



# Abertay University

## Manipulating The Windows API

Callum Leonard

CMP320: Advanced Ethical Hacking

BSc (Hons) Ethical Hacking Year 3

2022/23

*Note that Information contained in this document is for educational purposes.*

# Abstract

---

The Windows API allows direct interfacing with multiple Windows components, including system services, user input, diagnostics, devices, security, and more. In recent years hackers have found new ways to utilise the Windows API to perform malicious activities such as developing malware, evading native and 3rd party antiviruses, and performing privilege escalation via token privilege manipulation. The Windows API manipulation script was built to grow the understanding and increase cyber resilience in this field. The script focuses on manipulating process access tokens to enable vulnerabilities, spawning processes, and killing processes. This report aims to inform, detail, evaluate, and discuss how the Windows API Manipulation script was created. More specifically, the process of moving from theorised implementation to actual script completion.

Coherent script development was achieved by utilising the official Windows API documentation which provided vital information on API call structures, parameters, return types, and the order of different API calls to execute. Each script function was individually created by issuing associated Windows API calls to obtain the initial desired result of the function. Executing each function independently resulted in the successful change in token access privileges of a targeted Windows process, the successful termination of a targeted Windows process, and the successful creation of a Windows process. Each function was subject to necessary error testing by feeding arbitrary input which returned no unforeseen errors or unexpected results.

The Windows API manipulation script proved to be very effective in all the functionality that it offers. More specifically, each function proved to be consistently reliable and successful in either targeting a process or handling an error. Future work in this field would include increasing cyber awareness by adding new script functionality to exploit vulnerabilities created by the script. Specific emphasis on vulnerability countermeasures would also be covered to ensure cyber resilience is built-up to the niche threat.

# Contents

---

1	Introduction .....	1
1.1	Background .....	1
1.2	Aims.....	2
1.3	Tools Used.....	3
2	Script Prerequisites .....	4
2.1	Windows API Call Structure .....	4
2.2	Windows API Call Extensions .....	4
2.3	Windows DLL's .....	5
2.4	Windows API Call Error Handling.....	5
3	Program and Development.....	6
3.1	Overview of Program and Development .....	6
3.2	Script Imports.....	6
3.3	Code Walkthrough (Function: Collecting User Input).....	7
3.3.1	Function Testing.....	8
3.4	Code Walkthrough (Function: Token Privilege Manipulation) .....	10
3.4.1	Function Iteration .....	18
3.4.2	Function Testing.....	19
3.5	Code Walkthrough (Function: Terminating Processes) .....	24
3.5.1	Function Testing.....	25
3.6	Code Walkthrough (Function: Spawning Processes) .....	27
3.6.1	Function Testing.....	29
4	Discussion.....	31
4.1	General Discussion.....	31
4.2	Script Evaluation .....	31
4.2.1	Function: Collecting User Input Evaluation.....	31
4.2.2	Function: Token Privilege Manipulation Evaluation .....	32
4.2.3	Function: Killing Processes Evaluation .....	32
4.2.4	Function: Spawning Processes Evaluation .....	33
5	Future Work .....	34
6	References .....	35
7	Appendices.....	38

Appendix A – Token Privilege Change Structures..... 38

Appendix B – Restoring Token Privileges..... 39

Appendix C – Destroying Token Privileges..... 42

Appendix D – Spawning Process Structures ..... 45

# 1 INTRODUCTION

## 1.1 BACKGROUND

In 2022 the Windows Operating System recorded over 907 vulnerabilities being published against it, far higher than its main competitor Apple Inc's MacOS (*Windows Vulnerability Report 2022* 2021). A breakdown of the vulnerabilities published yielded 132 critical vulnerabilities with the majority related to privilege escalation. Alongside the published Windows vulnerabilities, 60 million new malware samples were also recorded within the first three quarters of 2022, with 95.6% of samples recorded pertaining to Windows (*Windows Malware Statistics* 2022).

Launching within the Windows OS on the 20th of November 1985, the Windows API provides native functionality to interact with a multitude of key components of the Windows Operating System such as system services, devices, diagnostics, user input, security, and more (*Windows API index* 2022). Successful utilisation of the Windows API allows for easier development of applications due to pre-defined functions and also allows for application deployment to be successful across all versions of Windows whilst taking advantage of each version's unique capabilities (*Microsoft: Why the Windows API* 2019).

As with almost everything in the modern internet era, the Windows API has been successfully abused by hackers to perform a host of malicious actions/attacks ranging from manipulating Windows Access Token Privileges resulting in user account impersonation (ultimately the NT Authority/SYSTEM) (*Token manipulation attacks* 2020) to creating various types of malware such as keyloggers to record user keystrokes (*Windows API Keylogger* 2019). Figure 1 shows the Windows API call structure to successfully achieve a working user impersonation program.

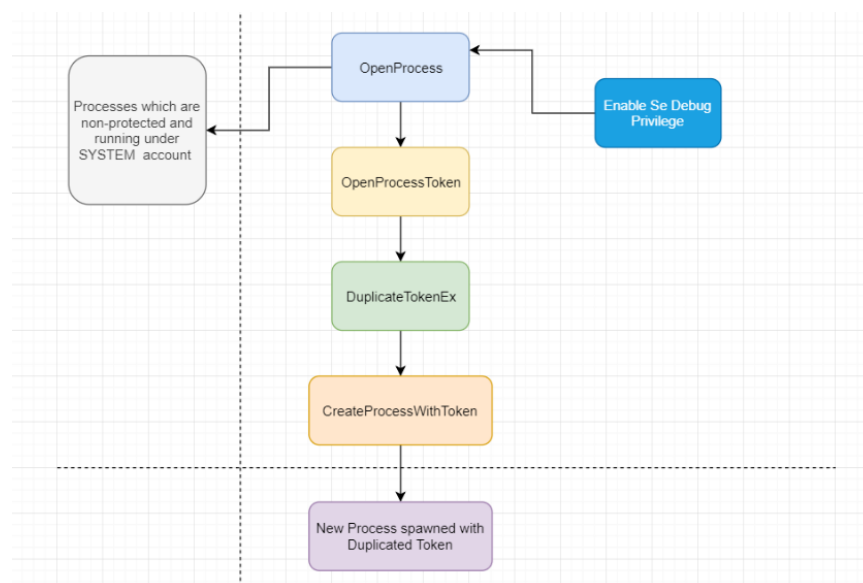


Figure 1 – Account Impersonation Windows API Call Structure

## 1.2 AIMS

---

This report aims to inform, detail, evaluate, and discuss how the Windows API Manipulation script was built. More specifically, the process of moving from theorised implementation to actual script completion. This overall aim encapsulates several sub-aims:

- Inform the report reader about the general structure of Windows API calls and DLLs.
- Efficiently detail how the code of the script works.
- Evaluate the script results and if the script achieved the initial theorised results.
- Discuss future work regarding the potential exploitation of vulnerabilities facilitated by the script.

## 1.3 TOOLS USED

---

The table below details the tools used throughout the development of Windows API Manipulation script.

Tool Table

TOOL/SOFTWARE	VERSION	OPERATING SYSTEM
Notepad++	8.5	LINUX
Google Chrome	111.0	WINDOWS, LINUX
Sysinternals – Process Explorer	17.02	WINDOWS
Windows Task Manager	N/A	WINDOWS

Note that all figures throughout the report were manually created using Figma, and can be accessed by following this link:

<https://www.figma.com/file/3rcjIXdSNE7xoV8H0XVlhA/Untitled?node-id=0%3A1&t=fPdTbGdXV4GU8MJD-1>

## 2 SCRIPT PREREQUISITES

### 2.1 WINDOWS API CALL STRUCTURE

---

Windows API calls are similar to typical functions across multiple different programming languages. Each Windows API call has its own pre-defined structure with its associated return type, parameters, and datatypes. Figure 1 below details an “OpenProcess” API call which returns a handle to a specified process. It is also important to note that all 3 parameters of the “OpenProcess” API call must all be populated for the API call to be successful (unless optional). Figure 2 details the breakdown of the API call structure.

```
HANDLE OpenProcess(  
    [in] DWORD dwDesiredAccess,  
    [in] BOOL bInheritHandle,  
    [in] DWORD dwProcessId  
);
```

Figure 1 – OpenProcess Windows API Call Structure

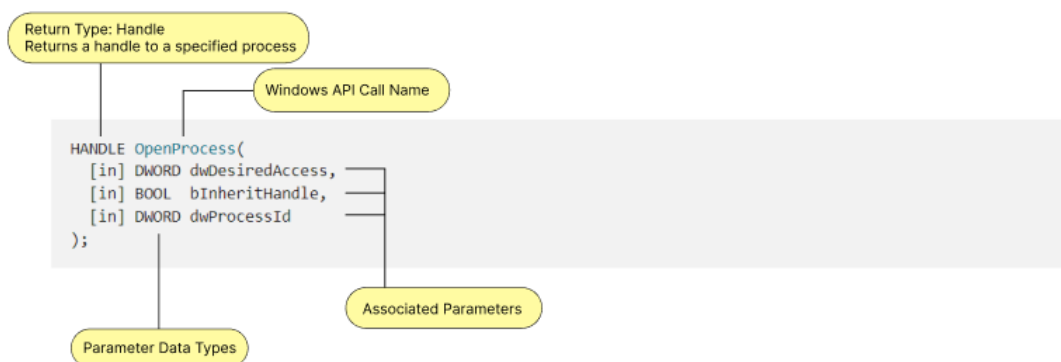


Figure 2 – OpenProcess Windows API Call Structure Breakdown

### 2.2 WINDOWS API CALL EXTENSIONS

---

Windows API call extensions are typically denoted by an additional character at the end of an API call and indicate different character sets to be accepted by the API calls associated parameters.

- ‘W’ – Represents a Unicode string
- ‘A’ – Represents an ANSI string
- ‘Ex’ – Provides additional functionality or additional parameters to the API call

Note that a string being passed to an API call with extension ‘A’ must have appropriate ANSI encoding.



## 2.3 WINDOWS DLL's

---

All Windows API calls are stored within Windows DLL (Dynamic Link Library) files. It is important to note that a handle to a specific DLL must first be achieved before an associated Windows API call is made. Specific Windows API calls are separated into different DLL files. The bullet points below illustrate information relating to different Windows DLL files:

- Advapi32.dll – Handles Windows Registry, User Accounts and Services
- Netapi32.dll – Responsible for all Networking components
- Kernel32.dll – Pertains to interacting with processes, threads, and file systems.
- User32.dll – Handles fonts, graphical user interface, mouse and more
- Shell32.dll – Handles Windows Shell functionality

## 2.4 WINDOWS API CALL ERROR HANDLING

---

Although touched upon later in this report, every Windows API call has a return value for indicating whether the call was successful or unsuccessful. Throughout the code, all API calls have appropriate error checking. Figure 1 below details the code structure of how errors are handled throughout the script after a Windows API call has been executed.

```
253 adjtkn_error = ctypes.WinDLL("Kernel32.dll")
254 if adjTkn_resp != 0:
255     print("\n[+] All Token Privilege Modifications Successful [+]")
256 else:
257     print("[-] Token Privilege Modifications Unsuccessful. Error code: {0}".format(adjtkn_error))
```

Figure 1 – Windows API Call Error Handling

As shown in Figure 1 line 254, an if condition checking the response of the “AdjustTokenPrivileges” API call was conducted. By reading the documentation of the API call, it was apparent that a successful call would have a return value of “non-zero” hence if the response is not equal to 0 the call was successful. It is important to note that different API calls have different return values which indicate successful calls. Line 253 is responsible for interfacing with the kernel32.dll file with a primary focus on utilising the “GetLastError()” function when required. This function displays an error code which can be searched online to help troubleshoot the error generated. See section 6 (*GetLastError function (errhandlingapi.h) - win32 apps 2021*) and (*System error codes (0-499) (winerror.h) - win32 apps 2022*) for error documentation.

# 3 PROGRAM AND DEVELOPMENT

## 3.1 OVERVIEW OF PROGRAM AND DEVELOPMENT

---

This section of the report initially details the required script imports to enable the script to successfully compile. Furthermore, this section also covers each function residing within the script describing in-depth what the purpose is. Code analysis and function testing, specifically collecting data and results from the function and error testing is covered.

## 3.2 SCRIPT IMPORTS

---

Imports for the Windows API Manipulation script were crucial in providing a working script. In order to issue Windows API calls the library “ctypes” was used (*Ctypes - a foreign function library for Python* 2021). Importing ctypes allowed for interfacing directly to a Windows DLL file by creating a handle to it. After a handle to the DLL file was achieved the associated API call was performed. The module “argparse” was also imported and allows for user-friendly command-line arguments (*Argparse - parser for command-line options* 2020). The script also made use of the “re” import (regex library), the purpose of the regex within this code was to strip unnecessary characters from a returned process id (*RE - regular expression operations* 2019). Lastly, “from time import sleep” was imported which allowed the sleep function to be called causing a delay before executing the next line of code (*Python time sleep function* 2022). Figure 1 below details all imports previously discussed.

```
42 import ctypes
43 import argparse
44 import re
45 from time import sleep
```

Figure 1 – Script Imports

### 3.3 CODE WALKTHROUGH (FUNCTION: COLLECTING USER INPUT)

---

The function associated with creating command line arguments and capturing user input was defined as “args” and made use of the previously discussed library “argparse”. Figure 1 below exhibits the function declaration.

```
417 def args():
```

Figure 1 – “args” Function Declaration

Before individual arguments could be created for the command line interface “argparse.ArgumentParser” had to be declared. This acts as a container for all argument specifications and allows the parser to be applied as a whole. As it is a container a general description was applied describing the purpose of the script and its functionality. Line 421 of Figure 2 below details the argparse container and the description of the script.

```
421 parser = argparse.ArgumentParser(description='Use this script to spawn and kill Windows processes.\nThis
```

Figure 2 – Argparse container

After the argparse container declaration, individual arguments were created by making use of the “add\_argument\_group()” and “add\_argument()” functions. This works by binding individual arguments to the parser itself. Figure 3 below details the argument group and the individual arguments “-s”, “-k” and “-p”. The argument “-s” spawns Windows processes, “-k” kills Windows processes, and “-p” manipulates a specific processes access token privileges.

```
422 all_args = parser.add_argument_group('All arguments')
423 all_args.add_argument("-s", help="The application to spawn | Usage: py
424 all_args.add_argument("-k", help="The current running application to k
425 all_args.add_argument("-p", help="Enable and Disable all Windows token
```

Figure 3 – Individual Arguments

As the arguments had been declared the last step involved running the parser and placing the data collected from the arguments into an argparse namespace object. This was done via the statement “args = parser.parse\_args()”. Lastly, to allow for easier code readability each argument's data was placed inside a variable, this also enabled future function calls to contain the specific variables as parameters. Figure 4 below details the function to run the parser and the associated variables being initialized to arguments.

```
427 args = parser.parse_args()
428
429 spawn = args.s
430 kill = args.k
431 en_dis = args.p
```

Figure 4 – Parser execution and variable initialization

The last step in completing the function involved creating multiple conditions which checked whether a specific argument was selected by the user and proceed to subsequently execute the arguments

associated function. Figure 5 below details the process of executing functions based on detected arguments.

```
433 | if args.s == None and args.k == None and args.p == None:
434 |     sleep(0.35)
435 |     print("\n[-] Error: Script must take 1 argument \n[-] Script Usage: -h for help")
436 |     exit(1)
437 |
438 | elif args.s != None and args.k == None and args.p == None:
439 |     lpApplicationName = str(spawn)
440 |     spawnProc(lpApplicationName)
441 |
442 | elif args.k != None and args.s == None and args.p == None:
443 |     lpWindowName = ctypes.c_char_p(kill.encode('utf-8'))
444 |     killProc(lpWindowName)
445 |
446 | elif args.p != None and args.s == None and args.k == None:
447 |     lpWindowName2 = ctypes.c_char_p(en_dis.encode('utf-8'))
448 |     alterPrivs(lpWindowName2)
449 | else:
450 |     print("\n[-] Error: Script must only take 1 argument \n[-] Script Usage: -h for help")
451 |     exit(1)
```

Figure 5 – Argument data checking

To execute a specific function based on an argument, several if conditions were used. Specifically, any argument that did not contain any data had a value of null indicating that the argument was not selected. Therefore, appropriate checking could be conducted to identify which argument was currently populated. If the current argument being checked was not null but its siblings were the current arguments associated function was executed. Lastly, if no data was detected in any argument the program would terminate by issuing “exit(1)”.

### 3.3.1 Function Testing

After the function and all of its arguments were completed, several tests were conducted to ensure that the function performed as expected.

Process for function testing:

- Execute function with no parameters
  - Execute function with an invalid parameter
    - Execute function with the “help” parameter
- Ensure that the functions “help” argument provides sufficient information relating to the script

#### 3.3.1.1 Captured Results

The initial theorised results upon successful implementation of the argument function was the ability to reject invalid arguments, display useful help messages about the script, and appropriately handle no given arguments.

##### 3.3.1.1.1 Function Execution

Figure 1 below depicts the function being executed with no given parameters, an invalid parameter, and the “help” parameter. Analysis of the function execution below shows that all 3 function execution tests are appropriately handled as expected. Providing no argument to the function results in it generating an error informing 1 argument to be entered. Providing an invalid argument causes the function to generate

an error informing the user of the “unrecognized argument”. Providing the function with the “help” argument prompts the function to deliver clear and concise information about arguments and the script.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py

[-] Error: Script must take 1 argument
[-] Script Usage: -h for help

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -i
usage: script.py [-h] [-s S] [-k K] [-p P]
script.py: error: unrecognized arguments: -i

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -h
usage: script.py [-h] [-s S] [-k K] [-p P]

Use this script to spawn and kill Windows processes. This script may also be used to destroy and restore process privileges.

options:
  -h, --help  show this help message and exit

All arguments:
  -s S      The application to spawn | Usage: py script.py -s cmd.exe
  -k K      The current running application to kill | Usage: py script.py -k "Task Manager"
  -p P      Enable and Disable all Windows token privileges for the specificed application | Usage: py script.py -p "Command Prompt"

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

*Figure 1 – Function argument execution*

### 3.4 CODE WALKTHROUGH (FUNCTION: TOKEN PRIVILEGE MANIPULATION)

Figure 1 details below exhibits the entire approach taken which includes multiple semantic API calls and DLL interfaces to successfully achieve token privilege destruction or restoration.

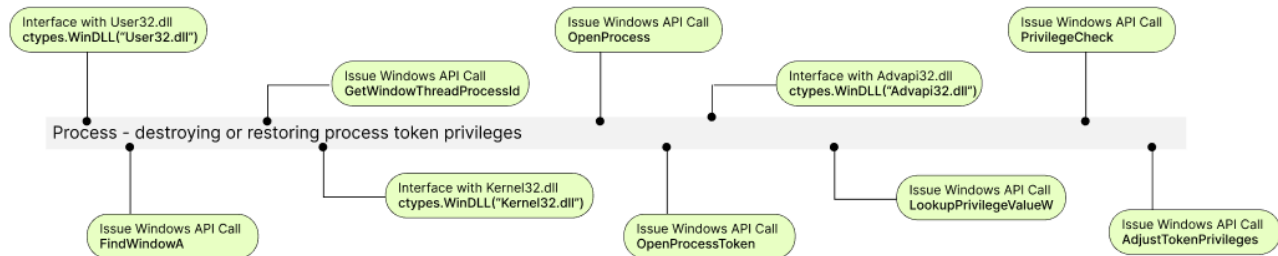


Figure 1 – Approach Taken - Token Privilege Destruction/Restoration

The function associated with performing the token privilege destruction or restoration was defined as “alterPrivs”. To automate and start the process of API calls, it took 1 argument which is a “ctypes.c\_char\_p” (a pointer to a string, passed in from argparse) and also was UTF-8 encoded. The reason for the encoding is due to the imminent Window API function call “FindWindowA” which must accept a “UTF-8” encoded parameter. Figure 2 details the function declaration with argument.

```
12 def alterPrivs(lpWindowArg):
```

Figure 2 – “alterPrivs” Function Declaration

It is important to note that the function argument “lpWindowArg” is populated with the argparse argument ‘-p’ process name. Figure 3 shows the content of argument ‘p’ being assigned to the variable “en\_dis”. Figure 4 exhibits logic checking, if the condition is met the content of “en\_dis” is appropriately encoded as required by the Microsoft “FindWindowA” API call and is passed as an argument to the “alterPrivs” function. See section 6 (Findwindowa function (winuser.h) - win32 apps 2022) for the FindWindowA API call structure documentation.

```
467 en_dis = args.p
```

Figure 3 – Argument P being assigned to variable en\_dis

```
488 elif args.p != None and args.s == None and args.k == None:
489     lpWindowName2 = ctypes.c_char_p(en_dis.encode('utf-8'))
490     alterPrivs(lpWindowName2)
```

Figure 4 – Logic checking and necessary encoding

After the encoded argument previously discussed is passed into the function it can be used directly with an API call. However, a handle must first be setup to the relevant DLL. Note that Microsoft details relevant DLL’s for associated Windows API calls. See figure 5 below.

```
63 user_handle = ctypes.WinDLL("User32.dll")
64 priv_choice = int(input("Press 1 to destroy process token privileges or 2 to restore process token privileges: "))
65 os.system('cls')
66
67 lpClassName = None
68 findWindowAPI_resp = user_handle.FindWindowA(lpClassName, lpWindowArg)
69
70 kernel_handle = ctypes.WinDLL("Kernel32.dll")
71 get_error = kernel_handle.GetLastError()
```

Figure 5 – User32 Handle, FindWindowA API Call, Kernel32 Handle

Figure 6 code line 63 details the code needed to interface with a specific DLL file. After a handle has been established to a DLL, the relevant API call parameters can be populated, and the API call executed as detailed on line 68. The parameters passed into the FindWindowA API call include “lpClassName” which is set to null as the Window retrieval was purely based on “lpWindowArg”. The return value of this function is a direct handle to the specified window name (lpWindowArg) which was entered by the user. E.g., if the user enters “Command Prompt” a handle to the window will be achieved. See section 6 (*Findwindowa function (winuser.h) - win32 apps 2022*) for the FindWindowA API call structure. Figure 6 details the script producing a successful API call execution and obtaining a handle to the specified process. Lastly, on line 70 a handle to the DLL “Kernel32.dll” was created. The reasoning behind this handle creation was due to imminent API calls which were only accessible via interfacing with the Kernel32.dll.

```
[+] Successfully Obtained Handle To Process
```

Figure 6 – Script Output, Successful Handle To Process

As a handle to the specified window has been achieved the next step is obtaining the process id of the window. This was obtained by using the “GetWindowThreadProcessId” API call. If successful this function returns the process id (the identifier of the thread that created the initial window). Figure 7 details the Windows API call to GetWindowThreadProcessId with the relevant parameter. See section 6 (*GetWindowThreadProcessId function (winuser.h) - win32 apps 2022*) for a direct link to GetWindowThreadProcessId API call structure.

```
83     getID = ctypes.c_ulong()
84     procAPI_resp = user_handle.GetWindowThreadProcessId(findWindowAPI_resp, ctypes.byref(getID))
```

Figure 7 – GetWindowThreadProcessId API Call

As detailed in Figure 7 the GetWindowThreadProcessId takes 2 parameters, the first parameter being the initial FindWindowA handle. The second parameter is a pointer to a variable that receives the new process id, this is declared as getID on line 83. As it is a pointer it needs to be passed appropriately, this is done by passing the argument by reference. By passing in the window handle as the first parameter the getID will be updated to the process id of the window handle. Figure 8 below details the script producing the output for a successful API call to obtain the process id.

```
[+] Successfully Obtained Process ID
```

Figure 8 – GetWindowThreadProcessId API Call

After the id of the process has been obtained the next step was to issue the Windows API call “OpenProcess”. See section 6 (*OpenProcess function (processthreadsapi.h) - win32 apps 2023*) for the “OpenProcess” Windows API call documentation. This call returns an open handle to the specified process and allows for future API calls to modify the process’s token privileges. Figure 9 below details the API call itself and its necessary parameters.

```
97     PROCESS_ALL_ACCESS = (0x000F0000 | 0x00100000 | 0xFFF)
98     dwDesiredAccess = PROCESS_ALL_ACCESS
99     dwProcessId = getID
100    bInheritHandle = False
101
102    procAPI_resp = kernel_handle.OpenProcess(dwDesiredAccess, bInheritHandle, dwProcessId)
103    p_ID = str(getID)
104    p_ID = re.sub('[^0-9]', '', p_ID)
105
106    if procAPI_resp == 0:
107        sleep(0.25)
108        print("[+] Could Not Open Specified Process. Error code: {0}".format(get_error))
109        exit(1)
110    else:
111        print("[+] Successfully Opened Process ID: " + p_ID)
112        sleep(0.25)
```

Figure 9 – OpenProcess API Call and Parameters

As shown in Figure 10 the OpenProcess API call takes in 3 parameters. The first parameter “dwDesiredAccess” is the level of access that the handle has to the process. In line 97 above “PROCESS\_ALL\_ACCESS” is used which grants all possible rights to a process object. See section 6 (*Process security and access rights - win32 apps 2022*) for the process security and access rights documentation. The second parameter “bInheritHandle” was set to false as inheriting the handle in this instance was not necessary. The last parameter “dwProcessId” serves as the identifier of the process to be opened. The previously discussed pointer “get\_ID” contains the current PID of the process in question and is therefore assigned to the dwProcessId parameter. It is important to note that after the API call on line 102, regex was used. The purpose of regex was to strip all arbitrary characters and only deliver the process id for future command line output. Condition checking was then conducted on lines 106 to 112, following Windows documentation, if the call fails the value will be null, else it was successful. Figure 10 below details the script producing the output for a successful API call to open the process and also display the process id.

```
[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 40176
```

Figure 10 – Opening Process API Call and Displaying Process ID

After a successful open handle to the process was achieved the next step involved accessing the processes token. This was conducted by using the API call “OpenProcessToken” which directly opens an access token associated with a process. It is important to note that OpenProcessToken is needed before any token modifications. See section 6 (*OpenProcessToken function (processthreadsapi.h) - win32 apps 2022*) for the “OpenProcessToken” Windows API call documentation. Figure 11 below details the API call itself and its necessary parameters.



```

115     adv_handle = ctypes.WinDLL("Advapi32.dll")
116
117     hProcess = procAPI_resp
118     ProcessHandle = hProcess
119     TokenHandle = ctypes.c_void_p()
120     DesiredAccess = TOKEN_ALL_ACCESS
121
122     opnProc_resp = adv_handle.OpenProcessToken(ProcessHandle, DesiredAccess, ctypes.byref(TokenHandle))
123     if opnProc_resp != 0:
124         sleep(0.25)
125         print("[+] Handle To Process Access Token Succesfully Opened \n")
126         sleep(0.25)
127     else:
128         print("[-] Failed To Open Handle To Process Access Token. Error code: {}".format(kernel_handle.GetLastError()))
129         exit(1)
130         sleep(0.25)

```

Figure 11 – Opening Process Token API Call

As defined in the Windows documentation the OpenProcessToken API call required the DLL file “Advapi32.dll”. On line 115 of Figure 12 a handle to interface with the associated DLL was created. The first parameter “hProcess” requires a handle to the process in question and must also have the associated access rights. In Figure 10 above (line 102) the open process handle was captured within the variable “procAPI\_resp” and therefore was assigned to hProcess to satisfy the requirements of the first parameter. The second parameter “DesiredAccess” specifies the type of access to the access token. When the OpenProcessToken API call is made it gets a handle to the processes access token and the system checks the access rights. To allow for control of the token privileges the “TOKEN\_ALL\_ACCESS” structure was made (See Appendix A). See section 6 (*Token\_privileges (winnt.h) - win32 apps 2021*) for token access rights documentation. The last parameter “TokenHandle” is an empty handle, the purpose of this is to identify the new access token after the function returns. As the parameter is a pointer to a handle it was passed by reference. The TokenHandle parameter is important as it now contains a populated handle to a running processes access token. This is explicitly needed for future API calls in which a handle is needed to an access token to check token privileges. Figure 12 below details the script producing the output for a successful API call to obtain a handle to the process access token.

```

[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 40176
[+] Handle To Process Access Token Succesfully Opened

```

Figure 12 – Obtaining Handle To Process Access Token

Note that the process detailed below is an iterative process and is conducted within a for loop. Before discussing the for loop the general API call structure will first be discussed.

As a direct handle to the access token was gained the next step was to use the API call “LookupPrivilegeValueW” to retrieve the “LUID’s” existing within the token. It is important to note that a luid is a locally unique identifier that is associated with token privileges. Figure 13 below details the API call itself and its necessary parameters. See section 6 (*Lookupprivilegevaluw function (winbase.h) - win32 apps 2022*) for “LookupPrivilegeValueW” Windows API call documentation.

```

162     lpSystemName = None
163     lpName = priv_list[x]
164
165     catch_error = kernel_handle.GetLastError()
166
167     lkPriv_resp = adv_handle.LookupPrivilegeValueW(lpSystemName, lpName, ctypes.byref(lpLuid))

```

Figure 13 – LookupPrivilegeValueW API Call and Parameters

The LookupPrivilegeValueW API call takes 3 parameters “lpSystemName”, “lpName”, and “lpLuid”. The first parameter lpSystemName instructs the API call to identify a specific system name e.g., “Desktop452” to identify a specific privilege. As seen on line 162 this was set to “None”. By setting the value to None it instructs the API call when issued to search on the local machine for a privilege instead. The second parameter “lpName” specifies the name of the specific privilege in question for example “SEDebugPrivilege”. On line 163 lpName was set to a specific index (x) of a list named “priv\_list”. The content of priv\_list contained a majority of valid and invalid privilege names to obtain luids. Figure 14 shows the majority of privileges defined inside priv\_list.

```

145 |     priv_list = ["SEDebugPrivilege", "SECreateSymbolicLinkPrivilege", "SEImpersonatePrivilege",
146 |                 "InvalidPrivilegeTest", "SERestorePrivilege", "SEShutdownPrivilege", "Thisisinvalid",
147 |                 "SETimezonePrivilege", "SEBackupPrivilege", "SEChangeNotifyPrivilege", "SECreateGlobalPrivilege",
148 |                 "SECreatePagefilePrivilege", "SEDelegateSessionUserImpersonatePrivilege", "SESecurityPrivilege",
149 |                 "SETakeOwnershipPrivilege", "SEIncreaseWorkingSetPrivilege", "SESystemProfilePrivilege", "SERemoteShutdownPrivilege",
150 |                 "SESystemtimePrivilege", "SEInvalidPriv", "SEUndockPrivilege", "SESystemEnvironmentPrivilege", "SEProfileSingleProcessPrivilege",
151 |                 "SEManageVolumePrivilege", "SELoadDriverPrivilege"]

```

Figure 14 – Priv\_list defined privileges

The third parameter “lpLuid” was specifically important. The reason for this is the following, whenever a privilege lookup is performed on a token it is done based on the luid and not on the privilege name itself. Each luid is unique, and the structure of a luid consists of a “lowpart” and a “highpart” both of which have preset constant values based on the specific privilege. The luid structure for the script can be viewed in Appendix A. See section 6 (*luid (igpupvdev.h)* 2020) for luid structure documentation.

After the API call error checking was used to identify if the LUID retrieval was successful, as per the Windows documentation a return value of non-zero indicates a successful API call. On line 174 in Figure 15 below a second condition was set up to identify whether a luid (privilege) was valid or invalid. If a luid is invalid both its highpart and its lowpart will be equal to 0. This condition checking was demonstrated within the script by feeding it false privilege names contained within the “priv\_list” list. The last error checking refers to a general API call fail and makes use of interfacing with the “Kernel32.dll” to utilise the “GetLastError()” function. This displays a system error code that can be troubleshot.

Figure 15 – Privilege Value Lookup API Error Checking

```

170 |         if lkPriv_resp != 0:
171 |             print("[+] LUID Successfully Retrieved")
172 |             sleep(0.25)
173 |
174 |         elif lpLuid.HighPart == 0 and lpLuid.LowPart == 0:
175 |             print("[+] LUID Successfully Retrieved \n[-] Privilege: " + lpName + " is not a valid privilege associated with this process \n")
176 |             rejected_privs.append(lpName)
177 |             continue
178 |
179 |         elif response == 0:
180 |             print("[+] Could Not Obtain LUID. Error code: {}".format(catch_error))

```

Following the successful API call to retrieve the LUID for a specific privilege the next step was to check whether the privilege was enabled or disabled. This was conducted by using the “PrivilegeCheck” API call. See section 6 (*Privilegecheck function (securitybaseapi.h) - win32 apps* 2019) for “PrivilegeCheck” Windows API call documentation. The API call takes in 3 parameters, the 1st being a “ClientToken” which is a handle to an access token of the specified process. The 2nd parameter “RequiredPrivileges” is a pointer to a “PRIVILEGE\_SET” structure. See section 6 (*Privilege\_set (winnt.h) - win32 apps* 2020) for the “PRIVILEGE\_SET” structure. The privilege set structure contains the “Privilege Count” i.e., how many privileges there are, “Privilege Control” which grants access to modify any privileges detected within the access token, and lastly, a “Privilege” which is an array of a “LUID\_AND\_ATTRIBUTES” structure containing a LUID value and attributes of the LUID (flags). Note that the privilege set structure and luid and attributes structure can be viewed in Appendix A. Lastly, the 3rd parameter “pfResult” is a pointer to a boolean value

that determines whether a specific privilege in an access token is enabled or disabled. Figure 16 below details the API call parameters prior to the call itself.

```
135 clientToken = TokenHandle
136 requiredPrivileges = PRIVILEGE_SET()
137 requiredPrivileges.PrivilegeCount = 1
138 requiredPrivileges.Privilege = LUID_AND_ATTRIBUTES()
139 requiredPrivileges.Privilege.Luid = LUID()
140 lpLuid = requiredPrivileges.Privilege.Luid
141 pfResult = ctypes.c_long()
```

Figure 16 – PrivilegeCheck API Call Parameter Population

As discussed in the report above (page 13) figure 11 and onwards, the “TokenHandle” parameter of the “OpenProcessToken” API call returned a handle to the specific processes access token. Therefore, this handle was used to satisfy the requirements of the 1st parameter (“clientToken”) of the “PrivilegeCheck” API call (line 135 of Figure 17). As seen on line 136 in figure 17 and as required by the Windows documentation the 2nd parameter “RequiredPrivileges” is set to a “PRIVILEGE\_SET”. The “PrivilegeCount” was subsequently set to 1 meaning that only 1 privilege at a time is to be expected. As defined in the Windows documentation the “Privilege” member of the structure was set to an array of “LUID\_AND\_ATTRIBUTES”. Before the privilege check API call is issued the “Privilege” member was set to the “LUID” structure. This allows indicating which set of privileges to check and also to perform error checking by targeting the “lowpart” and “highpart” of a specific luid. Figure 17 exhibits the “PrivilegeCheck” API call with all parameters previously discussed.

```
182 privCk_resp = adv_handle.PrivilegeCheck(clientToken, ctypes.byref(requiredPrivileges), ctypes.byref(pfResult))
183
184 if privCk_resp != 0:
185     print("[*] Running Privilege Checks On Process...")
186
187     sleep(1.50)
188 else:
189     print("[-] Could Not Run Privilege Check on Process. Error code: {0}".format(catch_error))
```

Figure 17 – PrivilegeCheck API Call

Figure 18 exhibits the API call with the response being captured within “privCk\_resp”. An if statement was subsequently implemented to perform necessary error checking. Defined within the API call documentation any value not equal to 0 indicates success. If the return was 0 then an appropriate system error was displayed by the kernel 32 function “GetLastError()”. See script reference [10] for the “PrivilegeCheck” Windows API call documentation. Figure 18 below shows the script being executed within the command prompt and displays the LUID being retrieved for the privilege and the associated privilege checks being conducted.

```
[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 40176
[+] Handle To Process Access Token Successfully Opened

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
```

Figure 18 – Script Execution LUID Retrieval and Priv Check

Upon successful identification of the token privilege status if and elif conditions were implemented to obtain and alter the token privileges based on the users input. Figure 19 below details the if and elif statements.

```
191 SE_PRIVILEGE_ENABLED = 2
192
193 if priv_choice == 1 and pfResult:
194     print("[!] Privilege: {0}".format(lpName) + " is Enabled")
195     print("[!] Destroying Enabled Privilege [!]\n")
196     requiredPrivileges.Privilege.Attributes = 0
197     sleep(1)
198
199 elif priv_choice == 1 and not pfResult:
200     print("[!] Privilege: {0}".format(lpName) + " Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...")
201     print("[*] Confirmed [*] \n")
202     requiredPrivileges.Privilege.Attributes = 0
203     sleep(0.35)
204
205 elif priv_choice == 2 and not pfResult:
206     print("[!] Privilege: {0}".format(lpName) + " is Destroyed")
207     print("[!] Restoring Destroyed Privilege [!]\n")
208     requiredPrivileges.Privilege.Attributes = SE_PRIVILEGE_ENABLED
209
210 elif priv_choice == 2 and pfResult:
211     print("[!] Privilege: {0}".format(lpName) + " Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...")
212     print("[*] Confirmed [*] \n")
213     requiredPrivileges.Privilege.Attributes = SE_PRIVILEGE_ENABLED
214
215 else:
216     print("Invalid choice, exiting...")
217     sleep(0.30)
218     exit(1)
219
```

Figure 19 – If and elif statements, altering current privilege

As seen on line 191 of Figure 20 a variable named “SE\_PRIVILEGE\_ENABLED” has a value of 2. As per the Microsoft documentation, the value of 2 indicates an enabled privilege. Line 193 provides an if condition containing the variable “priv\_choice”. As previously discussed “priv\_choice” relates to whether the user wants to enable or disable privileges for a particular process. The 2nd condition in the statement is if “pfResult” is true. The reason for this is that pfResult is a pointer to a boolean that returns true if the specified privilege is enabled, otherwise it returns false. The pointer is populated with the true or false from the previous “PrivilegeCheck” API call detailed on page 15. If the choice is 1 which relates to destroying privileges and the pointer is true we know that the current privilege is enabled therefore it is disabled on line 197 by assigning the associated luid attribute to 0. Note that only “SE\_PRIVILEGE\_ENABLED” is mentioned in the Microsoft documentation and not disabled therefore disabling a privilege is done directly by assigning the value of 0 to the specific luid attribute. As shown in Figure 20 above on line 200 if the choice is 2 which restores privileges and pfResult is false then it is known that the current privilege is disabled therefore the specific luid attribute can be enabled as done on line 209 by utilising the “SE\_PRIVILEGE\_ENABLED” variable. See section 6 (*NativeMethods Microsoft Framework* 2021) for SE\_PRIVILEGE\_ENABLED and disabled values. The remaining elif statements display separate text if the choice is to destroy and the privilege is already disabled, likewise with enabled.

The “AdjustTokenPrivileges” Windows API call was used to solidify the enabled and disabled token privileges by updating the processes access token. See section 6 (*AdjustTokenPrivileges function (securitybaseapi.h) - win32 apps* 2021) for the “AdjustTokenPrivileges” Windows API call documentation. This function takes 6 parameters in total, parameter 1, “TokenHandle” refers to the processes access token previously obtained. The 2nd parameter “DisableAllPrivileges” is a boolean that relates to whether the API call disables all of the access tokens privileges. The “NewState” parameter is a pointer to a “TOKEN\_PRIVILEGES” structure, this structure specifies privileges and their associated attributes for an access token and allows for relevant privileges to be accessed. See Appendix A for “TOKEN\_PRIVILEGES”

structure. The 4th parameter “BufferLength” specifies the size of data within the populated “NewState” parameter, it is important to note that the buffer must be big enough to receive all privileges. “PreviousState” is a pointer containing the “TOKEN\_PRIVILEGES” structure, containing the previous state of all privileges in the access token. The final parameter “ReturnLength” is a pointer to a variable that contains the buffer of the “PreviousState” parameter. Figure 20 below details the “AdjustTokenPrivileges” API call parameters.

```
223 TokenHandle = clientToken
224 DisableAllPrivileges = False
225 NewState = TOKEN_PRIVILEGES()
226 BufferLength = ctypes.sizeof(NewState)
227 PreviousState = ctypes.c_void_p()
228 ReturnLength = ctypes.c_void_p()
```

Figure 20 – AdjustTokenPrivileges API Call Parameters

As per the Windows documentation, the 1st parameter needed a handle to a processes access token, therefore, the variable “clientToken” which was most recently used on the “PrivilegeCheck” API call was assigned to “TokenHandle” (line 223). The 2nd parameter “DisableAllPrivileges” was not specifically needed for the API call and was disabled by setting the boolean value to false. As required by the Windows documentation the “NewState” parameter was assigned to the token privileges structure which allows it to access an array of privileges and their associated attributes. To populate the BufferLength parameter “ctypes.sizeof” was used to obtain the size in bytes of the populated NewState parameter as seen on line 226 above. The remaining 2 parameters PreviousState and ReturnLength were both not needed for this API call and were both set to void pointers.

Before the API call was issued the current token privileges contained within the parameter NewState (line 225, figure 21) needed to be updated with the privilege attributes previously set. Figure 22 below details the process of updating “NewState”.

```
230 NewState.Privilege = requiredPrivileges.Privilege
231 NewState.PrivilegeCount = 1
```

Figure 21 – NewState Parameter Update

Line 230 in figure 22 above accesses the current token privilege structure for the specific privilege in question and updates it with the new privilege value just previously set in figure 20. The last thing to be updated is to instruct the NewState structure how many privileges are expected, in this case it was 1 (the current privilege being updated).

After all relevant parameters were successfully populated the API call “AdjustTokenPrivileges” could be executed. Figure 22 below details the API call execution.

```
233 adjTkn_resp = adv_handle.AdjustTokenPrivileges(TokenHandle, DisableAllPrivileges, ctypes.byref(NewState),
234 BufferLength, ctypes.byref(PreviousState), ctypes.byref(ReturnLength))
```

Figure 22 – AdjustTokenPrivileges API Call

### 3.4.1 Function Iteration

It is important to note that the entire process outlined above was repeated within a for loop for each privilege contained within “priv\_list” (page 14, figure 14). Figure 23 below shows all 3 lists and the for-loop condition.

```
145 priv_list = ["SEDebugPrivilege", "SECreateSymbolicLinkPrivilege", "SEImpersonatePrivilege",  
146 "InvalidPrivilegeTest", "SERestorePrivilege", "SEShutdownPrivilege", "Thisisinvalid",  
147 "SETimeZonePrivilege", "SEBackupPrivilege", "SEChangeNotifyPrivilege", "SECreateGlobalPrivilege",  
148 "SECreatePagefilePrivilege", "SEDelegateSessionUserImpersonatePrivilege", "SESecurityPrivilege",  
149 "SETakeOwnershipPrivilege", "SEIncreaseWorkingSetPrivilege", "SESystemProfilePrivilege", "SERemoteShutdownPrivilege",  
150 "SESystemtimePrivilege", "SEInvalidPriv", "SEUndockPrivilege", "SESystemEnvironmentPrivilege", "SEProfileSingleProcessPrivilege",  
151 "SEManageVolumePrivilege", "SELoadDriverPrivilege"]  
152  
153 valid_privs = []  
154  
155 rejected_privs = []  
156  
157 for x in range(len(priv_list)):
```

Figure 23 – All lists, for loop condition

All 3 lists above utilise the for loop. “priv\_list” contains the privileges to be manipulated, “valid\_privs” only receives valid privileges into the list once they have been verified with luid checking, and “rejected\_privs” only receives invalid privileges into the list once they have been verified to be invalid with luid checking. Line 157 instructs the for loop to continue iterating until the length of the priv\_list has been reached.

As previously discussed in this report a privilege is needed for associated API calls to be performed upon it, for example, luid retrieval. In Figure 24 below on line 160 the value of the current index of the for loop which is a privilege is assigned to the variable “lpName”. lpName is then used by multiple API function calls to perform necessary tasks.

```
157 for x in range(len(priv_list)):  
158  
159     lpSystemName = None  
160     lpName = priv_list[x]
```

Figure 24 – lpName priv\_list index assignment

Figure 25 below exhibits the code which is responsible for checking to see if a privilege is valid.

```
170 elif lpLuid.HighPart == 0 and lpLuid.LowPart == 0:  
171     print("[+] LUID Successfully Retrieved \n[-] Privilege")  
172     rejected_privs.append(lpName)  
173     del lpName  
174     continue
```

Figure 25 – Checking if privilege is valid

As previously discussed in this report the checking of if a privilege is valid is done based on the luids highpart and lowpart value. The code above instructs the “lpName” (current privilege) to be added to the “rejected\_privs” list. The variable is then subsequently deleted as it is currently not required for this iteration. As it is known that the privilege is not valid “Continue” is then issued to force the next iteration of the for loop which contains a new privilege to be checked. This is done instead of wasting resources by executing the rest of the code below on an invalid privilege.

If the privilege is valid the rest of the API calls below are executed to manipulate the token privilege. Before the next iteration of the for loop is reached naturally the valid privilege is added to the “valid\_privs”



list to be displayed and the variable deleted ready for the next iteration. Figure 26 below details appending a valid privilege to the appropriate list and deleting the variable.

```
233 valid_privs.append(lpName)
234 del lpName
```

Figure 26 – Checking if privilege is valid

After the for loop has iterated over the entire length of the priv\_list an if statement prints out the relevant list whether it be valid or invalid privileges based on the initial user choice. Figure 27 below exhibits the if conditions for 2 different options and their respective lists.

```
237 if priv_choice == 1:
238     print("[+] The following process-associated token privileges were either destroyed or are valid Windows privileges \n")
239     print(*valid_privs,sep='\n')
240
241     print("\n[+] The following token privileges were invalid \n")
242     print(*rejected_privs,sep='\n')
243
244
245 elif priv_choice == 2:
246     print("\n[+] The following process-associated token privileges were either restored or are valid Windows privileges \n")
247     print(*valid_privs,sep='\n')
248
249     print("\n[+] The following token privileges were invalid \n")
250     print(*rejected_privs,sep='\n')
```

Figure 27 – Printing out valid or invalid privileges

### 3.4.2 Function Testing

After the function and all of its associated Windows API calls were complete, several tests were conducted to ensure that the theorised results were identical to the results captured. The tool used to analyse the processes access token and the tokens relevant privileges was “Sysinternals – Process Explorer”. Section 1.3 details the tools used throughout development of the script.

Process for function testing:

- Retrieve base process access token privileges with Process Explorer
  - Execute function (either destroy or restore access token privileges)
    - Retrieve updated process access token privileges with Process Explorer
      - Analyse base and updated token privileges
- Perform handle error checking

#### 3.4.2.1 Captured Results

The initial theorised results upon successful implementation of all API calls was the destruction or restoration of all valid targeted process access token privileges.

##### 3.4.2.1.1 Base Access Token Privilege Retrieval

Figure 1 below depicts all base access token privileges for the process “Task Manager”.



Privilege	Flags	Privilege	Flags
SeBackupPrivilege	Disabled	SeIncreaseBasePriorityPrivilege	Disabled
SeChangeNotifyPrivilege	Default Enabled	SeIncreaseQuotaPrivilege	Disabled
SeCreateGlobalPrivilege	Default Enabled	SeIncreaseWorkingSetPrivilege	Disabled
SeCreatePagefilePrivilege	Disabled	SeLoadDriverPrivilege	Disabled
SeCreateSymbolicLinkPrivilege	Disabled	SeManageVolumePrivilege	Disabled
SeDebugPrivilege	Enabled	SeProfileSingleProcessPrivilege	Disabled
SeDelegateSessionUserImpersonatePrivilege	Disabled	SeRemoteShutdownPrivilege	Disabled
SeImpersonatePrivilege	Default Enabled	SeRestorePrivilege	Disabled

*Figure 1 – Task Manager All Process Access Token Privileges*

#### 3.4.2.1.2 Function Execution

Figures 1 and 2 below depicts the privilege manipulation function being executed. The function is targeting the process “Task Manager” with the intent of restoring/enabling all possible access token privileges. See Appendix B for all function screenshots.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -p "Task Manager"
Press 1 to destroy process token privileges or 2 to restore process token privileges: 2

[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 10276
[+] Handle To Process Access Token Succesfully Opened

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEDebugPrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreateSymbolicLinkPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEImpersonatePrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[-] Privilege: InvalidPrivilegeTest is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SERestorePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]
```

*Figure 1 – Privilege Manipulation Function Execution*



```
[+] The following process-associated token privileges were either restored or are valid Windows privileges
SEDebugPrivilege
SECreateSymbolicLinkPrivilege
SEImpersonatePrivilege
SERestorePrivilege
SEShutdownPrivilege
SETimeZonePrivilege
SEBackupPrivilege
SEChangeNotifyPrivilege
SECreateGlobalPrivilege
SECreatePagefilePrivilege
SEDelegateSessionUserImpersonatePrivilege
SESecurityPrivilege
SETakeOwnershipPrivilege
SEIncreaseWorkingSetPrivilege
SESystemProfilePrivilege
SERemoteShutdownPrivilege
SESystemtimePrivilege
SEUndockPrivilege
SEProfileSingleProcessPrivilege
SEManageVolumePrivilege
SELoadDriverPrivilege

[+] The following token privileges were invalid
InvalidPrivilegeTest
Thisisinvalid
SEInvalidPriv
SESystemEnvironmentPrivilege

[+] All Token Privilege Modifications Successful [+]
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

Figure 2 – Privilege Manipulation Function Execution

### 3.4.2.1.3 Updated Token Privileges Retrieval

Figures 1,2 and 3 below depicts all access token privileges after successful function execution against the specified process “Task Manager”.

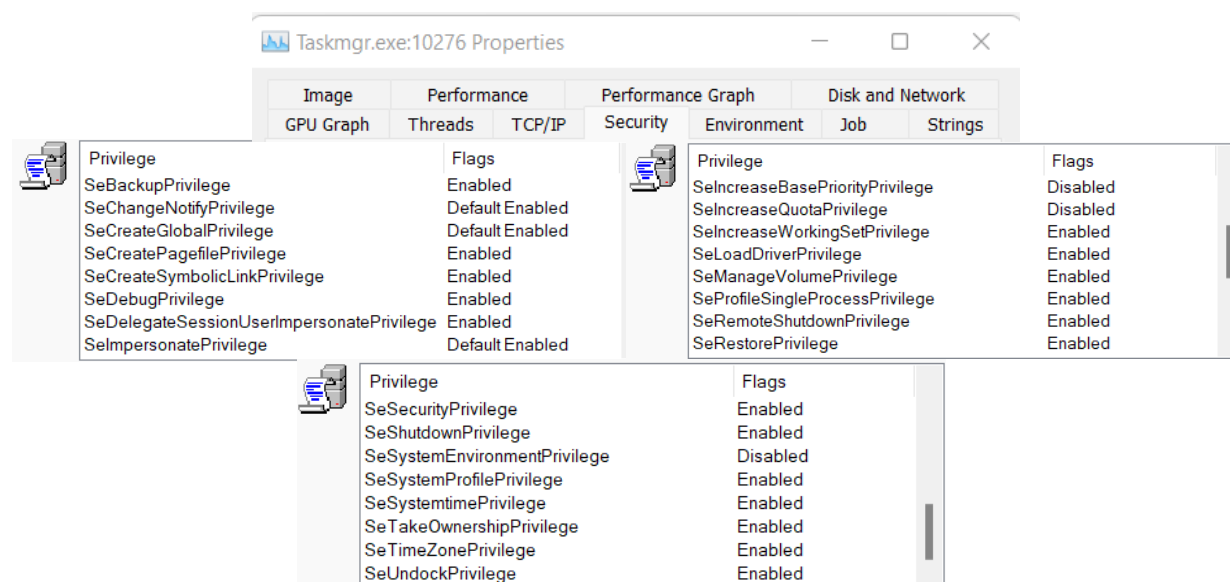


Figure 1 – Task Manager All Process Access Token Privileges

#### 3.4.2.1.4 Base and Updated Token Privilege Analysis

Analysis of the pre-function execution and post-function execution shows the targeted token access privileges being updated to an enabled state. Figure 1 below exhibits a visual representation of token access token privileges before and after the token manipulation function was successfully executed. Red indicates a destroyed/disabled privilege and green indicates a restored/enabled privilege. See Appendix C for the destruction of token privileges.



Figure 1 – Task Manager All Process Access Token Privileges

#### 3.4.2.1.5 Error Checking

In order to test for handle error checking the script was executed with its target being a process that did not currently exist. The expected output for this function would be error text with a system error code generated by the kernel function "GetLastError()". Figure 1 below depicts how the script handled a process that was not running.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -p "Task Manager"
Press 1 to destroy process token privileges or 2 to restore process token privileges: 2
[-] Could Not Obtain Handle. Error code: 6
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

*Figure 1 – Script Handle Error Checking*

As seen in figure 1 above by supplying a program which is not currently running no process for the program can be fetched and a handle can't be obtained. The associated system error code 6 refers to an invalid handle as detailed by the official Windows documentation.

### 3.5 CODE WALKTHROUGH (FUNCTION: TERMINATING PROCESSES)

Figure 1 details below exhibits the entire approach taken which includes multiple semantic API calls and DLL interfaces to successfully terminate Windows processes.

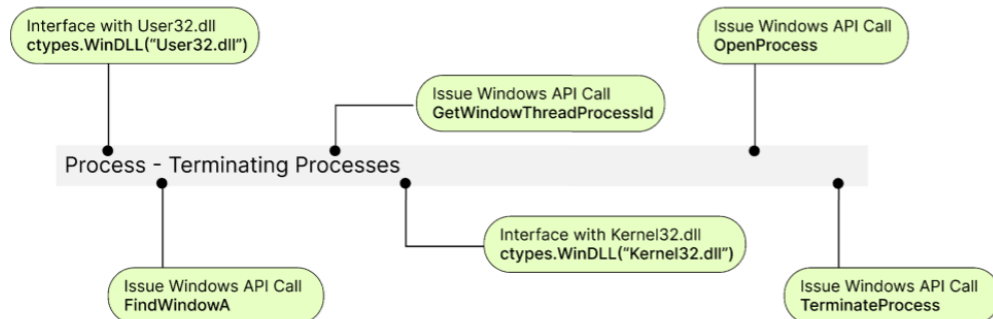


Figure 1 – Approach Taken – Process Termination

The structure of API calls associated with terminating Windows processes requires 4 API calls in total. It is important to note that the order in which API calls are executed aligns with the function previously discussed (manipulating token privileges). As the API calls of the previous function have been discussed only the remaining function “TerminateProcess” will be covered. See section 6 (*Terminateprocess function (processthreadsapi.h) - win32 apps 2021*) for TerminateProcess API call documentation. See section 3.2 and its API calls (FindWindowA, GetWindowThreadProcessId, OpenProcess).

The function associated with terminating Windows processes was defined as “killProc”. To automate and start the process of API calls, it took 1 argument which was a “ctypes.c\_char\_p” (a pointer to a string, passed in from argparse) and was also UTF-8 encoded. The reason for the encoding is due to the imminent Window API function call “FindWindowA” which must accept a “UTF-8” encoded parameter. Figure 2 details the function declaration with argument.

```
261 def killProc(lpWindowArg):
```

Figure 2 – Function killProc declaration

It is important to note that the function argument “lpWindowArg” is populated with the argparse argument ‘-k’ process name. Figure 3 shows the content of argument ‘k’ being assigned to the variable “kill”. Figure 4 exhibits logic checking, if the condition is met the content of “kill” is appropriately encoded as required by the Microsoft “FindWindowA” API call and is passed as an argument to the “killProc” function. See section 6 (*Findwindowa function (winuser.h) - win32 apps 2022*) for FindWindowA API call structure documentation.

```
422 kill = args.k
```

Figure 3 – Argument P being assigned to variable kill

```
437 elif args.k != None and args.s == None and args.p == None:
438     lpWindowName = ctypes.c_char_p(kill.encode('utf-8'))
439     killProc(lpWindowName)
```

Figure 4 – Logic checking and necessary encoding

After the following Windows API calls have been executed (FindWindowA, GetWindowThreadProcessId, OpenProcess) the API call “TerminateProcess” can be issued. It is important to note that the “Terminate Process” API call terminates both the specified process and all of its threads. The function takes 2 parameters, the 1st parameter is “hProcess” and is a handle to the specific process to be terminated. The handle to the process was achieved by issuing the API call “OpenProcess”. The 2nd parameter “uExitCode” is a value that is used by both the process and its associated threads that are terminated. Figure 5 below details the parameter initialization and the API call execution.

```

323     hProcess = opnProc_resp
324     uExitcode = ctypes.c_ulong(0)
325     term_proc = kernel_handle.TerminateProcess(opnProc_resp, uExitcode)
326
327     if term_proc !=0:
328         sleep(0.45)
329         print("[*] Process Successfully Killed")
330     else:
331         sleep(0.75)
332         print("[-] Could Not Terminate Process. Error code: {0}".format(error))

```

Figure 5 – TerminateProcess Parameters and Execution

Line 323 of Figure 5 shows “opnProc\_resp” being assigned to the parameter “hProcess”. In this instance, opnProc\_resp was the handle of the specific process, therefore, populating hProcess. The last parameter “uExitCode” was assigned with the value 0 as the exit code. However, it is important to note that the data type was “ctypes.c\_ulong” (an unsigned integer) as requested in the API call documentation. Line 325 exhibits the API call execution with the 2 populated parameters. The API call was conducted via a kernel32.dll handle. As seen on line 327 appropriate API call error checking was implemented to validate whether the process was successfully killed. If an error occurred as mentioned in [section 2.4](#) an appropriate error code would also be displayed.

### 3.5.1 Function Testing

After the function and all of its associated Windows API calls were complete, several tests were conducted to ensure that the theorised results were identical to the results captured. The tool used to analyse the processes access token and the token's relevant privileges was “Task Manager”. Section 1.3 details the tools used throughout the development of the script.

Process for function testing:


- Utilise Task Manager to correctly identify target programs process id
  - Execute function (terminate process)
- Perform handle error checking

#### 3.5.1.1 Captured Results

The initial theorised results upon successful implementation of all API calls was process termination for the specified program.

##### 3.5.1.1.1 Process ID Identification

Figure 1 below depicts the process id of “36020” which is related to cmd.exe. This information was obtained by utilising task manager.

Name	PID	Status	Username	CPU	Memory (ac...	Architec...	Description
 cmd.exe	36020	Running	CallumsPC	00	696 K	x64	Windows Command Processor

*Figure 1 – Cmd.exe Process ID*

#### 3.5.1.1.2 Function Execution

Figure 1 below exhibits the terminate process function being executed. Further inspection reveals that the handle to the process was successful, the associated process id was obtained, and the process was successfully killed.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -k "Command Prompt"
[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 36020
[*] Process Successfully Killed
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

*Figure 1 – killProc function execution*

#### 3.5.1.1.3 Error Checking

In order to test for handle error checking the script was executed with the target being a program that was currently not running. The expected output for this function would be error text with a system error code generated by the kernel function "GetLastError()". Figure 1 below depicts how the script handled a process that was not running.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -k "Command Prompt"
[-] Could Not Obtain Handle. Error code: 0
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

*Figure 1 – Script Handle Error Checking*

As seen in figure 1 above by supplying a program which is currently not running no process for the program can be fetched and a handle can't be obtained. The associated system error code refers to an invalid handle as detailed by the official Windows documentation.

## 3.6 CODE WALKTHROUGH (FUNCTION: SPAWNING PROCESSES)

Figure 1 details below exhibits the entire approach taken which includes 1 API call and 1 DLL interface to successfully spawn Windows processes.

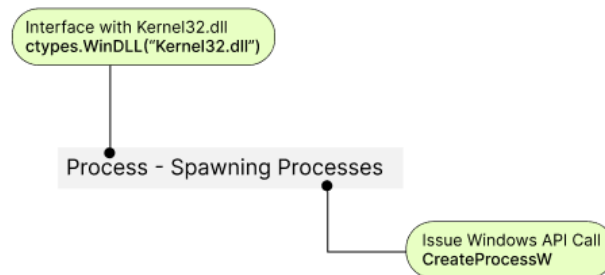


Figure 1 – Approach Taken – Spawning Processes

The function associated with spawning process was defined as “spawnProc”. To automate and start the process of the API call, 1 argument of which was a string (str) was passed as an argument to the function. This allowed for the 1st parameter of the imminent Window API function call “CreateProcessW” to be populated. The “CreateProcessW” API call works by creating a new process and its associated primary thread. See section 6 (*CREATEPROCESSW function (processthreadsapi.h) - win32 apps 2022*) for the “CreateProcessW” API documentation. Figure 2 details the function declaration with argument.

```
339 def spawnProc(lpApplicationName):
```

Figure 2 – “spawnProc” Function Declaration

It is important to note that the function argument “lpApplicationName” is populated with the argparse argument ‘-s’ process name. Figure 3 shows the content of argument ‘s’ being assigned to the variable “spawn”. Figure 4 exhibits logic checking, if the condition is met the content of “spawn” is cast to a string and then saved into the variable “lpApplicationName”. As seen on line 442 a function call to “spawnProc” is subsequently made with the application name passed as a parameter.

```
427 spawn = args.s
```

Figure 3 – Argument S being assigned to variable spawn

```
440 elif args.s != None and args.k == None and args.p == None:
441     lpApplicationName = str(spawn)
442     spawnProc(lpApplicationName)
```

Figure 4 – Logic checking and function call

The Windows API call “CreateProcessW” took 10 parameters in total with 3 parameters being separate structures as defined by the Windows documentation. The 1<sup>st</sup> parameter “lpApplicationName” already discussed is the name of the Windows module to be executed, its content can be a specific path to the module to run or a partial path. The 2<sup>nd</sup> parameter “lpCommandLine” relates to the command line text to be executed. “lpProcessAttributes” is a pointer to a “SECURITY\_ATTRIBUTES” structure which determines whether a child process can inherit a handle to the new process object by the API call. The 4<sup>th</sup> parameter “lpThreadAttributes” is a pointer to a “SECURITY\_ATTRIBUTES” structure which determines whether a child process can inherit a handle to the new thread object created by the API call. The “bInheritHandles”

parameter is a boolean which if true allows the calling process to be inherited by the new process spawned via the API call. As expected, if the value is false nothing is inherited. The 6<sup>th</sup> parameter “dwCreationFlags” controls how the new process is created. It is also important to note that the priority class is also controlled which primarily regards the priorities of the process’s threads. See section 6 (*Process creation flags (winbase.h)* - win32 apps 2022) for process creation flag documentation. “lpEnviroment” relates to the environment block for the newly created process. If the value is null the newly created process will utilise the environment of the process that called it. The 8<sup>th</sup> parameter of the API call “lpCurrentDirectory” refers to the process full path to the current directory. If the parameter has the value null the directory is defaulted to the same directory as the calling process. The 9<sup>th</sup> parameter ‘lpStartupInfo’ is a pointer to a “STARTUPINFO” structure. The “STARTUPINFO” structure is responsible for the standard handles and appearance of the main process window at the time of process creation. See section 6 (*Startupinfoa (processthreadsapi.h)* - win32 apps 2021) for “STARTUPINFO” documentation. See Appendix D for “STARTUPINFO” structure. Lastly, the “lpProcessInformation” parameter is a pointer to a “PROCESS\_INFORMATION” structure. The main function of this structure is to receive identification information about the newly created process and its associated primary thread. See section 5 figure [20] for the “PROCESS\_INFORMATION” structure Windows documentation. See Appendix D for the “PROCESS\_INFORMATION” structure. Figure 5 below shows all parameters being populated before the execution of the “CreateProcessW” API call.

```

382     kernel_handle = ctypes.WinDLL("Kernel32.dll")
383     userProgram = lpApplicationName
384     lpApplicationName = "C:\\Windows\\System32\\" + userProgram
385     lpCommandLine = None
386     lpProcessAttributes = None
387     lpThreadAttributes = None
388     lpEnviroment = None
389     lpCurrentDirectory = None
390     bInheritHandles = False
391     dwCreationFlags = 0x00000010
392     lpProcessInformation = PROCESS_INFORMATION()
393     lpStartupInfo = STARTUPINFOA()

```

Figure 5 – CreateProcessW parameters populated

As the “CreateProcessW” API call exists within the kernel32.dll a handle to the specific dll first had to be achieved as seen on line 382 in Figure 5 above. The 1st API call parameter “lpApplicationName” had a value that targeted the system32 folder. The “userProgram” variable containing the program which the user wants to run was concatenated with the “lpApplicationName” as seen on line 384. This provided an absolute path for the program to run, however, only for programs within the system32 folder. Note that the path structure would have to be altered to allow for a broader scope of programs to be run. The following 6 parameters were not required and were set to null or false (lpCommandLine, lpProcessAttributes, lpThreadAttributes, lpEnviroment, lpCurrentDirectory, bInheritHandles). The “dwCreationFlags” value was set specifically to “0x00000010” which allows for the process to be launched in a new console window. This prevents the new process from inheriting the caller's console (parent's console). As detailed within the Windows documentation both parameters “lpProcessInformation” and “lpStartupInfo” were both set to their respective structures with those being a “PROCESS\_INFORMATION” structure and a “STARTUPINFOA” structure.



After all parameters had been populated the “CreateProcessW” API call could be executed, and error checking performed. Figure 6 below shows the execution of the “CreateProcessW” API call with its associated parameters.

```

402 crProc_return = kernel_handle.CreateProcessW(lpApplicationName, lpCommandLine,
403 lpProcessAttributes, lpThreadAttributes, bInheritHandles, dwCreationFlags,
404 lpEnviroment, lpCurrentDirectory, ctypes.byref(lpStartupInfo), ctypes.byref(lpProcessInformation))
405
406 error = kernel_handle.GetLastError()
407 if crProc_return != 0:
408     sleep(0.45)
409     print("\n[+] Successfully Spawned: " + userProgram)
410     sleep(0.50)
411     print("[+] Creating New Window For: " + userProgram)
412     sleep(0.65)
413 else:
414     print("Error Could Not Spawn Program. Error code: {0}".format(error))

```

Figure 6 – CreateProcessW API Call & Error Checking

### 3.6.1 Function Testing

After the function and its associated Windows API call was complete, a simple test was conducted to ensure that the theorised results were identical to the results captured.

Process for function testing:

- Execute function and observe whether specified process pops up in a new window
- Perform error checking

#### 3.6.1.1 Captured Results

The initial theorised results upon successful implementation of all API calls was process creation for the specified program.

##### 3.6.1.1.1 Function Execution

Figure 1 below exhibits the process spawning function being executed.

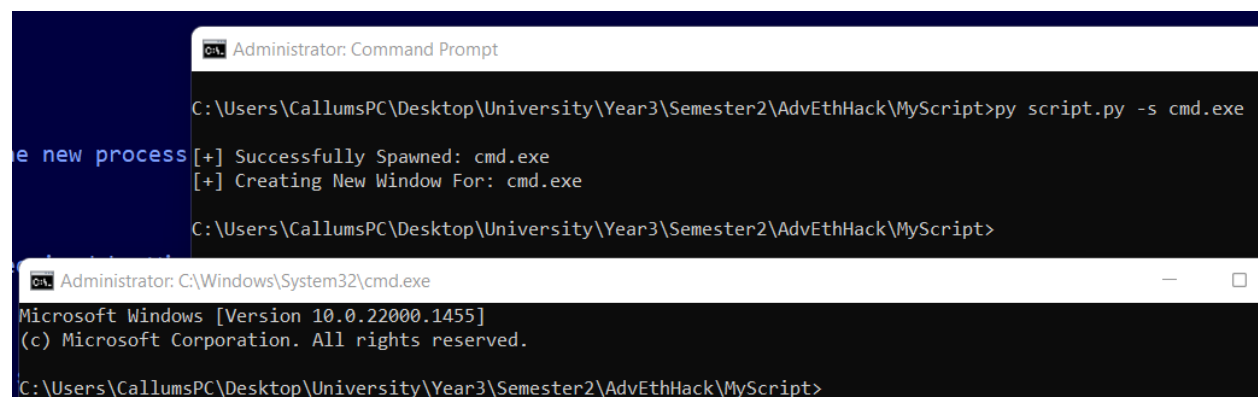


Figure 1 – spawnProc function execution

##### 3.6.1.1.2 Error Checking

In order to test for function error checking the script was executed with the target being a program that currently does not exist on the machine. The expected output for this function would be error text with a system error code generated by the kernel function “GetLastError()”. Figure 1 below depicts how the script handled a process that was not running.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -s nonExistent.exe
Error Could Not Spawn Program. Error code: 2

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

*Figure 1 – Script Handle Error Checking*

As seen in figure 1 above by supplying a program which currently does not exist the program can't be executed. The associated system error code refers to an invalid file as detailed by the official Windows documentation.

# 4 DISCUSSION

## 4.1 GENERAL DISCUSSION

---

The initial aims of this report were to inform, detail, evaluate and discuss the process of building the Windows API Manipulation script.

Utilising the extensive Windows API call documentation allowed for the development of the Windows API Manipulation Script to be conducted appropriately. More specifically, the documentation provided insight into the order in which API calls were executed, creating complex Windows API structures, and working with token privileges and access rights.

Within this section, all script functions are evaluated to provide clarity on whether they were successful, and if the initial theorised results were accurate.

## 4.2 SCRIPT EVALUATION

---

Each of the scripts functions was appropriately evaluated to identify whether each function was successful and performed optimally in performing its task.

### 4.2.1 Function: Collecting User Input Evaluation

The results produced and analysed in [section 3.3](#) indicate that the user-input function was successful in launching separate Windows API call functions associated with specific command-line arguments. It is also important to note that the success of the function was also measured based on the ability to handle incorrect and unexpected arguments. The function handled these unexpected and incorrect cases with ease and returned the expected output. A complete evaluation of the function concluded that the initial theorised results detailed within [section 3.3.1.1](#) were accurate and the function worked as intended. Figure 1 below exhibits function error handling.

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py
[-] Error: Script must take 1 argument
[-] Script Usage: -h for help

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -i
usage: script.py [-h] [-s S] [-k K] [-p P]
script.py: error: unrecognized arguments: -i

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -h
usage: script.py [-h] [-s S] [-k K] [-p P]

Use this script to spawn and kill Windows processes. This script may also be used to destroy and restore process privileges.

options:
  -h, --help  show this help message and exit

All arguments:
  -s S      The application to spawn | Usage: py script.py -s cmd.exe
  -k K      The current running application to kill | Usage: py script.py -k "Task Manager"
  -p P      Enable and Disable all Windows token privileges for the specified application | Usage: py script.py -p "Command Prompt"

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

Figure 1 – Function Error Handling Evaluation

#### 4.2.2 Function: Token Privilege Manipulation Evaluation

The results produced and analysed in [section 3.4](#) indicate that the token privilege manipulation function was successful in manipulating process access token privileges. Specifically, analysis of the pre-function execution and post-function execution revealed that the function was successful in manipulating every targeted privilege of the specified processes access token. The function handled each valid and invalid privilege appropriately and sequentially manipulated each related access token privilege based on the chosen option by the user. Furthermore, the success of the token privilege function also indicated that the subsequent structure of all Windows API calls to be executed as detailed within [section 3.4 figure 1](#) was correct. A complete evaluation of the function concluded that the initial theorised results detailed within [section 3.4.2.1](#) were accurate and the function worked as intended. Figure 1 below shows restored privileges for a specific process's access token.

```
[+] The following process-associated token privileges were either restored or are valid Windows privileges
SEDebugPrivilege
SECreateSymbolicLinkPrivilege
SEImpersonatePrivilege
SERestorePrivilege
SEShutdownPrivilege
SETimeZonePrivilege
SEBackupPrivilege
SEChangeNotifyPrivilege
SECreateGlobalPrivilege
SECreatePagefilePrivilege
SEDelegateSessionUserImpersonatePrivilege
SESecurityPrivilege
SETakeOwnershipPrivilege
SEIncreaseWorkingSetPrivilege
SESystemProfilePrivilege
SERemoteShutdownPrivilege
SESystemtimePrivilege
SEUndockPrivilege
SEProfileSingleProcessPrivilege
SEManageVolumePrivilege
SELoadDriverPrivilege

[+] The following token privileges were invalid
InvalidPrivilegeTest
Thisisinvalid
SEInvalidPriv
SESystemEnvironmentPrivilege

[+] All Token Privilege Modifications Successful [+]
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

Figure 1 – Privilege Manipulation Evaluation

#### 4.2.3 Function: Killing Processes Evaluation

The results produced and analysed in [section 3.5](#) indicate that the function associated with killing processes was successful. Analysis of the target process id generated by the script yielded identical results when compared to using “Task Manager”. This solidified the function's ability to correctly establish handles to the specified process and to obtain its process id before successfully terminating it. The efficiency, speed, and overall success of the function was observed and primarily linked to the appropriate structure of Windows API calls detailed within [section 3.5 Figure 1](#). A complete evaluation of the function concluded that the initial theorised results detailed within [section 3.5.1.1](#) were accurate and the function worked as intended. Figure 1 below exhibits the script for obtaining the correct process id and successfully terminating the process.

```

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -k "Command Prompt"

[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 36020
[*] Process Successfully Killed

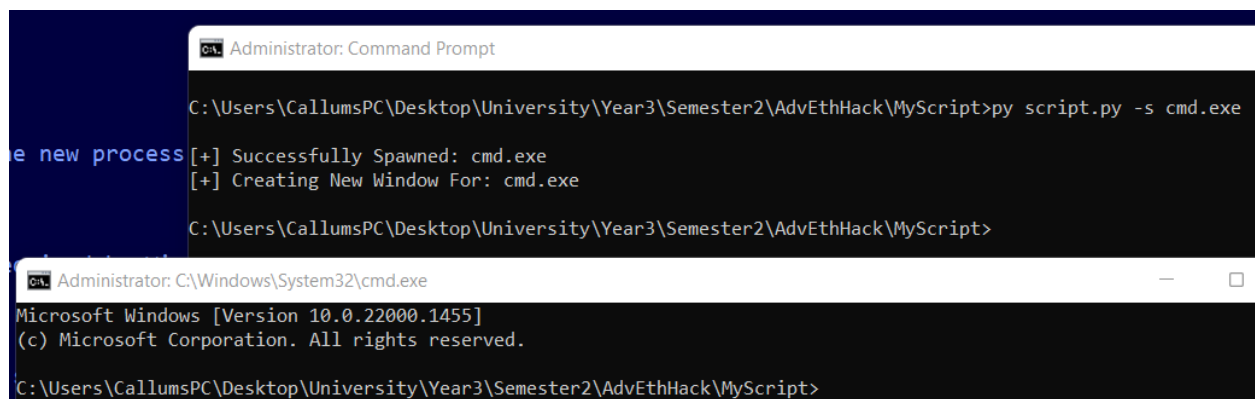
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>

```

Figure 1 – Killing Processes Evaluation

#### 4.2.4 Function: Spawning Processes Evaluation

The results produced and analysed in [section 3.6](#) indicate that the function associated with spawning Windows processes was successful. Visually observing new processes being spawned after the function execution showed that the function was consistent and effective in spawning new Windows processes. This ultimately revealed that the function dealt with specified arguments appropriately by spawning the correct process that the user had originally entered. A complete evaluation of the function concluded that the initial theorised results detailed within [section 3.6.1.1](#) were accurate and the function worked as intended. Figure 1 below details the original specified program to be run being successfully spawned in a new window.



```

Administrator: Command Prompt

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -s cmd.exe

[+] Successfully Spawned: cmd.exe
[+] Creating New Window For: cmd.exe

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>

```

```

Administrator: C:\Windows\System32\cmd.exe

Microsoft Windows [Version 10.0.22000.1455]
(c) Microsoft Corporation. All rights reserved.

C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>

```

Figure 1 – Spawning Processes Evaluation

## 5 FUTURE WORK

Succeeding the current report, future work would include a second report that would prove beneficial in communicating vulnerabilities facilitated by the Windows API Manipulation script. The paper would aim to increase awareness of threats that the vulnerabilities pose by discussing in detail how they interoperate and also by detailing a step-by-step process of how exploitation is successfully achieved. Lastly, relevant vulnerability countermeasures would be covered to increase cyber resilience for the audience of the report. Alongside the report new script functions would be added to the current script to specifically exploit any pre-existing vulnerabilities or vulnerabilities newly created by the script.

Note that the vulnerabilities detailed below are possible due to the scripts ability manipulate a processes privilege. Each vulnerability listed will be fully covered within the new report:

- Windows API Hashing – Makes malware analysis much harder by purposely hiding dubious imported Windows APIs from the IAT (Import Address Table) of an executable.
- Memory Dumping via SEDebugPrivilege – Allows a process to be debugged including writing and reading to the specific processes memory. The tool “ProcDump” can be utilised to dump the processes memory e.g., LSASS process (Local Security Authority Subsystem Service) which is responsible for storing the credentials of a user after the initial login to the system. The dump can be loaded into a memory extractor such as “Mimikatz” to obtain passwords for user accounts.
- Impersonation via SEImpersonatePrivilege – Allows a targeted remote processes token object to be duplicated, enabling a new process to be spawned with the duplicated token.. The main objective related to this vulnerability is to achieve a SYSTEM shell especially when the current user is “Administrator”. Achieving a SYSTEM shell can be obtained by targeting a process that is running as NT AUTHORITY/SYSTEM and duplicating its token enabling the new spawned process user to be NT AUTHORITY/SYSTEM.
- File read access via SeBackupPrivilege – Allows the system to authorise read access to any file on the system. Use cases of this vulnerability include reading password hashes of “Administrator” accounts residing in the Windows registry and utilising hash crackers to obtain the password.

## 6 REFERENCES

- Windows Vulnerability Report 2022 (2021) Healthcare IT News*. Available at: <https://www.healthcareitnews.com/news/report-windows-had-most-security-vulnerabilities-any-microsoft-product-last-year> (Accessed: March 13, 2023).
- KathleenDollard (no date) *Microsoft: Why the Windows API, Walkthrough: Calling Windows APIs - Visual Basic / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/com-interop/walkthrough-calling-windows-apis> (Accessed: March 12, 2023).
- GrantMeStrength (no date) *Windows API index, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list> (Accessed: March 12, 2023).
- Ghostlulz (2019) *Windows API Keylogger, Medium*. Medium. Available at: <https://medium.com/@ghostlulzhacks/malware-basic-windows-key-logger-746cc09e66aa> (Accessed: March 12, 2023).
- Ghosh, S. (2020) *Token manipulation attacks, Checkmate*. Available at: <https://niiconsulting.com/checkmate/2019/11/token-manipulation-attacks-part-2-process-of-impersonation/> (Accessed: March 13, 2023).
- staff, E.T.editorial (2022) *Windows Malware Stats , RSS*. Available at: <https://eandt.theiet.org/content/articles/2022/12/over-95-per-cent-of-2022-s-new-malware-threats-aimed-at-windows> (Accessed: March 13, 2023).
- Ctypes - a foreign function library for Python* (no date) *Python documentation*. Available at: <https://docs.python.org/3/library/ctypes.html> (Accessed: March 23, 2023).
- RE - regular expression operations* (no date) *Python documentation*. Available at: <https://docs.python.org/3/library/re.html> (Accessed: March 23, 2023).
- Pankaj (2022) *Python time sleep function, DigitalOcean*. DigitalOcean. Available at: <https://www.digitalocean.com/community/tutorials/python-time-sleep> (Accessed: March 23, 2023).
- Karl-Bridge-Microsoft (no date) *System error codes (0-499) (winerror.h) - win32 apps, (WinError.h) - Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-> (Accessed: April 6, 2023).
- Karl-Bridge-Microsoft (no date) *GetLastError function (errhandlingapi.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror> (Accessed: April 6, 2023).

Jwmsft (no date) *Findwindowa function (winuser.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-findwindowa> (Accessed: April 6, 2023).

Jwmsft (no date) *GetWindowThreadProcessId function (winuser.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getwindowthreadprocessid> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *OpenProcess function (processthreadsapi.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *Process security and access rights - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *OpenProcessToken function (processthreadsapi.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocesstoken> (Accessed: April 6, 2023).

GrantMeStrength (no date) *Token\_privileges (winnt.h) - win32 apps, TOKEN\_PRIVILEGES (winnt.h) - Win32 apps / Microsoft Learn*. Available at: [https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-token\\_privileges](https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-token_privileges) (Accessed: April 6, 2023).

GrantMeStrength (no date) *Lookupprivilegevaluw function (winbase.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-lookupprivilegevaluw> (Accessed: April 6, 2023).

Lorihollasch (no date) *luid (igpupvdev.h) , \_LUID (igpupvdev.h) - Windows drivers / Microsoft Learn*. Available at: [https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/igpupvdev/ns-igpupvdev-\\_luid](https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/igpupvdev/ns-igpupvdev-_luid) (Accessed: April 6, 2023).

Alvinashcraft (no date) *Privilegecheck function (securitybaseapi.h) - win32 apps, Win32 apps / Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-privilegecheck> (Accessed: April 6, 2023).

GrantMeStrength (no date) *Privilege\_set (winnt.h) - win32 apps, PRIVILEGE\_SET (winnt.h) - Win32 apps / Microsoft Learn*. Available at: [https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-privilege\\_set](https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-privilege_set) (Accessed: April 6, 2023).



*NativeMethods Microsoft FrameWork* (no date) *Reference Source*. Available at: <https://referencesource.microsoft.com/#system/compmod/microsoft/win32/NativeMethods.cs,98ce25e56b86729e> (Accessed: April 6, 2023).

Alvinashcraft (no date) *Adjusttokenprivileges function (securitybaseapi.h) - win32 apps, Win32 apps* / *Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-adjusttokenprivileges> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *Terminateprocess function (processthreadsapi.h) - win32 apps, Win32 apps* / *Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *CREATEPROCESSW function (processthreadsapi.h) - win32 apps, Win32 apps* / *Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *Process creation flags (winbase.h) - win32 apps, (WinBase.h) - Win32 apps* / *Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags> (Accessed: April 6, 2023).

Karl-Bridge-Microsoft (no date) *Process creation flags (winbase.h) - win32 apps, (WinBase.h) - Win32 apps* / *Microsoft Learn*. Available at: <https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags> (Accessed: April 6, 2023).

## 7 APPENDICES

### APPENDIX A – TOKEN PRIVILEGE CHANGE STRUCTURES

---

```
class LUID(ctypes.Structure):
    _fields_ = [
        ("LowPart", ctypes.c_ulong),
        ("HighPart", ctypes.c_ulong),
    ]

class LUID_AND_ATTRIBUTES(ctypes.Structure):
    _fields_ = [
        ("Luid", LUID),
        ("Attributes", ctypes.c_ulong),
    ]

class TOKEN_PRIVILEGES(ctypes.Structure):
    _fields_ = [
        ("PrivilegeCount", ctypes.c_ulong),
        ("Privilege", LUID_AND_ATTRIBUTES),
    ]

class PRIVILEGE_SET(ctypes.Structure):
    _fields_ = [
        ("PrivilegeCount", ctypes.c_ulong),
        ("Control", ctypes.c_ulong),
        ("Privilege", LUID_AND_ATTRIBUTES),
    ]

STANDARD_RIGHTS_REQUIRED = 0x000F0000
STANDARD_RIGHTS_READ = 0x00020000
TOKEN_ASSIGN_PRIMARY = 0x0001
TOKEN_DUPLICATE = 0x0002
TOKEN_IMPERSONATION = 0x0004
TOKEN_QUERY = 0x0008
TOKEN_QUERY_SOURCE = 0x0010
TOKEN_ADJUST_PRIVILEGES = 0x0020
TOKEN_ADJUST_GROUPS = 0x0040
TOKEN_ADJUST_DEFAULT = 0x0080
TOKEN_ADJUST_SESSIONID = 0x0100
TOKEN_READ = (STANDARD_RIGHTS_READ | TOKEN_QUERY)
TOKEN_ALL_ACCESS = (STANDARD_RIGHTS_REQUIRED |
    TOKEN_ASSIGN_PRIMARY |
    TOKEN_DUPLICATE |
    TOKEN_IMPERSONATION |
    TOKEN_QUERY |
    TOKEN_QUERY_SOURCE |
    TOKEN_ADJUST_PRIVILEGES |
    TOKEN_ADJUST_GROUPS |
    TOKEN_ADJUST_DEFAULT |
    TOKEN_ADJUST_SESSIONID)
```

## APPENDIX B – RESTORING TOKEN PRIVILEGES

---

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -p "Task Manager"
Press 1 to destroy process token privileges or 2 to restore process token privileges: 2

[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 10276
[+] Handle To Process Access Token Successfully Opened

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEDebugPrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreateSymbolicLinkPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEImpersonatePrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[-] Privilege: InvalidPrivilegeTest is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SERestorePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEShutdownPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[-] Privilege: Thisisinvalid is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SETimeZonePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEBackupPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEChangeNotifyPrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreateGlobalPrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
```

```

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreatePagefilePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEDelegateSessionUserImpersonatePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESecurityPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SETakeOwnershipPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEIncreaseWorkingSetPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESystemProfilePrivilege Is Valid, However Appears To Be Already Enabled Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SERemoteShutdownPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESystemTimePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[-] Privilege: SEInvalidPriv is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEUndockPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[-] Privilege: SESystemEnvironmentPrivilege is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEProfileSingleProcessPrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEManageVolumePrivilege is Destroyed
[!] Restoring Destroyed Privilege [!]

[+] LUID Successfully Retrieved

```

```
[+] The following process-associated token privileges were either restored or are valid Windows privileges
```

```
SEDebugPrivilege  
SECreateSymbolicLinkPrivilege  
SEImpersonatePrivilege  
SERestorePrivilege  
SEShutdownPrivilege  
SETimeZonePrivilege  
SEBackupPrivilege  
SEChangeNotifyPrivilege  
SECreateGlobalPrivilege  
SECreatePagefilePrivilege  
SEDelegateSessionUserImpersonatePrivilege  
SESecurityPrivilege  
SETakeOwnershipPrivilege  
SEIncreaseWorkingSetPrivilege  
SESystemProfilePrivilege  
SERemoteShutdownPrivilege  
SESystemtimePrivilege  
SEUndockPrivilege  
SEProfileSingleProcessPrivilege  
SEManageVolumePrivilege  
SELoadDriverPrivilege
```

```
[+] The following token privileges were invalid
```

```
InvalidPrivilegeTest  
Thisisinvalid  
SEInvalidPriv  
SESystemEnviromentPrivilege
```

```
[+] All Token Privilege Modifications Successful [+]
```

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>
```

## APPENDIX C – DESTROYING TOKEN PRIVILEGES

---

```
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>py script.py -p "Task Manager"
Press 1 to destroy process token privileges or 2 to restore process token privileges: 1

[+] Successfully Obtained Handle To Process
[+] Successfully Obtained Process ID
[+] Successfully Opened Process ID: 51088
[+] Handle To Process Access Token Successfully Opened

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEDebugPrivilege is Enabled
[!] Destroying Enabled Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreateSymbolicLinkPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEImpersonatePrivilege is Enabled
[!] Destroying Enabled Privilege [!]

[+] LUID Successfully Retrieved
[-] Privilege: InvalidPrivilegeTest is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SERestorePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEShutdownPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[-] Privilege: Thisisinvalid is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SETimeZonePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEBackupPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEChangeNotifyPrivilege is Enabled
[!] Destroying Enabled Privilege [!]
```

```

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SECreatePagefilePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEDelegateSessionUserImpersonatePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESecurityPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SETakeOwnershipPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEIncreaseWorkingSetPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESystemProfilePrivilege is Enabled
[!] Destroying Enabled Privilege [!]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SERemoteShutdownPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SESystemtimePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[-] Privilege: SEInvalidPriv is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEUndockPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[-] Privilege: SESystemEnvironmentPrivilege is not a valid privilege associated with this process

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEProfileSingleProcessPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEManageVolumePrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

```

```

[+] LUID Successfully Retrieved
[*] Running Privilege Checks On Process...
[!] Privilege: SEloadDriverPrivilege Is Valid, However Appears To Be Already Destroyed Or Is Not Used By This Process, Confirming...
[*] Confirmed [*]

[+] The following process-associated token privileges were either destroyed or are valid Windows privileges


SEDebugPrivilege
SECreateSymbolicLinkPrivilege
SEImpersonatePrivilege
SERestorePrivilege
SEShutdownPrivilege
SETimeZonePrivilege
SEBackupPrivilege
SEChangeNotifyPrivilege
SECreateGlobalPrivilege
SECreatePagefilePrivilege
SEDelegateSessionUserImpersonatePrivilege
SESecurityPrivilege
SETakeOwnershipPrivilege
SEIncreaseWorkingSetPrivilege
SESystemProfilePrivilege
SERemoteShutdownPrivilege
SESystemtimePrivilege
SEUndockPrivilege
SEProfileSingleProcessPrivilege
SEManageVolumePrivilege
SELoadDriverPrivilege

[+] The following token privileges were invalid


InvalidPrivilegeTest
Thisisinvalid
SEInvalidPriv
SESystemEnvironmentPrivilege

[+] All Token Privilege Modifications Successful [+]
C:\Users\CallumsPC\Desktop\University\Year3\Semester2\AdvEthHack\MyScript>


```

 Privilege	Flags
SeBackupPrivilege	Disabled
SeChangeNotifyPrivilege	Default Enabled
SeCreateGlobalPrivilege	Default Enabled
SeCreatePagefilePrivilege	Disabled
SeCreateSymbolicLinkPrivilege	Disabled
SeDebugPrivilege	Disabled
SeDelegateSessionUserImpersonatePrivilege	Disabled
SeImpersonatePrivilege	Default Enabled

 Privilege	Flags
SeIncreaseBasePriorityPrivilege	Disabled
SeIncreaseQuotaPrivilege	Disabled
SeIncreaseWorkingSetPrivilege	Disabled
SeLoadDriverPrivilege	Disabled
SeManageVolumePrivilege	Disabled
SeProfileSingleProcessPrivilege	Disabled
SeRemoteShutdownPrivilege	Disabled
SeRestorePrivilege	Disabled

 Privilege	Flags
SeSecurityPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeSystemEnvironmentPrivilege	Disabled
SeSystemProfilePrivilege	Disabled
SeSystemtimePrivilege	Disabled
SeTakeOwnershipPrivilege	Disabled
SeTimeZonePrivilege	Disabled
SeUndockPrivilege	Disabled



## APPENDIX D – SPAWNING PROCESS STRUCTURES

---

```
424 class STARTUPINFOA(ctypes.Structure):
425     _fields_ = [
426         ("cb", ctypes.c_ulong),
427         ("lpReserved", ctypes.c_char_p),
428         ("lpDesktop", ctypes.c_char_p),
429         ("lpTitle", ctypes.c_char_p),
430         ("dwX", ctypes.c_ulong),
431         ("dwY", ctypes.c_ulong),
432         ("dwXSize", ctypes.c_ulong),
433         ("dwYSize", ctypes.c_ulong),
434         ("dwXCountChars", ctypes.c_ulong),
435         ("dwYCountChars", ctypes.c_ulong),
436         ("dwFillAttribute", ctypes.c_ulong),
437         ("dwFlags", ctypes.c_ulong),
438         ("wShowWindow", ctypes.c_ushort),
439         ("cbReserved2", ctypes.c_ushort),
440         ("lpReserved2", ctypes.POINTER(ctypes.c_byte)),
441         ("hStdInput", ctypes.c_void_p),
442         ("hStdOutput", ctypes.c_void_p),
443         ("hStdError", ctypes.c_void_p),
444     ]
445     #PROCESS_INFORMATION struct, see windows ref
446 class PROCESS_INFORMATION(ctypes.Structure):
447     _fields_ = [
448         ("hProcess", ctypes.c_void_p),
449         ("hThread", ctypes.c_void_p),
450         ("dwProcessId", ctypes.c_ulong),
451         ("dwThreadId", ctypes.c_ulong),
452     ]
```