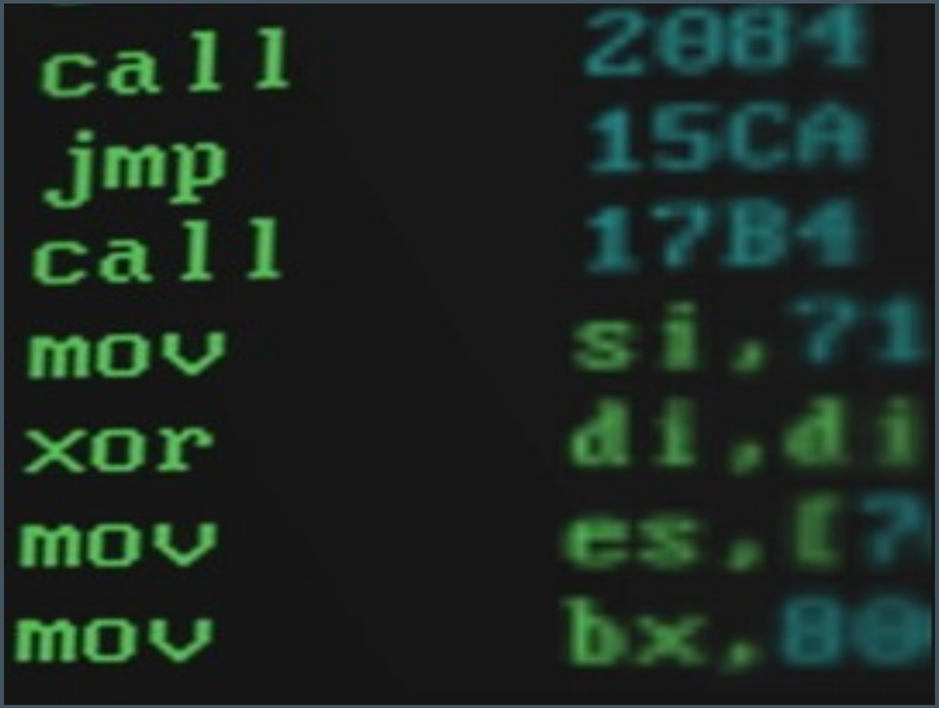
OXFORD BROOKES UNIVERSITY

COMP6016 Static Analysis II



Dr MUHAMMAD HILMI KAMARUDIN

Understanding C Code construct in Assembly

- Malware analyst need to be able to obtain a high-level picture of the code functionality.
- Skill that need time to develop.
- Typically malware is developed using high-level language, commonly C.
- A *code construct* is a code abstraction level that defines a functional property (not detail implementation).
- Eg: loops, conditional statement and so on.
- Discuss on popular C code construct.
- Malware analyst need to be able to go from disassembly to high level constructs,
- For help with C language, have a look look at the classic *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1988)
- Remember that the goal is to understand the overall functionality of a program, not to analyze every single instruction.

Global vs Local Variables

- *Global variables* can be accessed and used by any function in a program.

- *Local variables* can be accessed only by the function in which they are defined. Both global and local variables are declared similarly in C, but they look completely different in assembly.

```
int x = 1;
int y = 2;

void main()
{
   x = x+y;
   printf("Total = %d\n", x);
}
```

Figure: A simple program with two global variables

Global vs Local Variables cont

- *Global variables* can be accessed and used by any function in a program.

- *Local variables* can be accessed only by the function in which they are defined. Both global and local variables are declared similarly in C, but they look completely different in assembly.

```
int x = 1;
int y = 2;

void main()
{
   x = x+y;
   printf("Total = %d\n", x);
}
```

Figure 1: A simple program with two global variables

```
void main()
{
   int x = 1;

   int y = 2;

   x = x+y;
   printf("Total = %d\n", x);
}
```

Figure 2: A simple program with two global variables

Global vs Local Variables cont

```
00401003    mov     eax, dword_40CF60
00401008    add     eax, dword_40C000
0040100E    mov     dword_40CF60, eax ❶
00401013    mov     ecx, dword_40CF60
00401019    push    ecx
0040101A    push    offset aTotalD  ;"total = %d\n"
0040101F    call    printf
```

Fig 3: Assembly code for the global variable example in Fig 1

```
00401006    mov     dword ptr [ebp-4], 0
0040100D    mov     dword ptr [ebp-8], 1
00401014    mov     eax, [ebp-4]
00401017    add     eax, [ebp-8]
0040101A    mov     [ebp-4], eax
0040101D    mov     ecx, [ebp-4]
00401020    push    ecx
00401021    push    offset aTotalD  ; "total = %d\n"
00401026    call    printf
```

Fig 4: Assembly code for the local variable example in Fig 2

- The global variables are referenced by memory addresses, and the local variables are referenced by the stack addresses.
- In Listing Fig 3, the global variable x is signified by dword_40CF60, a memory location at 0x40CF60. Notice that x is changed in memory when eax is moved into dword_40CF60 at (1). All subsequent functions that utilize this variable will be impacted.
- In Fig 4, memory location [ebp-4] is used consistently throughout this function to reference the local variable x.

Disassembling Arithmetic Operations

```c
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

Fig 5: *C code with two variables and a variety of arithmetic*

Figure 5 shows the C code for two variables and a variety of arithmetic operations. Two of these are the -- and ++ operations, which are used to dec- rement by 1 and increment by 1, respectively. The % operation performs the *modulo* between the two variables, which is the remainder after performing a division operation.

Disassembling Arithmetic Operations

```
00401006    mov    [ebp+var_4], 0
0040100D    mov    [ebp+var_8], 1
00401014    mov    eax, [ebp+var_4] ❶
00401017    add    eax, 0Bh
0040101A    mov    [ebp+var_4], eax
0040101D    mov    ecx, [ebp+var_4]
00401020    sub    ecx, [ebp+var_8] ❷
00401023    mov    [ebp+var_4], ecx
00401026    mov    edx, [ebp+var_4]
00401029    sub    edx, 1 ❸
0040102C    mov    [ebp+var_4], edx
0040102F    mov    eax, [ebp+var_8]
00401032    add    eax, 1 ❹
00401035    mov    [ebp+var_8], eax
00401038    mov    eax, [ebp+var_4]
0040103B    cdq
0040103C    mov    ecx, 3
00401041    idiv   ecx
00401043    mov    [ebp+var_8], edx ❺
```

CDQ stands for Convert Double to Quadra and will extend the sign bit in EAX filling EDX as the division is EDX:EAX / ECX in this case! Where : indicates concatenation.

Fig 6: *Assembly code for the arithmetic example in Fig 5*

- In this example, a and b are local variables because they are referenced by the stack. IDA Pro has labeled a as var_4 and b as var_8.
- First, var_4 and var_8 are initialized to 0 and 1, respectively. a is moved into eax (1), and then 0x0b is added to eax, thereby incrementing a by 11. b is then subtracted from a (2). (The compiler decided to use the sub and add instructions (3) and (4), instead of the inc and dec functions.)
- The final five assembly instructions implement the modulo. When performing the div or idiv instruction (5), we are dividing edx:eax by the operand and storing the result in eax and the remainder in edx. That is why edx is moved into var_8 (5).

Recognizing if Statements

- If statement use by programmer to alter program execution based on certain conditions. if statements are common in C code and disassembly.
- Figure 7 displays a simple if statement in C with the assembly for this code shown in Figure 8. Notice the conditional jump jnz at (2). There must be a conditional jump for an if statement, but not all conditional jumps correspond to if statements.

```c
int x = 1;
int y = 2;

if(x == y){
     printf("x equals y.\n");
}else{
     printf("x is not equal to y.\n");
}
```

Fig 7: C code if statement example

```
00401006         mov     [ebp+var_8], 1
0040100D         mov     [ebp+var_4], 2
00401014         mov     eax, [ebp+var_8]
00401017         cmp     eax, [ebp+var_4] ❶
0040101A         jnz     short loc_40102B ❷
0040101C         push    offset aXEqualsY_ ; "x equals y.\n"
00401021         call    printf
00401026         add     esp, 4
00401029         jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B         push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030         call    printf
```

Fig 8: Assembly code for the if statement example in Fig 7

Graphic function with IDA PRO

- IDA Pro has a graphing tool that is useful in recognizing constructs, as shown in Figure 9. This feature is the default view for analyzing functions.

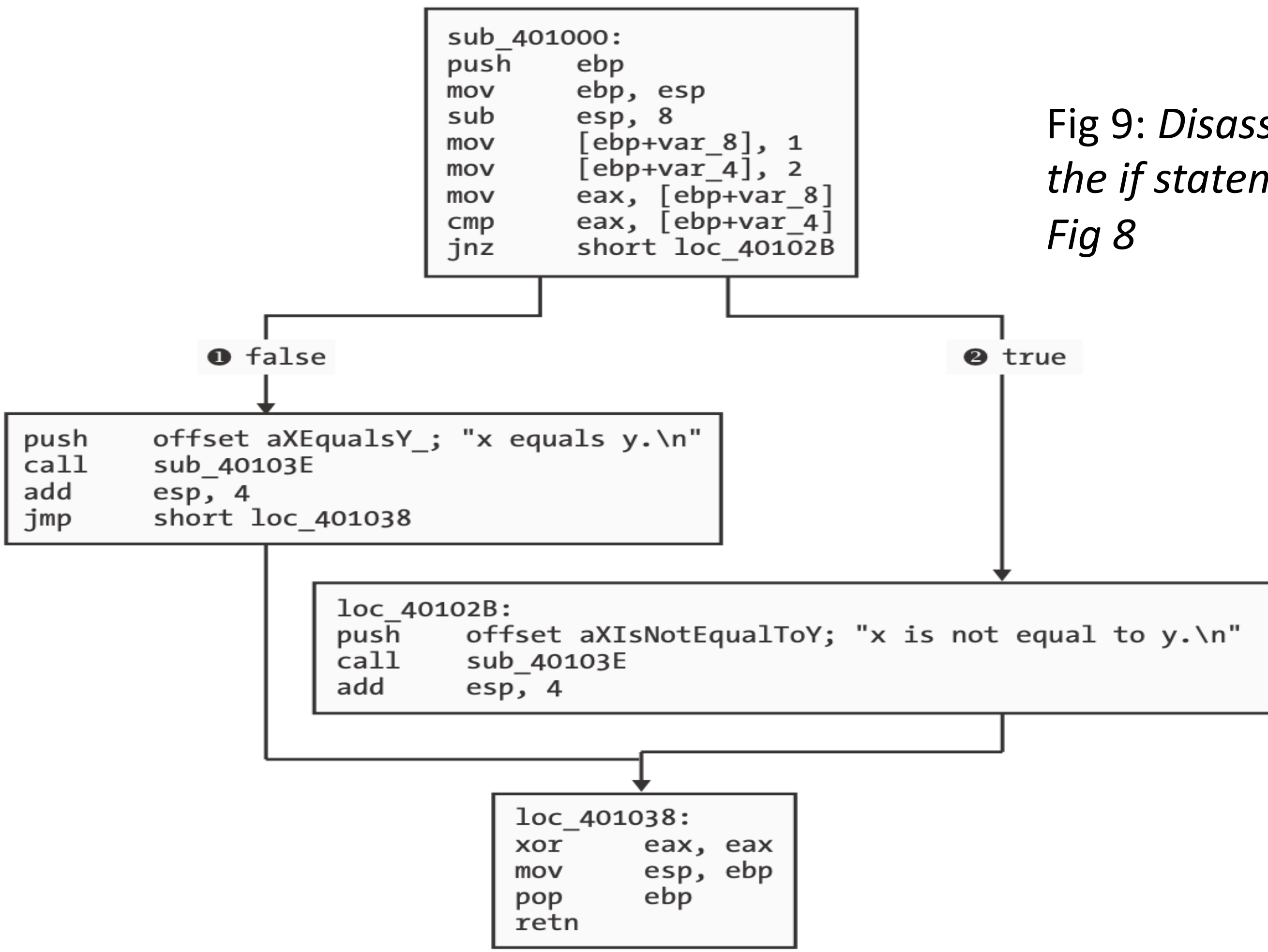


Fig 9: *Disassembly graph for the if statement example in Fig 8*

Recognizing for Loops

The for loop is a basic looping mechanism used in C programming. for loops always have four components: initialization, comparison, execution instructions, and the increment or decrement.

```
int i;

for(i=0; i<100; i++)
{
   printf("i equals %d\n", i);
}
```

Fig 10: *C code for a for loop*

- The initialization sets i to 0 (zero), and the comparison checks to see if i is less than 100.
- If i is less than 100, the printf instruction will execute, the increment will add 1 to i, and the process will check to see if i is less than 100.
- These steps will repeat until i is greater than or equal to 100

```
00401004         mov     [ebp+var_4], 0 ❶
0040100B         jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D         mov     eax, [ebp+var_4] ❸
00401010         add     eax, 1
00401013         mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016         cmp     [ebp+var_4], 64h ❺
0040101A         jge     short loc_40102F ❻
0040101C         mov     ecx, [ebp+var_4]
0040101F         push    ecx
00401020         push    offset aID  ; "i equals %d\n"
00401025         call    printf
0040102A         add     esp, 8
0040102D         jmp     short loc_40100D ❼
```

Fig 11: *Assembly code for the for loop example in Fig 10*

- In assembly, the for loop can be recognized by locating the four components—initialization, comparison, execution instructions, and increment/ decrement.
- For example, in Figure above, (1) corresponds to the initialization step. The code between (3) and (4) corresponds to the increment that is initially jumped over at (2) with a jump instruction. The comparison occurs at (5), and at (6) , the decision is made by the conditional jump.
- If the jump is not taken, the printf instruction will execute, and an unconditional jump occurs at (7) , which causes the increment to occur.

Loops Graph Mode

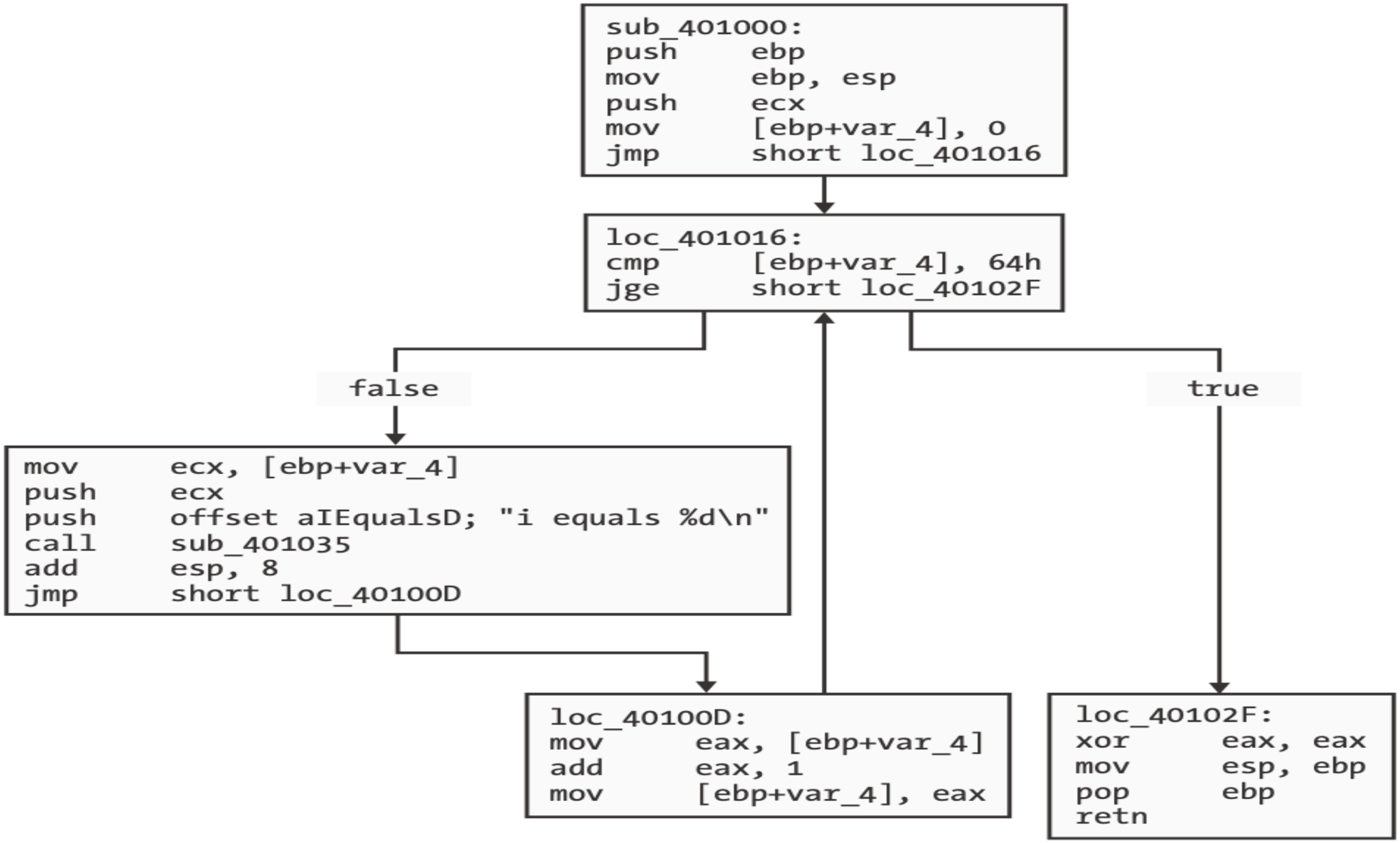


Figure 12: *Disassembly graph for the for loop example in Fig 11*

Recognizing While Loops

The while loop is frequently used by malware authors to loop until a condition is met, such as receiving a packet or command. while loops look similar to for loops in assembly, but they are easier to understand.

```
int status=0;
int result = 0;

while(status == 0){
     result = performAction();
     status = checkResult(result);
}
```

Fig 13: *C code for a while loop*

```
00401036         mov     [ebp+var_4], 0
0040103D         mov     [ebp+var_8], 0
00401044 loc_401044:
00401044         cmp     [ebp+var_4], 0
00401048         jnz     short loc_401063 ❶
0040104A         call    performAction
0040104F         mov     [ebp+var_8], eax
00401052         mov     eax, [ebp+var_8]
00401055         push    eax
00401056         call    checkResult
0040105B         add     esp, 4
0040105E         mov     [ebp+var_4], eax
00401061         jmp     short loc_401044 ❷
```

Fig 14: *Assembly code for the while loop example in Fig 13*

- The while loop in Fig 13 will continue to loop until the status returned from checkResult is 0.
- A conditional jump occurs at (1) and an unconditional jump at (2), but the only way for this code to stop executing repeatedly is for that conditional jump to occur.
- Revision: Read PMA page 119-132

Push and Move

- Compiler may choose to use different instructions for same operation
- Figure 15 shows a C code example of a function call.
- Function adder adds two arguments and return the results.

```
int adder(int a, int b)
{
   return a+b;
}

void main()
{
   int x = 1;
   int y = 2;

   printf("the function returned the number %d\n", adder(x,y));
}
```

Fig 15: *C code for function call*

```
00401730        push    ebp
00401731        mov     ebp, esp
00401733        mov     eax, [ebp+arg_0]
00401736        add     eax, [ebp+arg_4]
00401739        pop     ebp
0040173A        retn
```

Fig 16: *Assembly code for the adder function in fig 15*

Assembly code with 2 different calling conventions

OXFORD BROOKES UNIVERSITY

- Figure 17 shows different calling conventions used by two different compilers, (Microsoft Visual Studio and GNU Compiler Collection (GCC)).
- Prepared for both conventions- analyst wont have control over the compilers

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

Fig 17: *Assembly Code for a Function Call with Two Different Calling Conventions*

Disassembling Arrays

- Arrays are used to define an ordered set of similar data items.
- Malware sometimes uses array of pointer to strings contains multiple hostnames
- Figure 18 shows two arrays used by one program- set during iteration through *for* loop.
- Array a – locally defined
- Array b- globally defined

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

Fig 18: *C code for an array*

- Base address of array a corresponds to var_14
- Base address of array b corresponds to dword_A40A000
- Both array are integers-each element is of size 4, which is multiplied by 4 to account for the size of elements

```
00401006         mov     [ebp+var_18], 0
0040100D         jmp     short loc_401018
0040100F loc_40100F:
0040100F         mov     eax, [ebp+var_18]
00401012         add     eax, 1
00401015         mov     [ebp+var_18], eax
00401018 loc_401018:
00401018         cmp     [ebp+var_18], 5
0040101C         jge     short loc_401037
0040101E         mov     ecx, [ebp+var_18]
00401021         mov     edx, [ebp+var_18]
00401024         mov     [ebp+ecx*4+var_14], edx ❶
00401028         mov     eax, [ebp+var_18]
0040102B         mov     ecx, [ebp+var_18]
0040102E         mov     dword_40A000[ecx*4], eax ❷
00401035         jmp     short loc_40100F
```

Fig 19: *Assembly code for the array in Figure 18*

Analyzing Malicious Windows Programs

- The Windows API (Application Programming Interface)
- What is API?
  - Govern how programs interact with Microsoft libraries
- Concepts
  - Types and Hungarian Notation
  - Handles
  - File System Functions
  - Special Files

Types and Hungarian Notation

- Windows API has its own names to represent C data types
  - Such as DWORD for 32-bit unsigned integers and WORD for 16-bit unsigned integers
- Hungarian Notation
  - Variables that contain a 32-bit unsigned integer start with the prefix **dw**

Type and prefix	Description
WORD (w)	A 16-bit unsigned value.
DWORD (dw)	A double-WORD, 32-bit unsigned value.
Handles (H)	A reference to an object. The information stored in the handle is not documented, and the handle should be manipulated only by the Windows API. Examples include HModule, HInstance, and HKey.
Long Pointer (LP)	A pointer to another type. For example, LPByte is a pointer to a byte, and LPCSTR is a pointer to a character string. Strings are usually prefixed by LP because they are actually pointers. Occasionally, you will see Pointer (P)... prefixing another type instead of LP; in 32-bit systems, this is the same as LP. The difference was meaningful in 16-bit systems.
Callback	Represents a function that will be called by the Windows API. For example, the InternetSetStatusCallback function passes a pointer to a function that is called whenever the system has an update of the Internet status.

Fig 20: *Common API types*

Handles

•Items opened or created in the OS, like

•– Window, process, menu, file, ...

•Handles are like pointers to those objects

•– They not pointers, however

•The only thing you can do with a handle is store it and use it in a later function call to refer to the same object

•Handle Examples

- The CreateWindowEx function returns an HWND, a handle to the window
- To do anything to that window (such as DestroyWindow), use that handle

File System Functions

- CreateFile

  -This function is used to create and open files

- ReadFile and WriteFile

  -These functions are used for reading and writing to files. Both operate on files as a stream

- CreateFileMapping

  -The CreateFileMapping function loads a file from disk into memory.

- MapViewOfFile

  -The MapViewOfFile function returns a pointer to the base address of the mapping, which can be used to access the file in memory

Special FIles

- **Shared files** like \\server\share

– Or \\?\server\share

- Disables string parsing, allows longer filenames

- **Namespaces**

– Special folders in the Windows file system

\ Lowest namespace, contains everything

\\.\Device namespace used for direct disk input/output Witty worm wrote to

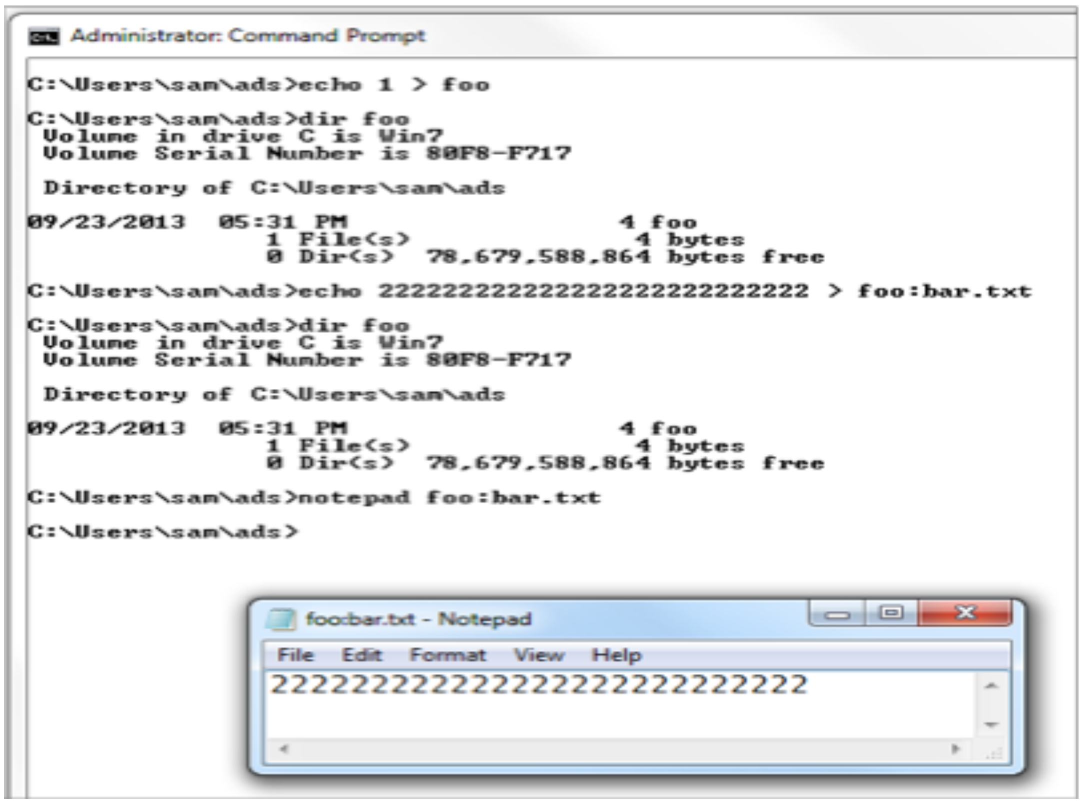
\\.\PhysicalDisk1 to corrupt the disk

OXFORD BROOKES UNIVERSITY

Alternate Data Stream

We link files together in NTFS and the second file won't show in directory listings. It does appear when reading the file's contents! It's a nice way to hide data

- Second stream of data attached to a filename
- File.txt:otherfile.txt



The Windows Registry

Registry Purpose

• Store operating system and program configuration settings

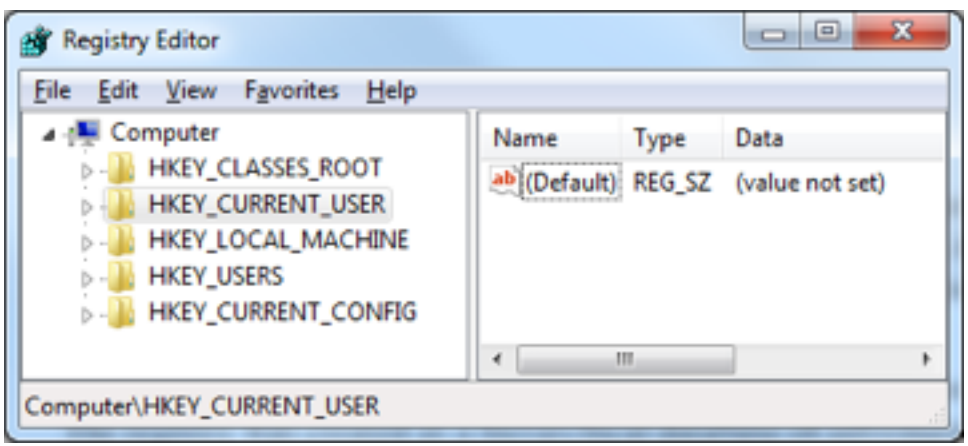
– Desktop background, mouse preferences, etc. • Malware uses the registry for **persistence**

– Making malware re-start when the system reboots

Registry Terms

- **Subkey**
- **Key**
- **Value entry**
- **Value** or **Data**
- REGEDIT

A folder within a folder
A folder; can contain folders or values Two parts: name and data
The data stored in a registry entry Tool to view/edit the Registry



Registry Root Keys

The registry is split into the following five root keys:

HKEY_LOCAL_MACHINE (HKLM) Stores settings that are global to the local machine

HKEY_CURRENT_USER (HKCU) Stores settings specific to the current user

HKEY_CLASSES_ROOT Stores information defining types

HKEY_CURRENT_CONFIG Stores settings about the current hardware configu- ration,

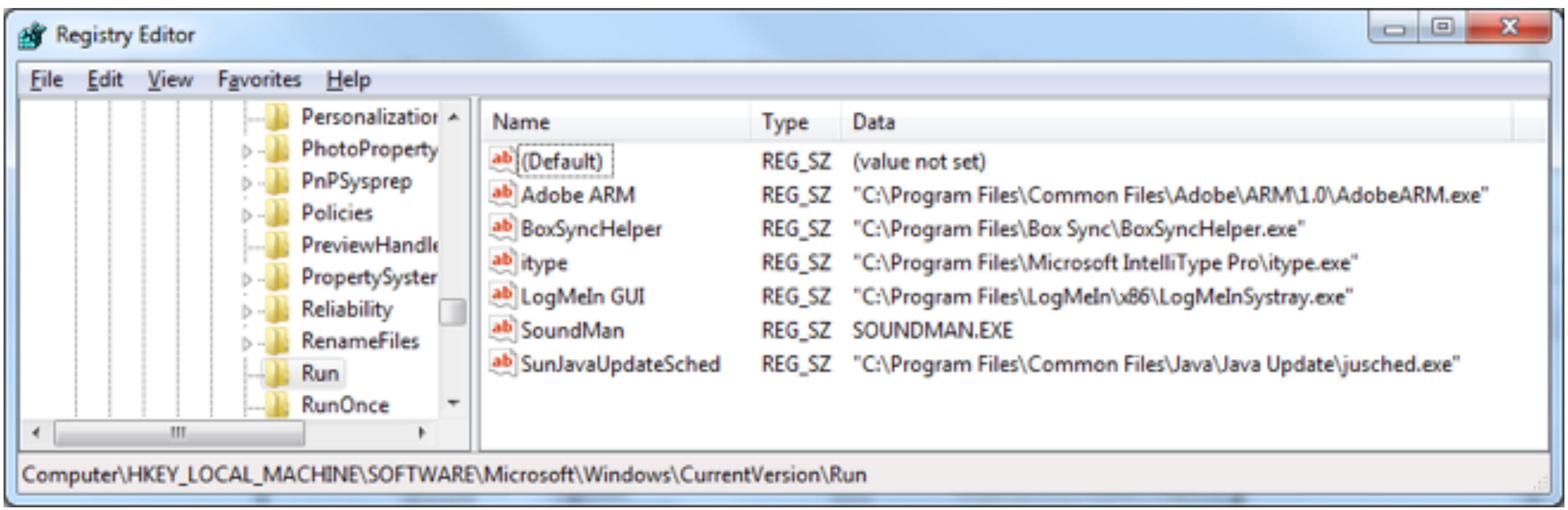
specifically differences between the current and the standard configuration

HKEY_USERS Defines settings for the default user, new users, and current users

Run key

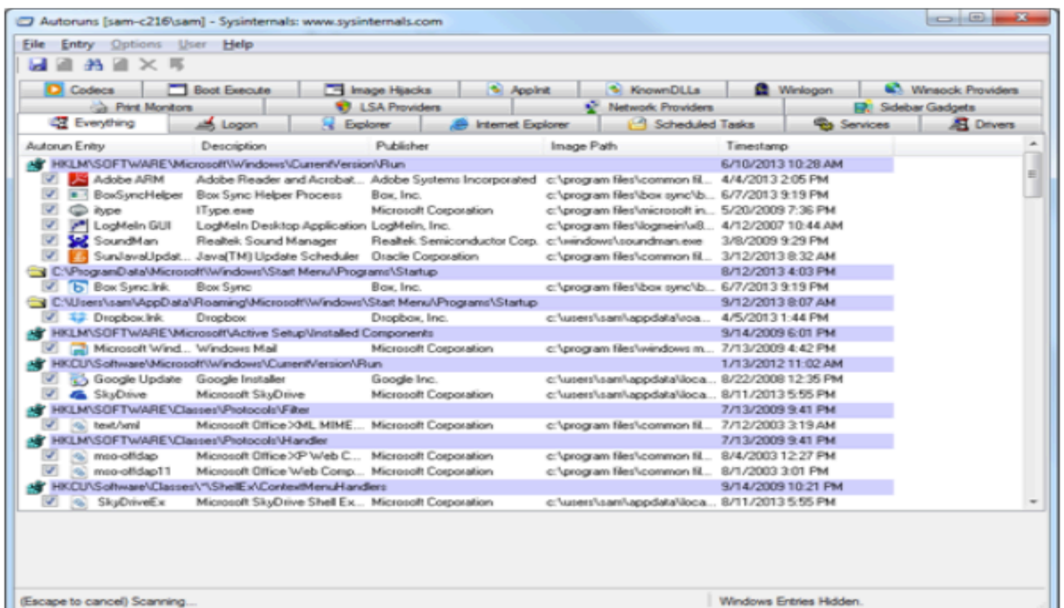
- **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion \Run**

– Executables that start when a user logs on



Autoruns

- Sysinternals tool
- Lists code that will run automatically when system starts
  - Executables
  - DLLs loaded into IE and other programs
  - Drivers loaded into Kernel
  - It checks 25 to 30 registry locations
  - Won't necessarily find all automatically running code



Be aware of Ex, A and W Suffixes

When evaluating unfamiliar Windows functions- easily get confused
Ex suffix- CreateWindowsEx.
Microsoft update function-same name with old one with added Ex suffix

Functions take strings as paameters include A or W at the end of their names.
CreateDirectoryW- simply indicates the function accepts a string parameter
A- for ASCII strings
W- Wide character strings

Networking APIs

Berkeley Compatible Sockets

- Winsock libraries, primarily in *ws2_32.dll* – Almost identical in Windows and Unix

– Berkeley compatible sockets

Function	Description
socket	Creates a socket
bind	Attaches a socket to a particular port, prior to the accept call
listen	Indicates that a socket will be listening for incoming connections
accept	Opens a connection to a remote socket and accepts the connection
connect	Opens a connection to a remote socket; the remote socket must be waiting for the connection
recv	Receives data from the remote socket
send	Sends data to the remote socket

NOTE

The WSAStartup function must be called before any other networking functions in order to allocate resources for the networking libraries. When looking for the start of network connections while debugging code, it is useful to set a breakpoint on WSAStartup, because the start of networking should follow shortly.

Fig 21: *Berkeley Compatible Sockets Networking Functions*

Server and Client Sides

- Server side

– Maintains an open socket waiting for connections

– Calls, in order, **socket**, **bind**, **listen**, **accept** – Then **send** and **recv** as necessary

- Client side

– Connects to a waiting socket

– Calls, in order, **socket**, **connect** – Then **send** and **recv** as necessary

Server socket

```
00401041  push    ecx             ; lpWSAData
00401042  push    202h            ; wVersionRequested
00401047  mov     word ptr [esp+250h+name.sa_data], ax
0040104C  call    ds:WSAStartup
00401052  push    0               ; protocol
00401054  push    1               ; type
00401056  push    2               ; af
00401058  call    ds:socket
0040105E  push    10h             ; namelen
00401060  lea     edx, [esp+24Ch+name]
00401064  mov     ebx, eax
00401066  push    edx             ; name
00401067  push    ebx             ; s
00401068  call    ds:bind
0040106E  mov     esi, ds:listen
00401074  push    5               ; backlog
00401076  push    ebx             ; s
00401077  call    esi ; listen
00401079  lea     eax, [esp+248h+addrlen]
0040107D  push    eax             ; addrlen
0040107E  lea     ecx, [esp+24Ch+hostshort]
00401082  push    ecx             ; addr
00401083  push    ebx             ; s
00401084  call    ds:accept
```

Fig 22: *A simplified program with a server socket*

The WinINet API

- Higher-level API than Winsock
- Functions in *Wininet.dll*
- Implements Application-layer protocols like HTTP and FTP
- **InternetOpen** – connects to Internet
- **InternetOpenURL** –connects to a URL
- **InternetReadFile** –reads data from a downloaded file

Following Running Malware

- **jmp** and **call** transfer execution to another part of code,

but there are other ways

– DLLs

– Processes

– Threads

– Mutexes

– Services

– Component Object Model (COM) – Exceptions

DLLs (Dynamic Link Libraries)

Share code among multiple applications

- DLLs export code that can be used by other applications
- Static libraries were used before DLLs

– They still exist, but are much less common

– They cannot share memory among running processes

– Static libraries use more RAM than DLLs

DLL Advantages

- Using DLLs already included in Windows makes code smaller
- Software companies can also make custom DLLs

– Distribute DLLs along with EXEs

How Malware Authors Use DLLs

- Store malicious code in DLL

– Sometimes load malicious DLL into another process

- Using Windows DLLs

– Nearly all malware uses basic Windows DLLS

- Using third-party DLLs

– Use Firefox DLL to connect to a server, instead of Windows API

Basic DLL Structure

OXFORD BROOKES UNIVERSITY

- DLLs are very similar to EXEs
- PE file format
- A single flag indicates that it's a DLL instead of an EXE
- DLLs have more exports & fewer imports
- **DllMain** is the main function, not exported, but
- specified as the entry point in the PE Header
  – Called when a function loads or unloads the library

Processes

- **CreateProcess**

– Can create a simple remote shell with one function call

– **STARTUPINFO** parameter contains handles for standard input, standard output, and standard error streams

- Can be set to a socket, creating a remote shell

```
004010DA    mov     eax, dword ptr [esp+58h+SocketHandle]
004010DE    lea     edx, [esp+58h+StartupInfo]
004010E2    push    ecx                 ; lpProcessInformation
004010E3    push    edx                 ; lpStartupInfo
004010E4 ❶mov     [esp+60h+StartupInfo.hStdError], eax
004010E8 ❷mov     [esp+60h+StartupInfo.hStdOutput], eax
004010EC ❸mov     [esp+60h+StartupInfo.hStdInput], eax
004010F0 ❹mov     eax, dword_403098
004010F5    push    0                   ; lpCurrentDirectory
004010F7    push    0                   ; lpEnvironment
004010F9    push    0                   ; dwCreationFlags
004010FB    mov     dword ptr [esp+6Ch+CommandLine], eax
004010FF    push    1                   ; bInheritHandles
00401101    push    0                   ; lpThreadAttributes
00401103    lea     eax, [esp+74h+CommandLine]
00401107    push    0                   ; lpProcessAttributes
00401109 ❺push    eax                 ; lpCommandLine
0040110A    push    0                   ; lpApplicationName
0040110C    mov     [esp+80h+StartupInfo.dwFlags], 101h
00401114 ❻call    ds:CreateProcessA
```

Fig 23: *Sample code using the CreateProcess call*

Threads

- Processes are containers
  - Each process contains one or more threads
- Threads are what Windows actually executes
- Threads
  - Independent sequences of instructions
  - Executed by CPU without waiting for other threads
  - Threads within a process share the same memory space
  - Each thread has its own registers and stack

•When a thread is running, it has complete control of the CPU
•Other threads cannot affect the state of the CPU
•When a thread changes a register, it does not affect any other threads
•When the OS switches to another thread, it saves all CPU values in a structure called the **thread context**

How Malware Uses Threads

- Use **CreateThread** to load a malicious DLL into a process
- Create two threads, for input and output

– Used to communicate with a running application

Interprocess Coordination with Mutexes

- **Mutexes** are global objects that coordinate multiple processes and threads
- In the kernel, they are called **mutants**
- Mutexes often use hard-coded names which can be used to identify malware

Functions for Mutexes

- WaitForSingleObject

– Gives a thread access to the mutex

– Any subsequent threads attempting to gain access to it must wait

- ReleaseMutex

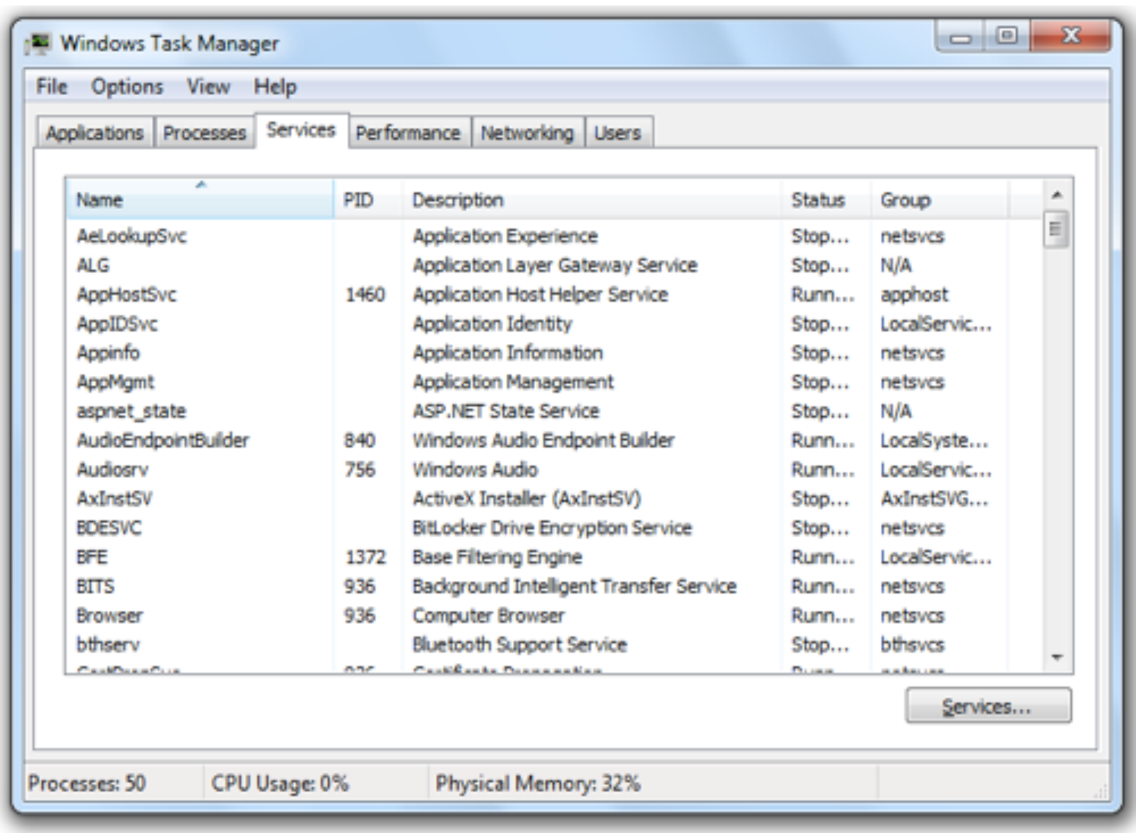
– Called when a thread is done using the mutex

- CreateMutex • OpenMutex

– Gets a handle to another process's mutex

Services

- Services run in the background without user input



Services

The task manager can¦t even show it! Sometimes…

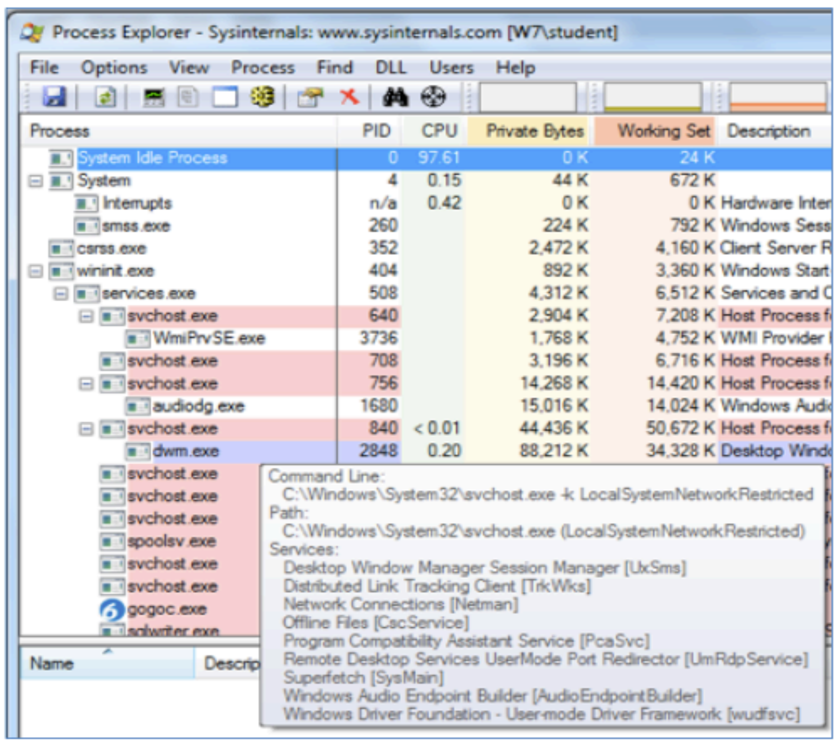
•Services often run as SYSTEM which is even more powerful than the Administrator
•Services can run automatically when Windows starts
  – An easy way for malware to maintain **persistence**
  – Persistent malware survives a restart

- OpenSCManager
  – Returns a handle to the Service Control Manager
- CreateService
  – Adds a new service to the Service Control Manager
  – Can specify whether the service will start automatically at boot time
- StartService
  – Only used if the service is set to start manually

Services

Svchost.exe

- WIN32_SHARE_PROCESS
  - Most common type of service used by malware
  - Stores code for service in a DLL
  - Combines several services into a single shared process named **svchost.exe**



OXFORD BROOKES UNIVERSITY

Services

SC Command

- Included in Windows
- Gives information about Services

```
C:\Windows\System32>sc qc Browser
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: Browser
        TYPE               : 20  WIN32_SHARE_PROCESS
        START_TYPE         : 3   DEMAND_START
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : C:\Windows\System32\svchost.exe -k netsvcs
        LOAD_ORDER_GROUP   : NetworkProvider
        TAG                : 0
        DISPLAY_NAME       : Computer Browser
        DEPENDENCIES       : LanmanWorkstation
                           : LanmanServer
        SERVICE_START_NAME : LocalSystem

C:\Windows\System32>
```

Component Object Model (COM)

- Allows different software components to share code
- Every thread that uses COM must call **OleInitialize** or **CoInitializeEx** before calling other COM libraries

REVISION: PMA page 155-161