Practical Malware Analysis

Lecture 7 | Analyzing Malicious Windows Programs

Goals

- Cover API basics
- Look for host-based indicators
- Discover unconventional methods for executing code
- Check out kernel mode

Goals

Cover API basics
Look for host-based indicators
Discover unconventional methods for executing code
Check out kernel mode

Windows API: Types and Hungarian Notation

Different names for C types

Variable prefixes indicate types

Table 7-1: Common Windows API Types

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.

The Windows API generally uses its own names for C variable types, so don't expect to see names like *int, short,* or *unsigned int*. Instead Windows uses Hungarian notation for function identifies, including a prefix naming scheme for variables that denote the variables' types. As seen in the above table, unsigned integers can be denoted by dw for a DWORD, or 32-bit unsigned value, and WORD (w) for a 16-bit unsigned value. Note that pointers (P) and long pointers (LP) are essentially the same for 32-bit architectures.

Windows API: Handles

- Like pointers
  - Reference objects or memory locations
    - Window, process, module, menu, file, etc
- Not like pointers
  - Not usable in arithmetic operations
  - Do not always represent object's address

Some Windows functions return handles to objects, used in later function calls to refer to the same object. Some handles may act as pointers, but generally they cannot be used in arithmetic operations and will not always represent an object's address in memory.

Windows API: File System Functions

- CreateFile
  - Open files, pipes, streams, I/O devices
  - Create files
- ReadFile, WriteFile
  - Operate on files as a stream
- CreateFileMapping, MapViewOfFile
  - Load file from disk -> memory, return a pointer to the base address of the mapping (respectively)
  - Used to replicate functionality of the OS loader in memory

CreateFile

Open files, pipes, streams, I/O devices

Create files (dwCreationDisposition controls open or create)

ReadFile, WriteFile

Operate on files as a stream

*"When you first call ReadFile, you read the next several bytes from a file; the next time you call it, you read the next several bytes after that"

CreateFileMapping, MapViewOfFile

Load file from disk -> memory, return a pointer to the base address of the mapping (respectively)

Easily jump to different memory addresses when parsing a file format

Used to replicate functionality of the OS loader in memory

"File mappings are commonly used to replicate the functionality of the Windows loader. After obtaining a map of the file, the malware can parse the PE header and make all necessary changes to the file in memory, thereby causing the PE file to be executed as if it had been loaded by the OS loader."

Windows API: Special Files

- Often used maliciously, not accessed by file path
  - Not visible in directory listings
  - May provide greater access to hardware and internal data
- Shared files
  - Access shared folders like *serverName**share* or \\?*serverName**share*
  - \\?\ disables string parsing, allows longer filenames

Often used maliciously, not accessed by file path

Not visible in directory listings

May provide greater access to hardware and internal data

Can be passed as strings to file-operation functions

Shared files

Access shared folders like \\serverName\share or \\?\serverName\share on a network

\\?\ tells the OS to disable string parsing, allows longer filenames

Windows API: Special Files

- Namespaces
  - Fixed number of folders, each stores different types of objects
    - NT namespace \ holds all devices, namespaces
      - WinObj Object Manager to view
    - Win32 device namespace \\.\
      - Often used maliciously to directly access devices, bypassing file system
- Alternate Data Streams (ADS)
  - *normalFile.txt:Stream:$DATA*
  - Add data to existing file
    - Not shown in directory listing or in contents of file
    - Only visible when accessed as a stream

Namespaces

Fixed number of folders, each stores different types of objects

NT namespace \ holds all devices, namespaces

WinObj Object Manager to view

Win32 device namespace \\.\

Often used maliciously to directly access devices, bypassing file system (and avoiding antivirus detection)

Prior to Windows server 2003 SP1, this could be used to write directly to memory from user-space, affecting kernel space

Alternate Data Streams (ADS)

normalFile.txt:Stream:$DATA

Add data to existing file

Not shown in directory listing or in contents of file

Only visible when accessed as a stream

The Windows Registry

Hierarchical database, stores OS and program configs

- Malware uses for persistence

Root key / hive / hkey	5 top-level ‘folders’ of the registry
Subkey	(subfolder)
Key	Folder containing folders or values
Value entry	Name, value ordered pair
Value / data	Information stored in a registry entry

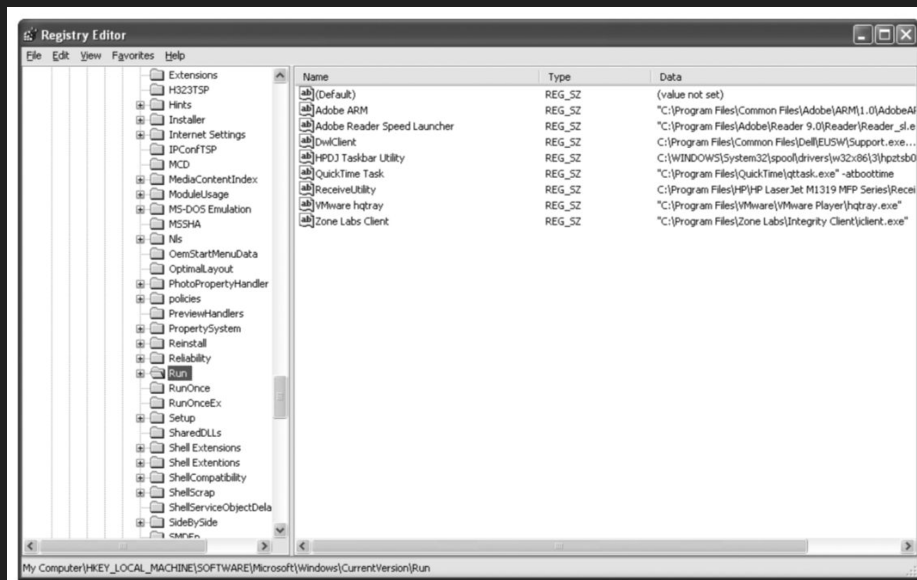
The Windows Registry is a hierarchical database that is used to store all of the operating system and program configuration details, including settings and options. It is a great place for malware to set up persistence, especially in the run keys which are used when the system boots. The registry is comprised of keys, which can be thought of as folders, with 5 top-level keys (also referred to as hives or hkeys) and many subkeys. Keys contain values, an ordered pair of a name and a value (data).

Registry Root Keys

HKEY_LOCAL_MACHINE (HKLM)	Global settings for the local machine
HKEY_CURRENT_USER (HKCU)	Current-user-specific settings
HKEY_CLASSES_ROOT	Information defining types
HKEY_CURRENT_CONFIG	Stores settings about the current hardware configuration, specifically differences between the current and the standard configuration
HKEY_USERS	Defines settings for the default user, new users, and current users

The above is a table of the five registry root keys, mostly for reference. The two most-referenced keys are HKLM and HKCU, with HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run being a very common key used by malware for persistence.

Regedit



Pictured: the built-in Windows registry editor (Regedit). Observe the listing of subkeys, including the currently selected Run key showing a listing of software (Name) that will be run on startup and the path (Data) to each program on disk. The Autoruns tool from Microsoft is useful for examining executables that run, DLLs loaded into Internet Explorer and other programs, and drivers loaded into the kernel when the OS starts.

Common Registry Functions

- RegOpenKeyEx
  - You can query and edit a key without opening it
    - Most programs don’t though
- RegSetValueEx
  - Add new value and set its data
- RegGetValue
  - Returns data for a value entry

RegOpenKeyEx
You can query and edit a key without opening it
Most programs don’t though
RegSetValueEx
Add new value and set its data
RegGetValue
Returns data for a value entry

```
0040286F   push    2               ; samDesired
00402871   push    eax             ; ulOptions
00402872   push    offset SubKey   ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877   push    HKEY_LOCAL_MACHINE ; hKey
0040287C ❶call    esi ; RegOpenKeyExW
0040287E   test    eax, eax
00402880   jnz     short loc_4028C5
00402882
00402882 loc_402882:
00402882   lea     ecx, [esp+424h+Data]
00402886   push    ecx             ; lpString
00402887   mov     bl, 1
00402889 ❷call    ds:lstrlenW
0040288F   lea     edx, [eax+eax+2]
00402893 ❸push    edx             ; cbData
00402894   mov     edx, [esp+428h+hKey]
00402898 ❹lea     eax, [esp+428h+Data]
0040289C   push    eax             ; lpData
0040289D   push    1               ; dwType
0040289F   push    0               ; Reserved
004028A1 ❺lea     ecx, [esp+434h+ValueName]
004028A8   push    ecx             ; lpValueName
004028A9   push    edx             ; hKey
004028AA   call    ds:RegSetValueExW
```

Prior to (1) we can see the parameters for RegOpenKeyEx being pushed onto the stack from right to left.

HKEY   hKey, a handle an open root key
LPCTSTR lpSubKey, the subkey to be opened
DWORD   ulOptions, an option to apply when opening the key, likely 0
REGSAM samDesired, the desired access rights to the key - the value of 2 specifies KEY_SET_VALUE allowing access to create, delete, or set a value.

At (2) a call to lstrlenW is used to get the size of the data about to be set in the RegSetValueExW call (pushed onto the stack at (3)). We can see a variety of other parameters pushed onto the stack, including the data at (4) and the name of the value that the data is stored under at (5).

Registry Scripting

.reg files - basically a script for modifying the registry

```
Windows Registry Editor Version 5.00 (Win XP)

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"
```

A .reg file, when run, automatically modifies the registry by merging the information contained in the file into the registry. Malware may use .reg files, but usually just directly edits the registry. Above is a simply example of the contents of a .reg file.

Networking APIs

Berkeley Compatible Sockets

- Winsock libraries -> ws2_32.dll
- WSAStartup

Table 7-2: Berkeley Compatible Sockets Networking Functions

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

Malware most commonly uses Berkeley Compatible Sockets of all the networking options available on Windows. You will most commonly see this functionality accessed via the ws2_32 dynamic link library, with the most common functions pictured in the table above.

When debugging code and looking for network functionality, look for the WSAStartup function being called before any other network functions in order to allocate resources.

Client-Server Architecture

Malware may operate as either client or server

- Client calls
  - socket -> connect -> send/recv
- Server calls
  - socket -> bind -> listen -> accept -> send/recv

Malware may operate as either client or server

Client function calls

socket -> connect -> send/recv

Server function calls

socket -> bind -> listen -> accept -> send/recv

Simplified Example: Client or Server?

```
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
```

The above example is a simplified display of network function calls that would be executed by a program acting as a server. Realistically, this example is lacking error handling (such as calls to WSAGetLastError) and parameter setup.

1. *WSAStartup* initializes the Win32 sockets system
2. A socket is created with *socket*
3. *bind* attaches the socket to a port
4. *listen* sets up the socket to listen
5. *accept* hangs, waiting for a connection from a remote socket

WinINet API

- Higher-level than Winsock
- Wininet.dll
- Implements application-layer protocols
    - *InternetOpen* initializes a connection to the Internet.
    - *InternetOpenUrl* connects to a URL.
    - *InternetReadFile* read data from a file downloaded from the Internet.

Higher-level than Winsock
Wininet.dll is the dynamic link library imported for access to this functionality
Implements application-layer protocols

InternetOpen initializes a connection to the Internet.
InternetOpenUrl connects to a URL.
InternetReadFile read data from a file downloaded from the Internet.

Other Sources of Executable Code

- DLLs
- Processes
- Threads
- Services
- The Component Object Model (COM)
- Exceptions

In this next section, we will examine multiple ways malware can access and run code outside of a file that it may have originated from.

Dynamic Link Libraries (DLLs)

Exports functions to be imported by other programs

- Shared code exists once in memory, accessible to many processes
- Reusable code, reduces size and complexity of third-party software

Exports functions to be imported by other programs
Shared code exists once in memory, accessible to many processes
Reusable code, reduces size and complexity of third-party software

Abusing DLLs

- Store malicious code
  - Only one .exe, but many .dll per process
  - Use a .dll to load malware into another process
- Use Windows DLLs
  - Functions the malware imports offer significant insight into functionality
- Use third-party DLLs
  - Ex. Firefox DLL for network functions
  - Ex. custom DLL for crypto functions

Store malicious code

Only one .exe, but many .dll per process

Use a .dll to load malware into another process

Use Windows DLLs

Functions the malware imports offer significant insight into functionality

Use third-party DLLs

Ex. Firefox DLL for network functions instead of the Windows API

Ex. custom DLL for crypto functions

DLL Structure

PE file format, IMAGE_FILE_DLL header flag set

- More exports, few imports
- DLLMain entry point
  - Called to notify the DLL whenever
    - A process loads/unloads the library, creates a new thread, or finishes an existing thread
  - Allows the DLL to manage any per-process or per-thread resources

PE file format, IMAGE_FILE_DLL header flag set
More exports, few imports
DLLMain entry point
Called to notify the DLL whenever
A process loads/unloads the library, creates a new thread, or finishes an existing thread
Allows the DLL to manage any per-process or per-thread resources

Process

A program being executed by Windows

- Separate resources (handles, memory, etc)
- 1 process -> multiple threads
  - Process = container, thread = instructions
- Malware creates new, or modifies existing

Each program being executed by Windows manages its own resources by creating a process and being allocated virtual memory by the OS (note that this means two programs may appear to be using the same memory address, but will be using different physical memory so they won't interfere with each other). Malicious programs often persist by creating their own processes or injecting themselves into existing processes for stealth. A single process is comprised of one or more threads to be executed by the processor.

Creating a Process

CreateProcess

- *STARTUPINFO* (struct)
  - Handles to stdin, stdout, stderr
    - (redirect malicious stdout to a socket for remote shell) ->
    - Malware often extracts a binary from its resource section and runs w/ CreateProcess

```
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
```

In the above example, 10 different parameters are passed into the CreateProcess function (6), including a StartupInfo struct containing handles to stdin, stdout, and stderr, which all get set to the SocketHandle (initialized prior to this) at (1,2,3). At (4) an offset to data at 0x00403098 is moved into eax, and we can see that it contains the lpCommandLine argument when eax is pushed onto the stack later at (5). 7 of the 10 parameters passed are NULLs or flags, uninteresting in this case.

Threads

“Independent sequences of instructions executed by the CPU”

- All share memory space w/i a process
- Own registers and stack (thread context)
  - While running, assumes control of CPU and core
  - Thread context (snapshot) saved when control is passed
- Fibers (Microsoft)
  - Managed by a thread
  - Share a single thread context

“Independent sequences of instructions executed by the CPU”
All share memory space w/i a process
Have their own registers and stack (thread context)
While running, assumes control of CPU and core
Thread context (snapshot of stack/registers) saved when control is passed off to another thread and restored when control is returned

“In addition to threads, Microsoft systems use fibers. Fibers are like threads, but are managed by a thread, rather than by the OS. Fibers share a single thread context.”

CreateThread

- *lpStartAddress* parameter specifies the address of the **start** function
  - Execution begins here, until the function returns or process exits
- *lpParameter* parameter specifies a pointer to a variable to be passed to the thread
- Malware can use to
  - Load a DLL with LoadLibrary
  - Create input / output threads for sending / receiving data

lpStartAddress parameter specifies the address of the start function
    Execution begins here, until the function returns or process exits
lpParameter parameter specifies a pointer to a variable to be passed to the thread
Malware can use to
    Load a DLL with LoadLibrary + the name of the library to be loaded (calls DLLmain)
    Create input / output threads for sending / receiving data
        one listens on a socket/pipe and then outputs to stdin of a process
        the other reads from stdout and forwards to a socket/pipe

```
004016EE  lea    eax, [ebp+ThreadId]
004016F4  push   eax              ; lpThreadId
004016F5  push   0                ; dwCreationFlags
004016F7  push   0                ; lpParameter
004016F9  push  ❶offset ThreadFunction1 ; lpStartAddress
004016FE  push   0                ; dwStackSize
00401700  lea    ecx, [ebp+ThreadAttributes]
00401706  push   ecx              ; lpThreadAttributes
00401707  call  ❷ds:CreateThread
0040170D  mov    [ebp+var_59C], eax
00401713  lea    edx, [ebp+ThreadId]
00401719  push   edx              ; lpThreadId
0040171A  push   0                ; dwCreationFlags
0040171C  push   0                ; lpParameter
0040171E  push  ❸offset ThreadFunction2 ; lpStartAddress
00401723  push   0                ; dwStackSize
00401725  lea    eax, [ebp+ThreadAttributes]
0040172B  push   eax              ; lpThreadAttributes
0040172C  call  ❹ds:CreateThread
```

In this example, the first function at (1) would call ReadFile and send, and the second function at (3) would call recv and WriteFile.

Interprocess Coordination w/ Mutexes (mutants)

"Mutexes are global objects that coordinate multiple processes and threads."

- Like a talking stick
  - Owned by one thread at a time
  - Accessed with *WaitForSingleObject*
  - Releases with *ReleaseMutex*
- Host-based indicator
  - Hard-coded names
    - Used by two processes that aren't communicating otherwise

"Mutexes are global objects that coordinate multiple processes and threads."
Like a talking stick
Owned by one thread at a time
Accessed with WaitForSingleObject
Releases with ReleaseMutex
Host-based indicator
Hard-coded names
Used by two processes that aren't communicating otherwise

CreateMutex, OpenMutex

- Malware uses to ensure only one instance is running

```
00401000   push  offset Name     ; "HGL345"
00401005   push  0               ; bInheritHandle
00401007   push  1F0001h         ; dwDesiredAccess
0040100C ❶call  ds:__imp__OpenMutexW@12 ; OpenMutexW(x,x,x)
00401012 ❷test  eax, eax
00401014 ❸jz    short loc_40101E
00401016   push  0               ; int
00401018 ❹call  ds:__imp__exit
0040101E   push  offset Name     ; "HGL345"
00401023   push  0               ; bInitialOwner
00401025   push  0               ; lpMutexAttributes
00401027 ❺call  ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

Malware will commonly attempt to open a mutex to ensure that only one version of the malware is running at a time

At (1) the malware checks to see if there is an existing mutex named “HGL345” by calling OpenMutex
If the mutex does not exist, (2) will return 0 and the jump at (3) will be taken to the CreateMutex function at (5)
If the mutex already exists the jump is not taken and the function exits at (4)

Services

Tasks that run without their own processes/threads

- Code scheduled, run by service manager
  - No user input
  - Background applications
  - Run as SYSTEM or privileged account
  - Great for persistence (look at Autoruns)

Services are tasks in Windows that are able to run without their own processes and threads. They are instead run under the service manager (typically by svchost.exe) and without user input. Scheduled tasks are only created with administrative privileges, and most run as SYSTEM or another privileged account. Scheduled tasks are great for persistence, as tasks can be linked to a variety of triggers, including when the OS starts.

The *net start* command will display **running** processes, but Autoruns give more information.

Service Functions

- *OpenSCManager*
  - Called by any code that interacts with services
  - Returns a handle to the service control manager (SCM)
- *CreateService*
  - Adds service to the SCM
  - Specifies auto- or manual-start
- *StartService*
  - Used to start a service manually

OpenSCManager
Called by any code that interacts with services
Returns a handle to the service control manager (SCM)
CreateService
Adds service to the SCM
Specifies auto- or manual-start
StartService
Used to start a service manually

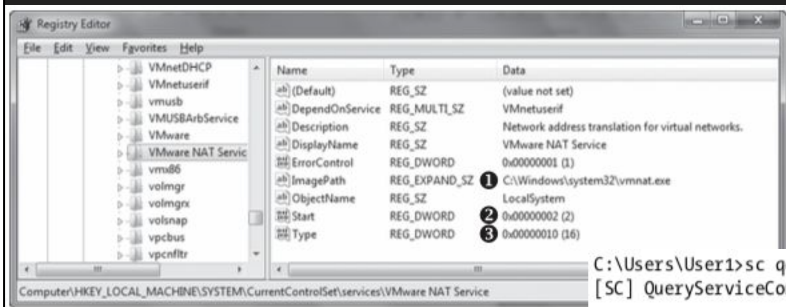
Service Types

- WIN32_SHARE_PROCESS
    - Code for service in DLL
    - Multiple services -> one process (svchost.exe)
- WIN32_OWN_PROCESS
    - Independent process
- KERNEL_DRIVER
    - Loads code into kernel

(Three common service types used by malware)

WIN32_SHARE_PROCESS
    Code for service in DLL
    Multiple services -> one process (svchost.exe)
WIN32_OWN_PROCESS
    Independent process
KERNEL_DRIVER
    Loads code into kernel

Services and the Registry



HKLM\SYSTEM\CurrentControlSet\Services

```
C:\Users\User1>sc qc "VMware NAT Service"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: VMware NAT Service
        TYPE               : 10  ❶WIN32_OWN_PROCESS
        START_TYPE         : 2    AUTO_START
        ERROR_CONTROL      : 1    NORMAL
        BINARY_PATH_NAME   : C:\Windows\system32\vmnat.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : VMware NAT Service
        DEPENDENCIES       : VMnetuserif
        SERVICE_START_NAME : LocalSystem
```

Listing 7-10: The query configuration information command of the SC program

Configuration information for local services is stored in subkeys under HKLM\SYSTEM\CurrentControlSet\Services

The top left image shows a view of entries for the VMware NAT Service service, including the path the the code for the service (1), the type of start - automatic or manual at (2) with a value of 0x02 (AUTO_START), and the type of process (0x10 = WIN32_OWN_PROCESS) at (3).

The bottom right image displays the output of using the qc (query configuration) command of the SC (service control) program. This command essentially displays the same information as what exists in the registry entry.

The Component Object **Model** (COM)

Interface **standard** for abstracted inter-software communication

- Client-server framework
- Thead must call *OleInitialize* or *CoInitializeEx* prior to calling COM library functions
    - Need to identify object

Microsoft describes COM as “a platform-independent, distributed, object-oriented system for creating binary software components that can interact”, though note that it is not restricted to use by object-oriented systems specifically. COM is a standard which allows software components to interact (think interprocess communication) without knowledge of each other’s implementation details because they are designed to behave in an expected manner. The objects themselves act as producers (servers) and are called by consumers (clients) accessing their functionality. COM is used by the OS, Microsoft applications, and third-party applications. Learning how to recognize COM objects in use is therefore handy when analyzing malware, once you see a call to *OleInitialize* or *CoInitializeEx,* search for identifiers of the object (covered next).

COM Object Identifiers

- Accessed via globally unique identifiers (GUIDs)
  - Classes identified with class identifiers (CLSIDs)
  - Interfaces identified with interface identifiers (IIDs)
- *CoCreateInstance* called to access COM functionality
  - (HKCU | HKLM)\ SOFTWARE\Classes\CLSID\ have class file info
  - Returns a pointer to a struct of function pointers
- *Navigate* launches browser to a URL
  - Part of *IWebBrowser2* interface
    - Must be implemented by a program, called the **class** (IE in this case)

```
00401024 lea     eax, [esp+18h+PointerToComObject]
00401028 push    eax                 ; ppv
00401029 push    ❶offset IID_IWebBrowser2 ; riid
0040102E push    4                   ; dwClsContext
00401030 push    0                   ; pUnkOuter
00401032 push    ❷offset stru_40211C ; rclsid
00401037 call    CoCreateInstance
```

Accessed via globally unique identifiers (GUIDs)
Classes identified with class identifiers (CLSIDs)
Interfaces identified with interface identifiers (IIDs)
CoCreateInstance called to access COM functionality
(HKCU | HKLM)\ SOFTWARE\Classes\CLSID\ have host file info
Returns a pointer to a struct of function pointers
An example: *Navigate* launches browser to a URL
(Part of *IWebBrowser2* interface)
An interface must be implemented by a program,
called the **class** (IE in this case)

In the example above, clicking the structures at (1) and (2) in IDA would display the IDs for the interface and class respectively. In this case the IID represents the *IWebBrowser2* interface and the CLSID represents Internet Explorer. The pointer returned from the call to *CoCreateInstance* points to a structure containing pointers to functions that can be called via their offsets within the structure.

Exploring COM Interface Structures

To find which function is called:

1. [insert] -> Add Standard Structure
2. <InterfaceName>Vtbl
   ex. *IWebBrowser2Vtbl*
3. Right-click offset (1) to change label
   2Ch -> *Navigate*

```
0040105E  push    ecx
0040105F  push    ecx
00401060  push    ecx
00401061  mov     esi, eax
00401063  mov     eax, [esp+24h+PointerToComObject]
00401067  mov     edx, [eax]
00401069  mov     edx, [edx+❶2Ch]
0040106C  push    ecx
0040106D  push    esi
0040106E  push    eax
0040106F  call    edx
```

Check header files for interface if function not in IDA

Above we can see the COM client code calling a function whose location is stored at an offset in the previously returned structure.

At 0x00401063 the pointer to the COM object is dereferenced to access the first value in the structure (a pointer to a table of function pointers)
The following *mov edx, [eax]* dereferences the pointer to the table of function pointers.
The final *mov* dereferences the pointer to the function at offset 0x2C in the table, the *Navigate* function that is called at the end of this disassembly

To find which function is called:

[insert] -> Add Standard Structure
<InterfaceName>Vtbl
ex. IWebBrowser2Vtbl
Right-click offset (1) to change label
2Ch -> Navigate
Check header files for interface if function not in IDA

COM Continued

- COM objects as DLLs
  - Loaded into process space of COM client exe
  - CLSID subkey will be *InprocServer32* instead of *LocalServer32*
- COM Server must export
  - DllCanUnloadNow, DllGetClassObject, DllInstall, DllRegisterServer, and DllUnregisterServer
- Malicious servers usually use Browser Helper Objects (BHOs)
  - Third-party plugins for IE
  - “allows them to monitor Internet traffic, track browser usage, and communicate with the Internet, without running their own process”

While the previous example started IE as its own process when *CoCreateInstance* was called, this does not always happen:
COM objects as DLLs
Loaded into process space of COM client exe
CLSID subkey will be InprocServer32 instead of LocalServer32

Malicious programs may implement a COM server for use by other applications
COM Server must export
DllCanUnloadNow, DllGetClassObject, DllInstall, DllRegisterServer, and DllUnregisterServer
Malicious servers usually use Browser Helper Objects (BHOs)
Third-party plugins for IE
“allows them to monitor Internet traffic, track browser usage, and communicate with the Internet, without running their own process”

Structured Exception Handling (SEH)

Error -> transfer execution to a special routine

- Hardware, software, or manual (with *RaiseException* call)
- In 32-bit systems, SEH info -> stack

```
01006170  push  ❶offset loc_10061C0
01006175  mov     eax, large fs:0
0100617B  push  ❷eax
0100617C  mov     large fs:0, esp
```

Structured Exception Handling is how exceptions are handled in Windows. When an exception occurs, such as an error from dividing by zero (hardware) or an invalid memory address (software), execution is transferred to a special routine which resolves the error. Above, we can see the SEH frame loaded onto the stack at (1), followed by the fs:0 segment register containing a pointer to an address for storing exception information. The stack contains the location of an exception handler, as well as the exception handler used by the caller at (2).

1. Windows looks in fs:0 for the stack location that stores the exception information
2. The exception handler is called
3. Execution returns to the main thread

Exception Recursion

Nested: current frame handler -> caller’s handler -> … -> top-level handler crashes application

Using a stack overflow to overwrite a pointer to an exception handler -> code execution

Exception handlers are nested: current frame handler -> caller’s handler -> … -> top-level handler crashes application

Using a stack overflow to overwrite a pointer to an exception handler -> code execution

User vs Kernel Mode (Processor privilege levels)	
Most code	OS, hardware drivers
Own memory, permissions, resources	Share resources, memory (single process context)
Use API to change kernel state or hardware (SYSENTER, SYSCALL, or INT 0x2E)	Manipulate user-space code
OS terminates process on crash	OS crashes (harder to develop for)
Privileged / unprivileged users	Privileged, no OS audits, great for malware

All of the functions covered thus far have been user-mode functions, which have their own virtual memory space, security permissions, and resources. The must call upon well-defined interfaces such as the Windows API to interact with the kernel or hardware. If a program crashes, the OS is able to terminate its process and recover the resources. In user-mode there is distinguishment between running privileged and unprivileged code based on user/account permissions.

Kernel-level code mostly involves the OS and hardware drivers. Code in the kernel shares resources and memory space under a single process context, and therefore will crash the OS if things go awry. The Kernel may directly manipulate user-space code and is not audited by any OS auditing features - many tools such as antivirus programs or firewalls run at the kernel level in order to access and monitor everything on the system. For malware, this means that it may bypass these tools and run privileged code if run in kernel-mode, however it is significantly harder to develop and therefore mainly a feature of only sophisticated malware.

The Native API

Lower-level interface (ntdll APIs and structures), commonly used in malware

- Bypasses Windows API (evasion)
- Additional functionality

Traditional call:

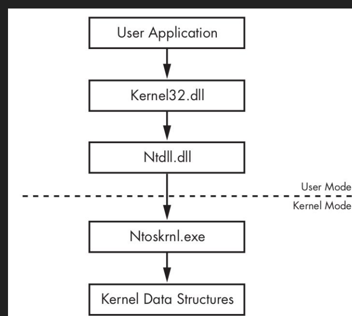


Figure 7-3: User mode and kernel mode

Bypassing poor security:

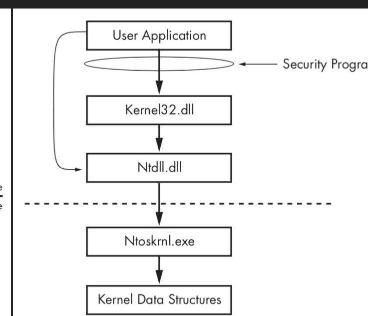


Figure 7-4: Using the Native API to avoid detection

The Native API is the set of functions and structures called upon by ntdll in the traditional chain of execution when a Windows API function is called. The Native API is not meant to be accessed by user-space programs directly but there is nothing to prevent it, and malware often takes advantage of this to bypass calls to Kernel32.dll which may be monitored by a security program (though better security programs would also be monitoring calls to everywhere), and to use functionality that is not viable in the standard Windows API.

Native API Functions

- NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread, NtQueryInformationFile, and NtQueryInformationKey
  - Allow you to view and set more attributes than Win32 calls
- NtContinue
  - Return from an exception, transfer control to specified location
    - Usually main thread, but maliciously can be used to confuse an analyst
- Functions may also be prefixed by “Zw”
- Native applications (rely entirely on Native API) are rare
  - Shows in PE header subsystem field

NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread, NtQueryInformationFile, and NtQueryInformationKey

Allow you to view and set more attributes than Win32 calls

NtContinue

Return from an exception, transfer control to specified location

Usually main thread, but maliciously can be used to confuse an analyst

Functions may also be prefixed by “Zw”

Native applications (rely entirely on Native API) are rare

Shows in PE header subsystem field

Resources

WinObj Object Manager namespace viewer (Sysinternals)

Autoruns (Sysinternals)

Windows NT/2000 Native API Reference by Gary Nebbett
ntinternals.net Native API help