contain extra asserts and produce

verbose output. This output will be on the system console if launched through a standard way, i.e. double clicking on it. Alternatively, it will appear in the terminal window if launched from within a terminal window. These additional assertions can be used to help identify when problems in the application arise as early as possible, something nice to do when fuzzing, for example.

Core Dumps

Core dumps can be useful while fuzzing. They can be enabled globally by editing the */etc/launchd.conf* file. Or, within a terminal, they can be enabled by using the *ulimit* program. Core files generated by the system can be found in /cores.

Environment Variables

There are a number of interesting environment variables that can be used for memory allocation.

Variable	Description
MallocScribble	If set, free() sets each byte of released memory to 0x55
MallocPreScribble	If set, malloc() sets each byte of newly allocated memory to 0xAA
MallocGuardEdges	If set, adds guard pages before and after large memory allocations.
MallocCheckHeap	The number of allocations until malloc() begins validating the heap.
MallocCheckHeapEach	The number of allocations between heap validations.
INIT_Processes	If set, delay launch of applications by 15 seconds.

Setting these environment variables can be extremely useful during fuzzing. The main benefit they provide is quickly identifying inputs that have corrupted memory. A couple of common problems when fuzzing is that the heap may become corrupt, but unless the application uses the corrupted memory, it will not crash. This makes finding off-by-one errors especially difficult. Another problem is a program that crashes well after the memory corruption occurs. This makes finding the vulnerability time consuming and difficult. These environment variables can help alleviate these problems.

For example, you can have *malloc()* check the heap integrity every so many allocations. Additionally, using MallocScribble will help detect double free situations when they arise. Using MallocGuardEdges is similar to using Electric Fence for Linux and will identify buffer overflows when they occur. Using these variables helps make fuzzing more efficient and effective.

Another tool, which is similar to the MallocGuardEdges environment variable is the Guarded Memory Allocator, libgmalloc. This is analogous to Electric Fence and is started in a similar fashion,

```
DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib
TextEdit
```

Command Line Tools

There are a number of tools available that can be used to monitor applications, similar to the suite of tools available from sysinternals for Windows. Below are some of the most useful,

fs_usage: Records file system access for a given process. Similar to a non-GUI filemon.

sc_usage: Records system call information for a given process.

vmmap: Dumps virtual memory for a process.

heap: Gives information about the heap of a process

malloc_history: A tool that can either monitor all memory allocation/deallocation or can log memory allocated/deallocated at a specified address.

lsof: The standard UNIX tool.

Using these tools, information can be gained from a running process that can help identify files being used as well as observing how memory is being used.

CrashReporter

Much of the fuzzing literature discusses the need to attach a debugger to the target application in order to monitor when it crashes. In fact, tools like SPIKE, PaiMei, and FileFuzz all do exactly this for their various platforms. Apple was kind enough to let us not worry about such things, thanks to CrashReporter.

Figure 2: Standard CrashReporter dialogue

When an application crashes, CrashReporter will record the crash in the system log (`/var/log/system.log`) and put details of the crash in the crash log (`/Library/Logs/CrashReporter/<ProgramName>.crash.log`). This crash log includes the application name, pid, exception information, context information, and a backtrace. If it is a GUI application, it presents you with a dialog box like in Figure 2.

Crashreporter can be configured using CrashReporterPrefs application to allow the option to attach to the process with GDB after it crashes, see Figure 3. Also, note that this will run the commands from your .gdbinit file, too.

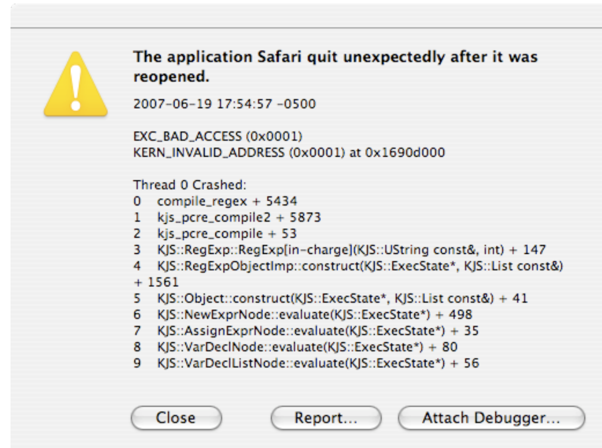


Figure 3: Developer information incorporated into CrashReporter

Source Code

Having source code can make analysis easier when it is available. For example, applications can be rebuilt with debugging symbols or instrumented to provide code coverage.

For Mac OS X, source code is available for the kernel, any open source software included, and WebKit, the HTML engine used in various applications, including Safari, Mail, and Dashboard.

For example, to build WebKit to report code coverage information, simply add the -coverage flag,

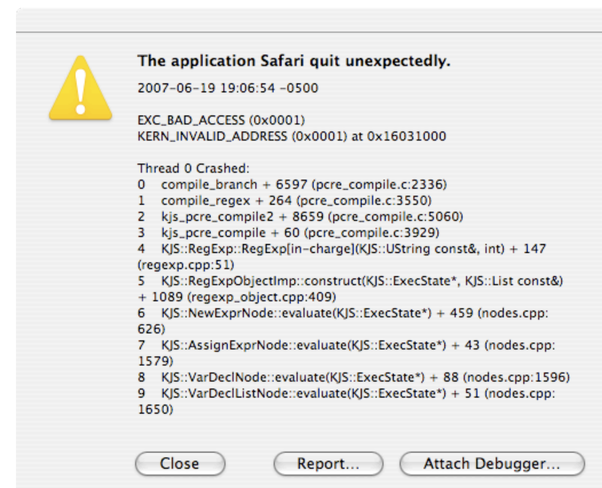
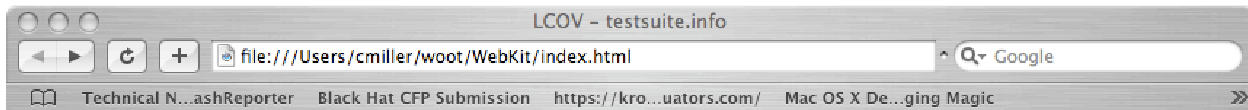


Figure 4: CrashReporter with a debug version of WebKit.



LTP GCOV extension - code coverage report

Current view:	directory		
Test:	testsuite.info		
Date:	2007-06-01	**Instrumented lines:**	13622
Code covered:	59.3 %	**Executed lines:**	8073

Directory name	Coverage		
/System/Library/Frameworks/CoreFoundation.framework/Headers		100.0 %	1 / 1 lines
/System/Library/Frameworks/JavaVM.framework/Headers		0.0 %	0 / 53 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/API		0.0 %	0 / 474 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings		0.0 %	0 / 530 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/c		0.0 %	0 / 190 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/jni		0.0 %	0 / 890 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/objc		0.0 %	0 / 476 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/kjs		79.3 %	5723 / 7219 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/pcre		54.7 %	1338 / 2445 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/wtf		0.0 %	0 / 56 lines
/usr/include		100.0 %	2 / 2 lines
/usr/include/architecture/i386		100.0 %	3 / 3 lines
/usr/include/c++/4.0.0/bits		50.0 %	4 / 8 lines
/usr/share		89.7 %	96 / 107 lines
JavaScriptCore/kjs		84.8 %	357 / 421 lines
kjs		0.0 %	0 / 39 lines
wtf		76.9 %	528 / 687 lines
wtf/unicode/icu		100.0 %	21 / 21 lines

Generated by: LTP GCOV extension version 1.5

Figure 5: LCOV generated code coverage information for WebKit.

```
WebKit/WebKitTools/Scripts/build-webkit -cov-
erage
```

After fuzzing Safari or any other WebKit enabled application, code coverage can be viewed using lcov, or a similar package, see Figure 5.

Also, don't symbols make the crash reported earlier much nicer to read (see Figure 4)?

Let Robots Do the Work

Another great way to fuzz is using the Automator application. This application allows you to perform repetitive actions automatically, exactly what we want for fuzzing! There are many built-in actions that can be used in order to piece together something useful. Additionally, arbitrary Applescript can be executed. Most Mac OS X applications have embedded code that allows Applescript to access almost any part of the application for automation, i.e. menu selection, button presses, etc.

For a simple example, we'll build a Preview fuzzer. Simply start automator, drag a few actions into the workflow,

- Ask for Finder Items

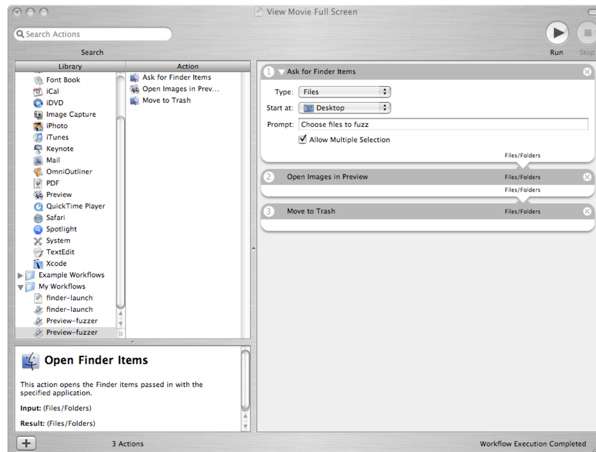


Figure 6: The workflow for a simple Preview fuzzer

- Open Images in Preview
- Move to Trash

After this, select File->Save As and choose Application. Next, create a bunch of fuzzed files that Preview will read. Finally, launch the fuzzer you built, select the files, and let the thing run!

Dtrace

One of the most exciting additions to be introduced in Leopard is the inclusion of Dtrace, originally for Solaris. Dtrace is a dynamic tracing mechanism built directly into the kernel and many of the applications. Dtrace uses the language D, a subset of the common C programming language. Programs, or traces, can be written that can be used to dynamically instrument any application running on Mac OS X.

Dtrace works because the operating system has a set of probes located throughout the kernel. Dtrace can bind an action to each of these probes. As each probe fires, the data is returned to Dtrace for reporting. Best of all, Dtrace is designed such that inactive probes or probes that are not firing cause no slowdown to the application or operating system.

What can security researchers use Dtrace for? One of the first things that a researcher needs to known when assessing an application for potential vulnerabilities is what code is executed. The application may consist of a binary and numerous libraries. Using Dtrace, the researcher can quickly identify the components that are affected by user controlled input and can get code coverage or instructions traces with little difficulty. It is also possible to monitor applications access to file system, network, or other resources.

A set of simple Dtrace programs is provided in the Appendix. The first one emulates the windows program Filemon by monitoring and recording access to the open, close, read, and write system calls. The next sample program is analogous to the Sharefuzz environment variable fuzzer. It simply returns long strings when programs attempt to call getenv(). Such a program is literally 3 lines of D. Next, a sample program that behaves like the *ltrace* library tracing program is provided. Additionally, a program that can do instruction traces in a target program is given. Such a program could be automatically produced from an IDA Pro plugin. Finally, a Dtrace program is given that records and prints code coverage in a target program.

Leopard also comes with a new application called Xray. This allows tools, including Dtrace applications, to display their results in real time using a timeline editor similar to GarageBand. This customizable application could be configured to be a fuzzing control panel!

Conclusions

As Mac OS X becomes more prevalent, it also becomes a more obvious target for security researchers. However, for a researcher coming from the Windows or Linux world, there can be a steep learning curve to

Figure 7: Advertisement for X-ray.

find the equivalent tools that they are accustomed to using. By using the tools outlined in this paper, which include both Mac versions of well-known tools and some Mac OS X exclusive tools, security researchers can comfortably focus their efforts on this platform.

References

Mac OS X internals: A Systems Approach, Amit Singh, Addison Wesley, 2006

Apple - Mac OS X - Leopard Sneak Peak: http://www.apple.com/macosx/leopard/

Mac OS X Debugging Magic: http://developer.apple.com/technotes/tn2004/tn2124.html

CrashReporter: http://developer.apple.com/technotes/tn2004/tn2123.html

AppleScript: http://www.apple.com/macosx/features/applescript/

Automator: http://www.apple.com/macosx/features/automator/

PaiMei: http://pedram.redhive.com/PaiMei/docs/

Abusing Mach on Mac OS X: http://www.uninformed.org/?v=4\&a=3\&t=pdf

DTrace to be included in Next Mac OS X: http://sun.systemnews.com/articles/102/2/news/16842

DTrace User Guide http://docs.sun.com/app/docs/doc/819-5488

MacPython OSA Modules: http://www.python.org/doc/2.3.5/mac/scripting.html

Fuzzing Software Tools // iDefense Labs: http://labs.idefense.com/software/fuzzing.php\#more_spikefile

Kernel Programming Guide: Mach API Reference: http://developer.apple.com/documentation/Darwin/Conceptual/

OS X Heap Exploitation Techniques http://felinemenace.org/papers/p63-0x05_OSX_Heap_Exploitation_Technqiues.txt

Hack a Mac, get $10,000 http://news.com.com/8301-10784_3-9710845-7.html

Safari for Windows: Released and hacked in a day http://www.infoworld.com/article/07/06/11/Safari-for-Windows-released-and-hacked-in-a-day_1.html

Trends in Mac Market Share http://arstechnica.com/journals/apple.ars/2007/04/05/trends-in-mac-market-share

With Windows port, a bug-hunting Safari for Apple http://www.infoworld.com/article/07/06/12/With-Windows-port-a-bug-hunting-Safari-for-Apple_1.html

Mac OS X PPC Shellcode Tricks http://uninformed.org/?v=1&a=1&t=pdf

Appendix

Filemon in dtrace

```
/*
 *  dtrace -qs filemon.d <PID>
*/

syscall::open:entry,
syscall::open64:entry
/pid == $1 /
{
        printf("%s(%s)", probefunc, copyinstr(arg1));
}

syscall::open64:return,
syscall::open:return
/pid == $1 /
{
        printf("\t\t = %d\n", arg1);
}

syscall::read:entry,
syscall::write:entry
/pid == $1/
{
        printf("%s(%d, %s, %4d)", probefunc, arg0, stringof(copy-
in(arg1,arg2)), arg2);
}

syscall::read:return,
syscall::write:return
/pid == $1/
{
        printf("\t\t = %d\n", arg1);
}

syscall::close:entry
/pid == $1/
{
        printf("%s(%d)\n", probefunc, arg0);
}
```

sharefuzz in dtrace

```
/*
 * dtrace -wp `pgrep test` -s sharefuzz.d
*/

pid$target::getenv:return
/execname == "test"/
{

copyoutstr("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
, arg1, 128);
}
```

ltrace in Dtrace

```
/*
 * dtrace -qp `pgrep test` -s ltrace.d
*/

pid$target:::entry
/execname == "test"/
{
        printf("%s", probefunc);

}

pid$target:::return
/execname == "test"/
{
        printf("\t\t=%d\n", arg1);
}
```

Instruction tracer in Dtrace

```
/*
 * dtrace -qp `pgrep test` -s trace.d
*/

pid$target:a.out:foo:*
/execname == "test2"/
{
        printf("08%x\n", uregs[R_EIP]);
}
```

Code coverage using Dtrace

```
/*
 * dtrace -qp `pgrep test` -s code_coverage.d
*/

pid$target:a.out:foo:*
/execname == "test2"/
{
        @code_coverage[uregs[R_EIP]] = count();
}
```