Mac OS X Malware Vulnerabilities (November 2008)

Michael Grebenyuk, Albert Sodyl, Byron Thiessen, Sijia Wang

Abstract—**Mac OS X is commonly known to be a secure operating system however this is a misconception. Mac OS X does have security flaws and a proof-of-concept virus was created by the authors to exploit universal binary files. This virus infects other binaries on the system and captures private user data, emailing it to the attacker. There are several possible defence strategies to defend against exploits such the one presented. These include prevention using stricter access control with sandboxing, and detection by various methods such as signature detection, change detection, and activity scanning.**

Index Terms—**Computer security, computer viruses**

I. Introduction

Apple advertises their Mac OS X operating system as being very secure through their marketing campaigns. A quote from their website claims that it "delivers the highest level of security through the adoption of industry standards, open software development and wise architectural decisions. Combined, this intelligent design prevents the swarms of viruses and spyware that plague PCs these days" [1].

As a result of this, and the lack of prevalent malware on the platform, many Macintosh users believe that Mac OS X really is secure. Advertisements such as the popular Mac vs. PC commercials only contribute to the hype, even for non-Macintosh users. Therefore, most are oblivious to malware threats on Mac OS X. However, the security of Mac OS is a myth, as it has vulnerabilities just like other operating systems, and should exploits become more common, it would significantly increase the risk of using a Mac OS X [2], [3].

This project analyzes how effective the security of Mac OS X really is when it comes to malware, such as viruses and spyware. By presenting a working exploit in the form of a Mac OS X virus that does alarmingly malicious actions, we hope to gain the attention of Mac OS X users and provide insight on how Mac OS X is just as vulnerable as other operating systems, and present strategies for defending against such threats.

II. Binary Formats

A. Mach-O Format

Operating systems use various object formats to store executable code and their libraries and associated metadata. They act as blueprints for mapping out processes images in memory. Object files are created by compilers or assemblers and read by the operating system loader for that object format. Various object formats exist, such as ELF, PE/COFF, and Mach-O.

The Mach-O (Mach Object) object format is used by operating systems based on the Mach kernel, such as NeXTSTEP and Mac OS X. The Mach-O format provides both intermediate and final storage of the machine code and data. Features for dynamic linking were later added on top of the statically linked executable code at runtime, resulting in a single file format that is both dynamic and statically linked. The basic Mach-O file contains three major regions (Fig. 1).

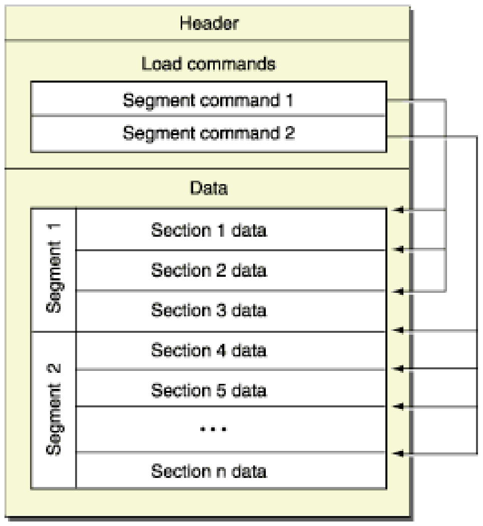

Fig. 1. Mach-O file format basic structure. The 3 main regions are header, load commands and sections. Each segment command has zero or more section commands associated with it.

At the beginning of every Mach-O file is a header *mach_header* that identifies the file as a Mach-O type using a magic number *0xfeedface*. The header also contains information such as the CPU type, file type, total size of the load commands and various other flags that will affect the interpretation of the rest of the file. The *mach_header* structure is as follows:

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

Common CPU types supported include PowerPC (0x12) and x86 (0x7). CPU subtypes define specific models of processors and are usually 0x0 for PowerPC and 0x3 for Intel binaries.

Following the header are a series of load commands that specify the layout and linkage characteristics of the file.

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

This structure specifies the initial layout of the file in virtual memory, the initial execution state of the main thread of the program as well as location of symbol tables and names of shared libraries. The most important load commands are LC_SEGMENT, LC_THREAD, and LC_UNIXTHREAD. In particular, LC_THREAD holds the initial state of the registers when a thread starts, this *load_command* can be used to retrieve or modify the entry point of a thread.

The third section of the Mach-O is the Data section which is divided into one or more segments each containing zero or more sections pointed to by an LC_SEGMENT load command (Fig. 1). Each segment defines a region of virtual memory that the dynamic linker maps into and each section of a segment contains code or data of a particular type. Such information includes symbol tables and string tables.

B. *Fat Binaries and Universal Binaries*

Multiple Mach-O files can be combined into a multi-architecture binary. A binary file as such is also called a fat binary due to the increased size of the file. Fat binaries include machine code for multiple instruction set architectures (ISAs) and thus can be run on multiple processor types. The fat binary was developed by NeXT for their NeXTSTEP operating system to allow Motorola 68k and Intel IA-32 based processors to run the operating system without making separate versions. After Apple purchased NeXT in 1996, fat binaries continued to be supported in Mac OS X, but it wasn't until Apple announced their transition to Intel processors in 2005 that they really took off. Fat binaries are not actually binary files themselves but rather an archive of binaries.

Every fat binary starts with a structure *fat_header* similar to that of a single Mach-O file which contains two unsigned integers *magic* and *nfat_arch*. The magic number is similar to that in a Mach-O file since it also identifies the file as a fat binary archive. However, for a fat binary the magic number used is *0xcafebabe*. The second unsigned integer is used to define how many Mach-O binaries are contained within the archive. In Mac OS X, fat binaries are used to support optimized codes for multiple variants of an architecture such as 32-bit and 64-bit PowerPC generations.

Later, Apple promoted the use of fat binaries through their transition from PowerPC to Intel's x86 architecture, giving fat binaries the name "Universal Binary". It is interesting to note that PowerPC is a big endian (higher order bytes stored at lowest address) architecture, and Intel uses little endian (lower order stored in memory at the lowest address). However, the *fat_header* structure is big endian on both architectures, thus forcing a byte swap on Intel machines. The *fat_header* structure also contains information such as the archive offset which allows the loader to take the native binary required for certain types of architecture and run it as if it was a single Mach-O. Because the native code is being run, it executes just as fast as a pure native Mach-O file. The only drawback to this is the universal binary's size. As mentioned before, universal binaries are simply archives of two Mach-O files which generally are double in size of a single Mach-O. However, this drawback doesn't affect the end user significantly since file storage sizes have grown an astounding amount.

III. Infecting Mac OS X Binaries

A. *Concatenation Method*

Apart from utilizing specific Mach-O binary features, a simple way of infecting binaries is with the concatenation method. This method works by concatenating two Mach-O executable objects together. When the result is executed, only the first binary will "run". To use this as an exploit, a binary needs to be created that knows its own size and contains a potentially harmful payload within. We will call this binary the 'parasite'. To infect a host binary, it simply inserts itself in the beginning of the host binary. In other words, it concatenates the host binary with itself, the copies itself to the location of the host binary, overwriting it. When the infected host binary executes, the parasite must seek to the end of itself, copy the rest of the binary, which is the host program, into a temporary file and execute it as a child process. Then it executes whatever payload desired either before, during, or after the host binary is run. [6]

Such a method is trivial to implement in Mac OS X because most binaries are writable and it is possible for a process to open a file descriptor to its own binary. Contrast this with Linux where /proc/pid/mem must be used. The location of the temporary file does not matter for all but some Objective-C applications that use *@executable_path* where the temporary binary must reside in the same folder as it was originally in. The arguments and environment must be passed unmodified when executing the temporary file in a fork so the application has the illusion that its first argument (the location of the binary), has not changed.

B. *Resource Fork Method*

Mac OS X file system uses HFS+, a replacement for Apple's Hierarchical File System(HFS). Each file on a HFS+ file partition has two forks, data and resource. To access a file's resource fork, the command *<filename>/rsrc* or *<filename>/..namedfork/rsrc* is used. To use this for an exploit, one can copy the host binary into the resource fork of the parasite file. The parasite file is then copied over top of the original host binary, replacing it. When the new binary is executed, it simply executes the payload section before or after executing its resource fork. This gives the illusion of the host binary being run unmodified.

A limitation with this approach is that it only works with the HFS file system. Since much data is received from CDs, DVDs, the internet, and non-Mac OS X servers, this approach could have difficulties spreading through certain mediums. Luckily, the built in archive utility for creating /extracting ZIP files in Mac OS X supports preserving resource forks, as well as tar and other UNIX utilities when ran on a Macintosh system. Similar constraints to arguments and environment variables exist for this method as with the concatenation method.

C. *Hook Mach-O Entry or Exit Points*

In the Mach-O binary, the entry point for the initial thread can be found in a LC_THREAD or LC_UNIXTHREAD load command. The struct for this command contains an additional struct (*cpu_thread_state state*) that stores the initial state of each register. A free tool called HTE <http://hte.sourceforge.net> can be used to manipulate object files and headers trivially, supporting code disassembly as well. It can be used to view the srr0 field in the Mach-O header and modify it to the specified address of code execution. Of course this can be automated to define the entry point of parasite code or shellcode beginnings.

Since changed entry points can be easily detected by antivirus software, other entry points can be used. For example the *__DATA,__mod_init_func* sections for C++ applications can be used as hooks, of which that have been compiled with g++ can use *__TEXT,__contructor* and *__TEXT,__destructor* sections, even if they don't use them. These can be used as entry or exit points and store malicious code. These places make it difficult for antivirus software to detect modified binaries since they look like valid code, compared to an LC_THREAD construct pointing to a strange memory address.

The approach used by the MachoMan virus uses LC_UNIXTHREAD to modify the EIP register (on Intel), to jump to segments which contain no sections (like *__LINKEDIT* and more importantly *__PAGEZERO*). This makes code loaded into *__PAGEZERO*, which is usually not accessible by the loader, to be run. It provides sufficient room for malicious code and is invisible to the loader, making it harder to detect binary execution flow alterations.

IV. Felcator Virus

A. *Actions*

The Felcator virus is a proof-of-concept virus for Mac OS X created by the authors of this paper. It is slightly more than proof-of-concept as it does have a malicious payload. Using the concatenation method, it recursively infects all universal binaries in the user's Application directory (~/Applications) when executed. The payload shows a pop up dialog that the file is infected, along with printing that message to stdout. It also captures the user's bookmarks, website history, recently downloaded files list, information about the operating system and network configuration, as well as all the user's keychains. The keychains in Mac OS X store all passwords and possibly other sensitive data such as credit card numbers. The keychain is by default encrypted with the user's login password. The login password is not captured by the virus, but easily could be through key logging or reading unencrypted swap files as the root user. The captured information is then emailed to a specific email address. The virus does this every time an infected application is run.

B. *Mechanism*

The way the virus works is quite simple. The virus binary is compiled as a universal binary and inserted in the beginning of the desired binary to be infected manually. This is accomplished with a Perl script that compiles the virus code once to get the resulting binary size, then modifies the source code with the updated file size, and compiles it again. Compilation is done with the help of the *make* utility and the result is a binary that knows its own file size. The Perl script then copies the virus code to a temporary file, appends the desired host binary to it, and then overwrites the host binary with the temporary file. This process is known as virus code bootstrapping. Once the virus infected a living binary, it can then reproduce on its own.

When the infected binary is run, the virus code's main function is executed. The virus starts by seeking to the end of itself and copying the rest of the binary, which is the untouched host binary, to a temporary location. This location was originally in /tmp, but later revisions of the virus chose to use the same directory as the virus was in. The reason for this is that many Cocoa applications load dynamic libraries that expect them to be relative to the location pointed to by *@executable_path* or in some cases for newer applications, *@loader_path*. The linker resolves these paths at runtime, and looks for dynamic libraries there. If the binary is moved away from its application bundle (where frameworks, libraries, and other resources are contained) the linker will not be able to find the necessary libraries and the binary will fail to load. We could not find a way to change *@executable_path* at runtime, so the "hack" we used was to use the binary directory for the temporary location and keep the filename the same, but append the PID (Process Identification) number to the temporary file.

C. *Infection*

Once the temporary binary is written, the application *fork()*'s, changes the permissions of the binary as necessary, and runs it as a child process using the *execve()* system call. As the host binary is running, the virus code proceeds to infect other targets. Originally, the code would infect all global applications (/Applications) and user applications (~/Applications) about 95% of the time. The other 5% of the time it is run, it will scan the whole file system for binaries to infect, including /bin, /sbin, /usr/bin, /usr/sbin, etc, and any network drives or other mounted file systems. For the sake of easier testing and demonstration, the virus has been modified to only infect the user applications (~/Applications), leaving the rest intact. The demo version also attempts to infect a binary named "test" in the current directory.

Finding binaries to infect is not a trivial task. Many Mach-O binaries are non-executable shared or static libraries, and some system libraries are not writable to by default for any user. The reconnaissance phase must also be very robust, because if the virus code crashes, the host application will go down with it. Many system calls are used to find host binaries, so stealth will always be a challenge.

For each specified directory, the virus uses the *fts_read()* system call to get a linked list tree of file paths in *FTSENT* nodes. These nodes are then filtered for everything but regular files. If the files have execute permissions by either the owner, group, or world, the file is sent to the infection stage.

This stage tests if the file is large enough to be a universal binary, and can be read in the first place. Only files that have read and execute permissions will be infected by the virus. It will try to set write permissions itself if it can, then restore the permissions afterwards. The virus does not attempt to cover its tracks by setting the modified date back to the original.

To determine if the virus is a universal binary, it tries to read the *fat_header* of the file and check the *magic* number (Section II.B). If the virus is running on a little endian machine, it will also need to byte swap the *fat_header* to read it. If the magic number matches, and the number of architectures is less than 20 (a hack for possible Java binary conflicts), we agree that it is a universal binary.

We then use a little hack by seeking to byte 0x1000 where the *mach_header* of the first architecture is usually stored (Section II.A). This value was based on experimentation, but sources suggest that it is static unless the fat archive contains many Mach-O binaries that wouldn't fit into the 0x1000 boundary [7]. At this offset, the *mach_header* is read and checked for its magic value, *MH_MAGIC* or *MH_CIGAM* for little endian machines. If these tests are successful, we make sure the binary is an executable by checking the *mach_header.filetype* field for *MH_EXECUTE or MH_EXECUTE << 24* on little endian machines.

These checks are continued (e.g. make sure binary is not already infected, etc) until the binary can be copied to a temporary location in /tmp, overwritten with the contents of the virus, and the original appended to the virus from the temporary location. All this is done by using the file descriptor of the running infected binary to read from itself. Permissions can be a problem, which the virus tries to get around by setting permissions manually, but if this fails, the virus gives up infecting the target and moves onto the next one. Using this method the virus infects all universal binary applications, and once it has finished, it executes its payload.

D. Payload

The payload is up to the virus implementer. As discussed in Section A, the virus prints messages and collects user data and emails it. Most of this is done with the *system()* call. This invokes the command shell (*/bin/sh*) with the specified commands to run, effectively enabling shell scripts to be executed. This easily allowed us to make a dialog box with AppleScript (osascript -e 'tell application "Finder" to display dialog "This application has been infected."'), and collect user information with a combination of *tar* and *mail* commands. The creativity of a virus writer, however, never ends, and more interesting payloads could be invented to work with Felcator. For example, Felcator could email itself to everyone on the infected user's address book.

E. Gaining Root Access with Roota

As mentioned earlier, the virus gives up infecting files that it does not have permissions to write to. This might include system directories like /bin and /sbin since by default these are owned by root. Note that all applications in /Applications are writable by any admin user on the system, which by default is the first user of Mac OS X. Most people run their operating systems as administrators simply due to the fact it is the default. The safe defaults principle could very well apply here. Any additional applications installed in /Applications and ~/Applications are writable by the person that installed them, making write permissions even more open.

However, if you are a non-admin user on a Mac OS X system, it would be much easier for the virus to spread if it could gain root access. Enter Roota, a local root exploit for Mac OS X using a vulnerability in pppd that is installed by default in Mac OS X. Details on how the exploit works will not be covered in this paper to length constraints. However, the basis of the exploit is that it creates a malicious pppd plugin that starts a root shell, and loads the plugin in such a way that gets around pppd's check to make sure the stdin descriptor is owned by root. The pppd binary is setuid, and thus executes with root privileges.

Apple has since patched this flaw with a security update, but it serves as a demonstration on how an up-to-date system a few months ago could be vulnerable to such an attack. Using root privileges Felcator could infect all binaries on the system.

F. Demonstration

Here is a demonstration with some output snipped:

```
$ make
gcc parasite.c -g3 -arch ppc -arch i386 -o psite
gcc host.c -g3 -arch ppc -arch i386 -o host
$ file host
host: Mach-O universal binary with 2 architectures
host (for arch ppc):    Mach-O executable ppc
host (for arch i386):   Mach-O executable i386
$ ./host
Hello, I am the host binary.
$ cp /bin/ls test
/bin/ls -> test
$ file test
test: Mach-O universal binary with 2 architectures
test (for arch i386):   Mach-O executable i386
test (for arch ppc):    Mach-O executable ppc
$ ./infector.pl host
[+] Infecting file: ( host ).
[+] Using file size: ( 0x120d8 ).
[+] File Successfully infected.
$ ./host
Infecting 'test'
Hello, I am the host binary.
Infected 'test'
file infected.
```

```
$ ./test
File 'test' already infected, skipping
file infected.
Makefile        a.out           host.c
parasite.c      psite
host            infector.pl     payload.c       test
$
```

G. Purpose

Felcator demonstrates that Mac OS X is vulnerable, as any other modern operating system with privilege separation, is to virus attacks. Not only are these virus attacks proof-of-concept, but by showing that malicious acts can be malicious we realize that they are a real threat. Gaining all the user's saved passwords and internet history simply by running a malicious program unknowingly can cause a lot of damage.

Many users have been trained to click accept when Safari prompts them if they are downloading an application. The response will only be natural when a malicious program asks to be downloaded. Even though user interaction is required to spread the Felcator virus, it is not out of the ordinary to rely on user interaction as it is how a lot of malware spreads on other platforms, particularly Microsoft Windows XP.

V. DEFENSE STRATEGIES

The preceding sections showed how universal binaries in Mac OS X can be compromised and exploited to malicious ends. We will now investigate how to defend against such an attack. There are two basic defense strategies we will consider: prevention, and detection. We will deal with these two categories separately. Please note that the recommendations in this section are based on Mac OS X version 10.4 (Tiger). Version 10.5 was released during our work on this project.

A. Prevention Using Stricter Access Control

Prevention means not letting the exploit occur in the first place and this can be accomplished by utilizing the "least privilege" principle of designing secure systems. Mac OS X is based on the UNIX operating system and has the same access control model. This model is a discretionary access control model which uses access control lists where every file on the system has read, write, and execute privileges assigned to it for various users and groups.

Our exploit has the option of using the pppd vulnerability to gain root access but if the target is not vulnerable, it tries to infect binaries using whatever level of access the account in which it is running has. If the account that it is running in has write access to many binary files then the malware can infect many applications.

In Mac OS X there are three types of accounts: standard, administrator, and root. A standard account can only affect the files in his/her home folder. An administrator account has more access than a standard account and can not only affect the home folder, but also the Applications folder where applications are installed and the Library folder which contains application and OS related support files. The root account can only be accessed from an admin account and has ultimate powers in the system. This includes the ability to affect the System folder which contains the essential components of OS X. The root account is disabled by default and can be activated by an administrator.

A possibly dangerous property in Mac OS X is the fact that when the system is first initialized, the first account is created as a member of the admin group which has write access to many binaries files on the system. If a user in the admin group executed our malware by accident it could infect a large number of files because of the default permissions. A way to solve this problem is to adjust the OS to require the user to create two accounts upon loading the OS for the first time, one administrator account and one standard account. All application installation and system modification could be done in the admin account and everything else could be done in the normal account which would not have permission to write to any binary files. Using this method would be utilizing the "least privilege" design principle. Another important principle to consider when making this change is "psychological acceptability" because the reason why two accounts are required should be clearly explained to the user.

B. Sandboxing

Another prevention method is a technique called sandboxing. Sandboxing provides a tightly-controlled set of resources for programs to run in, such as scratch space on disk and memory. By setting a default sandbox for unknown programs, we could make sure that unknown programs could write only to a specified area of the disk and could not write outside of that. This way, there is no chance that our exploit could write to a binary file even if the current user has write access to it. Another way sandboxing could be useful is to sandbox known programs so if they do get compromised by our exploit, the resources the exploit can access will be limited to the host program's sandbox. Sandboxing is based on the "least privilege" principle of designing secure systems. It also utilizes the principle of "defense in depth" because it is an extra layer of access control on top of the user privilege access control.

VI. DETECTION METHODS

If the prevention methods fail and a binary file does get infected with our exploit, then it should be detected and removed. The tricky part of this is detection because once it is detected then it should be fairly easy to remove. Our exploit could be detected using signature detection, change detection, activity scanning or by making mach-o files more secure by redesigning the header. All virus detection methods attempt to ensure the integrity of software because they are designed to detect code that should not be there. They also indirectly attempt to ensure confidentiality and availability by detecting malicious software that might harm the confidentiality or availability of data. For example, if a virus is designed to perform a denial of service attack then a virus scanner would be ensuring data availability by detecting and removing the virus. Similarly, if a virus was designed to steal credit card numbers and email them to some email address then a virus scanner that detected it would be ensuring the confidentiality of the data.

A. *Signature Detection*

One possible way to detect if a binary has been infected with our exploit is called signature detection. In this case, the signature will be a string of bits found in the malicious portion of binary that was changed. All of the binaries on the system could then be scanned looking for the signature. The signature string would have to be of sufficient size as to minimize false positive matches. Two advantages of this method are that it is simple to implement and it provides specific identification of viruses. A disadvantage is that information about the exploit must be known in order to create a signature to scan for. Another disadvantage is that if a signature was created for the exploit, the exploit's creators could make subtle changes in the exploit's code such as redundant or useless code that wouldn't change its functionality but would change its signature. In this way, the malware creator could compile many versions of the exploit all with different signatures and a signature scanner would have to have a different signature for each different version. For example, Trudy could have the source code for a working virus but a signature has already been created to detect the virus. She could then go back to the source code, create a variable called 'x', and then every fourth line insert the line of code "x = 1". If the variable 'x' was not used in the previous version of the virus then this change would not affect the functionality of the virus but it would affect the compiled machine code for the virus and would very likely change its signature. She could then go back and change the additional lines to "x = 2" and compile a third unique version. This way she could create any number of unique viruses that all have the same functionality but different signatures.

B. *Change Detection*

Another way to detect our exploit is to use change detection. Change detection can be accomplished by computing a hash of every binary file on the system. Then, at regular intervals, the hashes are recomputed and compared to their original values. The main advantage of this method is that there are virtually no false negatives so if a file has been infected it will definitely be detected. This method also has several disadvantages. First, the original hash file must be computed before the infection occurs. If the binary is infected before a hash of the uninfected file is computed, then change detection will reveal nothing. Another disadvantage comes from the fact that binary files on a system do change for legitimate reasons like when they are updated. For this reason, a change detection scan would likely come up with numerous false positives which would place a heavy burden on the user to sort through them. It is also conceivable that a binary file could be infected shortly after it was updated for legitimate reasons and the infection would not be detected.

C. *Activity and Heuristic Scanner*

An activity based virus scanner monitors the activity of programs and reports any suspicious behavior. It may, for example, detect any calls to format the disk that were not made by the OS. It may also detect a program tying to modify or change binary files because this type of behavior is unusual. Programs other than installers usually only modify data files and not executables. An activity based scanner would be effective against our exploit because it is an application that tries to modify binary files and once it infects a binary file, it uses the host program to infect other binary files. For example, if our exploit infected a word processor, then it would use it to infect other binaries. An activity based scanner would monitor the activity of the word processor and alert the user if the word processor tried to write to an executable file. A similar type of scanner to an activity based scanner is a heuristic scanner. Instead of monitoring activity it looks for suspicious sections of code that are generally found in viral programs. For example, a heuristic scanner might scan for code in a program that looks for other program files. A big advantage of activity based and heuristic scanners is that they can scan for unknown viruses. The scanner does not need to know the exact nature of a virus because it will detect it as long as it does some kind of general suspicious thing that the scanner is programmed to catch. A big disadvantage is that it is difficult to define bad behavior that a virus does but a legitimate program never does. Nevertheless, activity and heuristic scanners still remain useful tools.

D. *Improve the Mach-O Format*

A third way to detect our exploit would be to modify the design of the Mach-O file header to include the program length. This would be specified as an offset value. Since our exploit concatenates itself to the end of existing binaries, it modifies the length of the binary. If the length had been specified in the header, then the OS could generate a warning if the system executed code at an address that the OS thought was the exit point of the program but the program did not exit. Of course the malware could simply write over information in the header file and change the exit point so the header would have to be signed and checked whenever the program is run. This detection method would be effective but difficult to implement since it involves changing the format of Mach-O files.

VII. CONCLUSION

Apple's Mac OS X operating system is just as vulnerable to virus attacks as other operating systems. The fact that it has not been a major target for attackers could change as Mac OS X is becoming more popular. And the fact that there are not as many viruses for Mac OS X as there are for other platforms does not reduce the severity of the threats that could be realized. It is therefore advised for Mac users to watch what they're downloading and running. Having antivirus software with change detection and heuristic activity scanning would help reduce the risk, as well as running sandboxed and signed applications. Hopefully we will see improvements to Mach-O in the future that will make it harder to infect applications.

References

[1] Apple (2007, October) Mac OS X Security [Online]. Available: http://www.apple.com.sg/macosx/features/security

[2] J. Jones. (2007, June 18) Days of Risk in 2006: Client OS Products [Online]. Available: http://blogs.csoonline.com/node/365

[3] J. Jones. (2007, June 13) Days of risk in 2006: Linux, Mac OS X, Solaris and Windows [Online]. Available: http://blogs.csoonline.com/days_of_risk_in_2006

[4] M. Muthanna. (2006, March 26) How OS X Executes Applications [Online]. Available: http://0xfe.blogspot.com/2006/03/how-os-x-executes-applications.html

[5] R. G Biv. (2006, October) Infecting Mach-O Files [Online]. Available: http://vx.netlux.org/lib/vrg01.html

[6] N. Archibald. (2004, November 18) Infecting the Mach-O Object Format [Online]. Available: http://felinemenace.org/~nemo/slides/mach-o_infection.ppt

[7] Jon. (2006, June 9) Mach-O and Universal Binaries [Online]. Available: http://hohle.net/scrap_post.php?post=197

[8] iDefense Labs. (2007, May 24) Public Advisory [Online]. Available: http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=537

[9] Wikipedia (2007, October 25) Universal Binary [Online]. Available: http://en.wikipedia.org/wiki/Universal_binary,

[10] Wikipedia (2007, October 24) Fat Binary [Online]. Available: http://en.wikipedia.org/wiki/Fat_binary#Mach-O_and_Mac_OS_X

[11] J. Hohle (2006, June 9) Mach-o and Universal Binaries [Online]. Available: http://hohle.net/scrap_post.php?post=197

[12] N. Dhanjani. (2004, June 27) New (local) Mac OS X vulnerability: Passwords in Swap files [Online]. Available: http://www.oreillynet.com/onlamp/blog/2004/06/new_local_mac_os_x_vulnerabili.html

[13] Wikipedia (2007, October 14) Mach-O [Online]. Available: http://en.wikipedia.org/wiki/Mach-O

[14] A. Adams of Symantec. (2006, November 13) DeepSight Threat Management System: Research Report [Online]. Available: http://downloads.securityfocus.com/downloads/MacOSX_DeepSight_Report.pdf