Code Analysis

Reverse Engineering & Code Analysis

`Reverse engineering` is a process that takes us beneath the surface of executable files or compiled machine code, enabling us to decode their functionality, behavioral traits, and structure. With the absence of source code, we turn to the analysis of disassembled code instructions, also known as `assembly code analysis`. This deeper level of understanding helps us to uncover obscured or elusive functionalities that remain hidden even after static and dynamic analysis.

To untangle the complex web of machine code, we turn to a duo of powerful tools: `Disassemblers` and `Debuggers`.

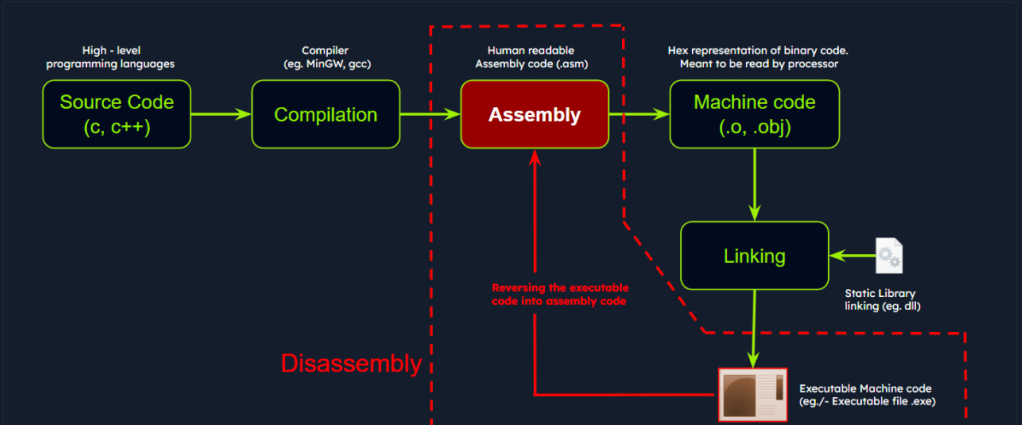
- A `Disassembler` is our tool of choice when we wish to conduct a static analysis of the code, meaning that we need not execute the code. This type of analysis is invaluable as it helps us to understand the structure and logic of the code without activating potentially harmful functionalities. Some prime examples of disassemblers include `IDA`, `Cutter`, and `Ghidra`.
- A `Debugger`, on the other hand, serves a dual purpose. Like a disassembler, it decodes machine code into assembly instructions. Additionally, it allows us to execute code in a controlled manner, proceeding instruction by instruction, skipping to specific locations, or halting the execution flow at designated points using breakpoints. Examples of debuggers include `x32dbg`, `x64dbg`, `IDA`, and `OllyDbg`.

Let's take a step back and understand the challenge before us. The journey of code from human-readable high-level languages, such as C or C++, to `machine code` is a one-way ticket, guided by the compiler. `Machine code`, a binary language that computers process directly, is a cryptic narrative for human analysts. Here's where the assembly language comes into play, acting as a bridge between us and the machine code, enabling us to decode the latter's story.

A disassembler transforms machine code back into assembly language, presenting us with a readable sequence of instructions. Understanding assembly and its mnemonics is pivotal in dissecting the functionality of malware.

`Code analysis` is the process of scrutinizing and deciphering the behavior and functionality of a compiled program or binary. This involves analyzing the instructions, control flow, and data structures within the code, ultimately shedding light on the purpose, functionality, and potential `indicators of compromise (IOCs)`.

Understanding a program or a piece of malware often requires us to reverse the compilation process. This is where `Disassembly` comes into the picture. By converting machine code back into assembly language instructions, we end up with a set of instructions that are symbolic and mnemonic, enabling us to decode the logic and workings of the program.



Resources

Go to Questions

Table of Contents

Introduction To Malware & Malware Analysis

Prerequisites

Windows Internals

Static Analysis

Static Analysis On Linux

Static Analysis On Windows

Dynamic Analysis

Dynamic Analysis

Code Analysis

Code Analysis

Debugging

Creating Detection Rules

Creating Detection Rules

Skills Assessment

Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left

Disassemblers are our allies in this process. These specialized tools take the binary code, generate the corresponding assembly instructions, and often supplement them with additional context such as memory addresses, function names, and control flow analysis. One such powerful tool is IDA, a widely used disassembler and debugger revered for its advanced analysis features. It supports multiple executable file formats and architectures, presenting a comprehensive disassembly view and potent analysis capabilities.

Let's now navigate to the bottom of this section and click on "Click here to spawn the target system!". Then, let's RDP into the Target IP using the provided credentials. The vast majority of the actions/commands covered from this point up to end of this section can be replicated inside the target, offering a more comprehensive grasp of the topics presented.

Code Analysis

```
MisaelMacias@htb[/htb]$ xfreerdp /u:htb-student /p:'HTB_@cademy_stdnt!' /v:[Target IP] /dynamic-res
```

Code Analysis Example: shell.exe

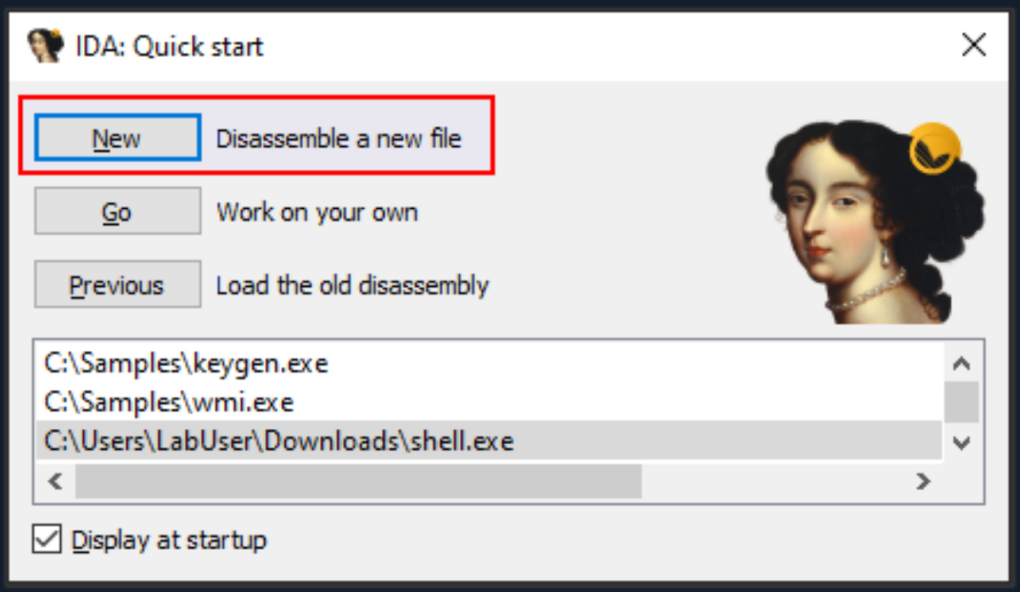
Let's persist with the analysis of the `shell.exe` malware sample residing in the `C:\Samples\MalwareAnalysis` directory of this section's target. Up until this point, we've discovered that it conducts `sandbox detection`, and that it includes a possible sleep mechanism - a `5-second ping` delay - before executing its intended operations.

Importing a Malware Sample into the Disassembler – IDA

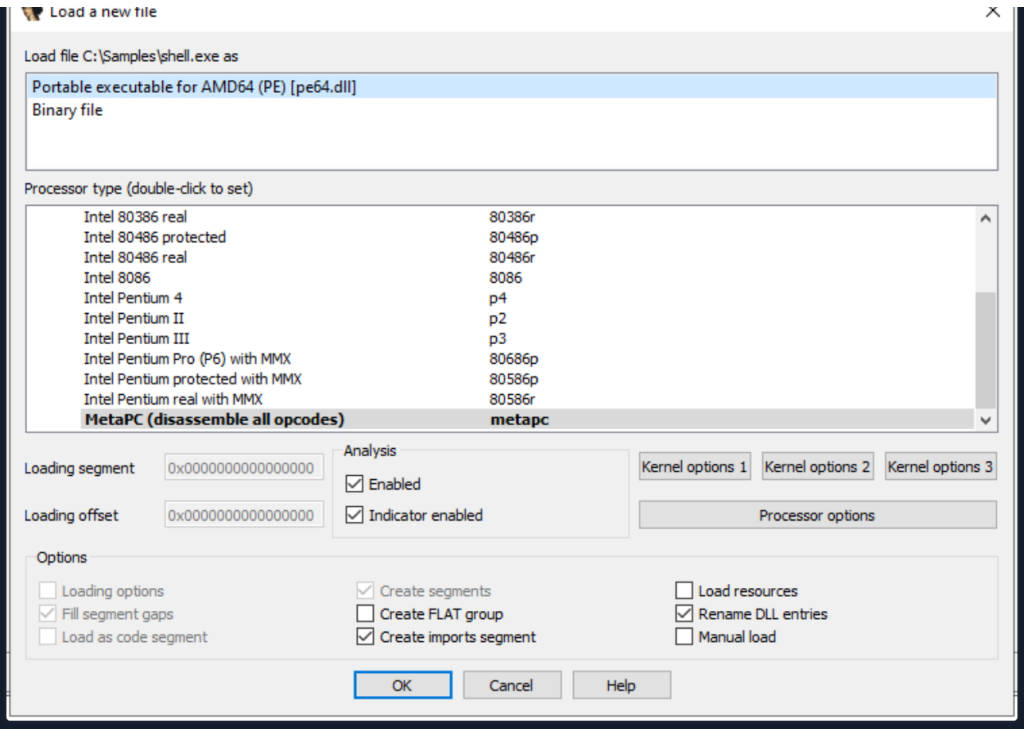
For the next stage in our investigation, we must scrutinize the code in `IDA` to ascertain its further actions and discover how to circumvent the sandbox check employed by the malware sample.

We can initiate `IDA` either by double-clicking the `IDA` shortcut that is placed on the Desktop or by right-clicking it and selecting `Run as administrator` to ensure proper access rights. At first, it will display the license information and subsequently prompt us to open a new executable for analysis.

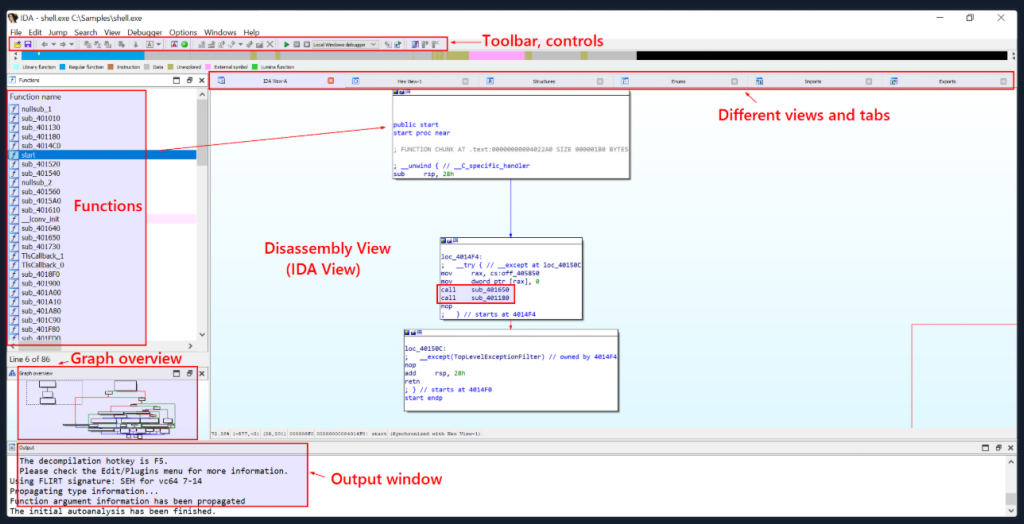
Next, opt for `New` and select the `shell.exe` sample residing in the `C:\Samples\MalwareAnalysis` directory of this section's target to dissect.



The `Load a new file` dialog box that pops up next is where we can select the processor architecture. Choose the correct one and click `OK`. By default, `IDA` determines the appropriate processor type.



After we hit `OK`, `IDA` will load the executable file into memory and disassemble the machine code to render the disassembled output for us. The screenshot below illustrates the different views in `IDA`.



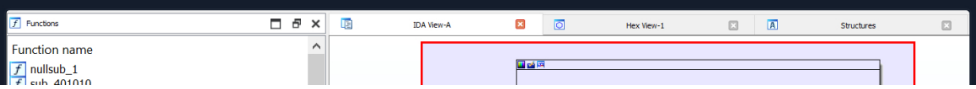
Once the executable is loaded and the analysis completes, the disassembled code of the sample `shell.exe` will be exhibited in the main `IDA-View` window. We can traverse through the code using the cursor keys or the scroll bar and zoom in or out using the mouse wheel or the zoom controls.

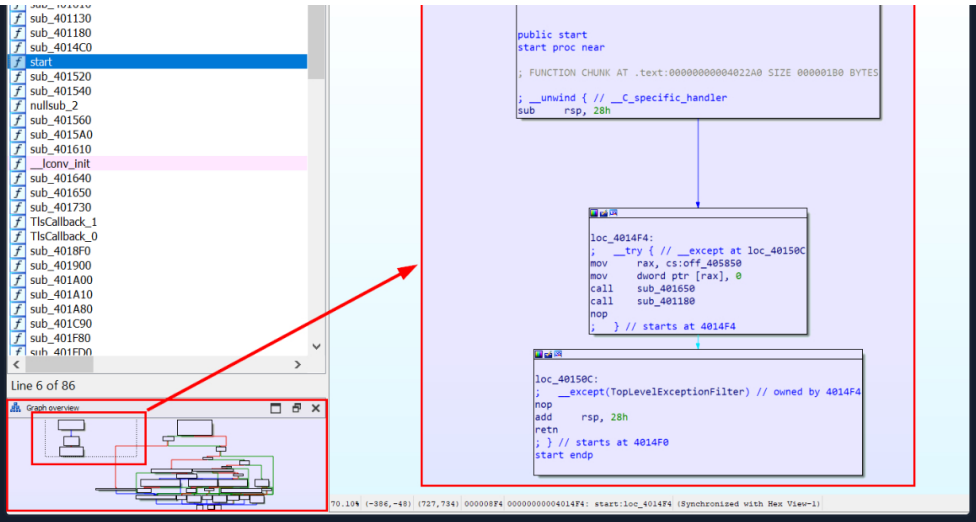
Text and Graph Views

The disassembled code is presented in two modes, namely the `Graph view` and the `Text view`. The default view is the `Graph view`, which provides a graphic illustration of the function's basic blocks and their interconnections. Basic blocks are instruction sequences with a single entry and exit point. These basic blocks are symbolized as nodes in the graph view, with the connections between them as edges.

To toggle between the graph and text views, simply press the `spacebar` button.

- The `Graph view` offers a pictorial representation of the program's control flow, facilitating a better understanding of execution flow, identification of loops, conditionals, and jumps, and a visualization of how the program branches or cycles through different code paths.



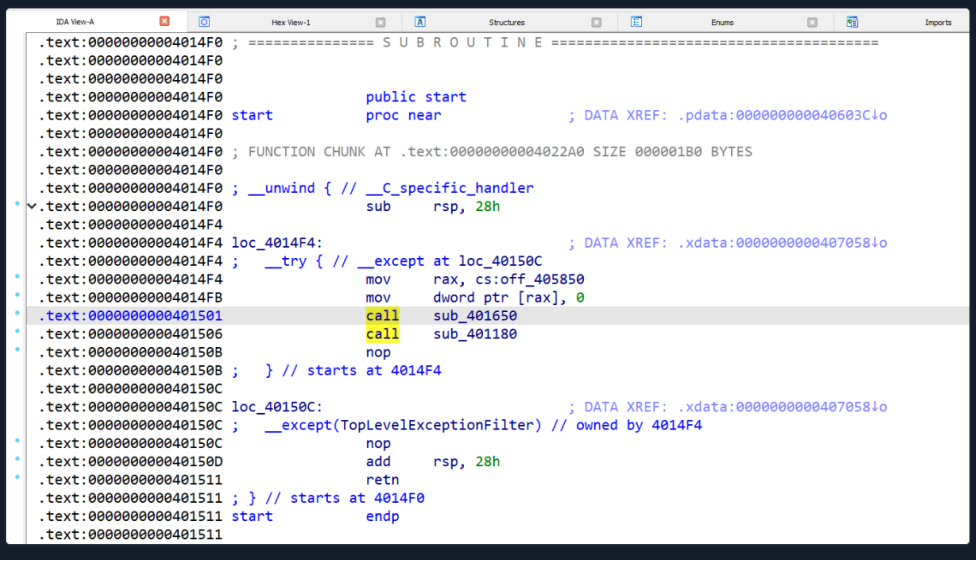


The functions are displayed as `nodes` in the `Graph view`. Each function is depicted as a distinct node with a unique identifier and additional details such as the `function name`, `address`, and `size`.

- The `Text view` displays the assembly instructions along with their corresponding memory addresses. Each line in the `Text view` represents an `instruction or a data element` in the code, beginning with the `section name:virtual address` format (for example, `.text:00000000004014F0`, where the section name is `.text` and the virtual address is `00000000004014F0`).

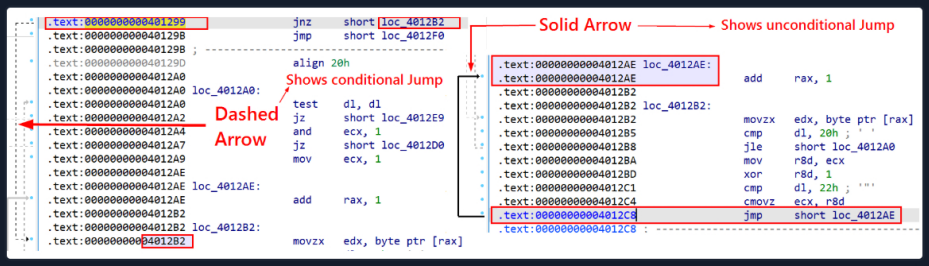
Code: ida

```
text:00000000004014F0 ; =============== S U B R O U T I N E =======================================
text:00000000004014F0
text:00000000004014F0
text:00000000004014F0                 public start
text:00000000004014F0 start           proc near               ; DATA XREF: .pdata:000000000040
text:00000000004014F0
text:00000000004014F0 ; FUNCTION CHUNK AT           .text:00000000004022A0 SIZE 000001B0 BYTES
text:00000000004014F0
text:00000000004014F0 ; __unwind { // __C_specific_handler
text:00000000004014F0                 sub     rsp, 28h
text:00000000004014F4
text:00000000004014F4 loc_4014F4:                             ; DATA XREF: .xdata:000000000040
text:00000000004014F4 ;   __try { // __except at loc_40150C
text:00000000004014F4                 mov     rax, cs:off_405850
text:00000000004014FB                 mov     dword ptr [rax], 0
text:0000000000401501                 call    sub_401650
text:0000000000401506                 call    sub_401180
text:000000000040150B                 nop
text:000000000040150B ;   } // starts at 4014F4
```



IDA's Text view employs arrows to signify different types of control flow instructions and jumps. Here are some commonly seen arrows and their interpretations:

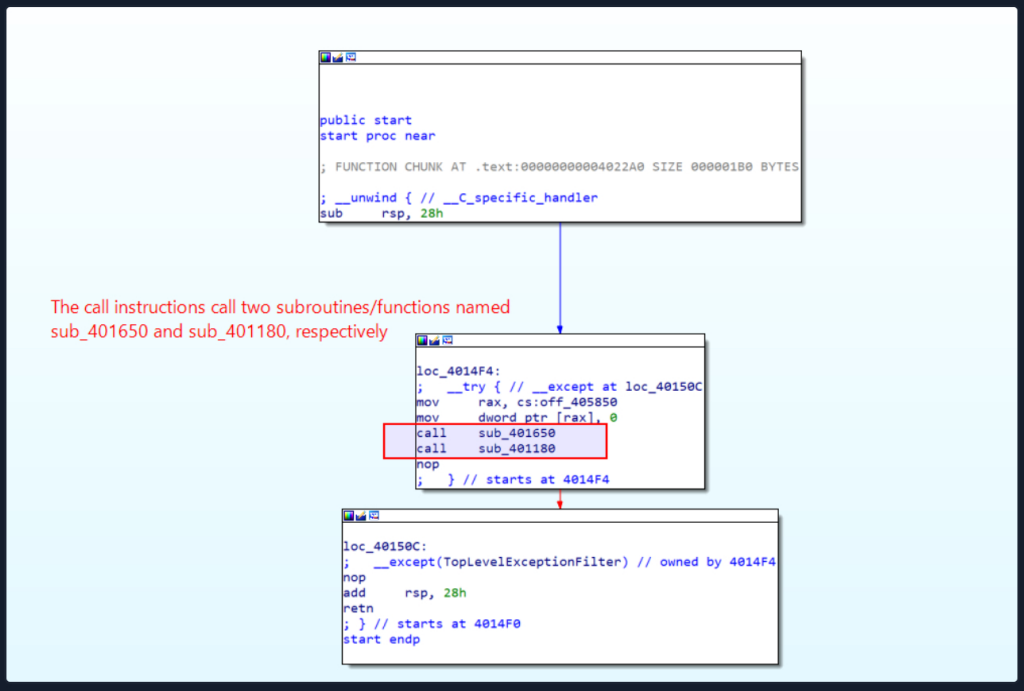
- Solid Arrow (→): A solid arrow denotes a direct jump or branch instruction, indicating an unconditional shift in the program's flow where execution moves from one location to another. This occurs when a jump or branch instruction like jmp or call is encountered.
- Dashed Arrow (---→): A dashed arrow represents a conditional jump or branch instruction, suggesting that the program's flow might change based on a specific condition. The destination of the jump depends on the condition's outcome. For instance, a jz (jump if zero) instruction will trigger a jump only if a previous comparison yielded a zero value.



By default, IDA initially exhibits the main function or the function at the program's designated entry point. However, we have the liberty to explore and examine other functions in the graph view.

Recognizing the Main Function in IDA

The following screenshot demonstrates the start function, which is the program's entry point and is generally responsible for setting up the runtime environment before invoking the actual main function. This is the initial start function shown by IDA after the executable is loaded.



Our objective is to locate the actual main function, which necessitates further exploration of the disassembly. We will search for function calls or jumps that lead to other functions, as one of them is likely to be the main function. IDA's graph view, cross-references, or function list can aid in navigating through the disassembly and identifying the main function.

However, to reach the main function, we first need to understand the function of this start function. This function primarily consists of some initialization code, exception handling, and function calls. It eventually jumps to the loc_40150C label, which is an exception handler. Therefore, we can infer that this is not the actual main function where

the program logic typically resides. We will inspect the other function calls to identify the `main` function.

The code commences by subtracting `0x28` (40 in decimal) from the `rsp` (stack pointer) register, effectively creating space on the stack for local variables and preserving the previous stack contents.

Code: ida

```
public start
start proc near

; FUNCTION CHUNK AT .text:00000000004022A0 SIZE 000001B0 BYTES

; __unwind { // __C_specific_handler
sub     rsp, 28h
```

The middle block in the screenshot above represents an exception handling mechanism that uses `structured exception handling (SEH)` in the code. The `__try` and `__except` keywords suggest the setup of an exception handling block. Within this, the subsequent `call` instructions call two subroutines (functions) named `sub_401650` and `sub_401180`, respectively. These are placeholder names automatically generated by IDA to denote subroutines, program locations, and data. The autogenerated names usually bear one of the following prefixes followed by their corresponding virtual addresses: `sub_<virtual_address>` or `loc_<virtual_address>` etc.

Code: ida

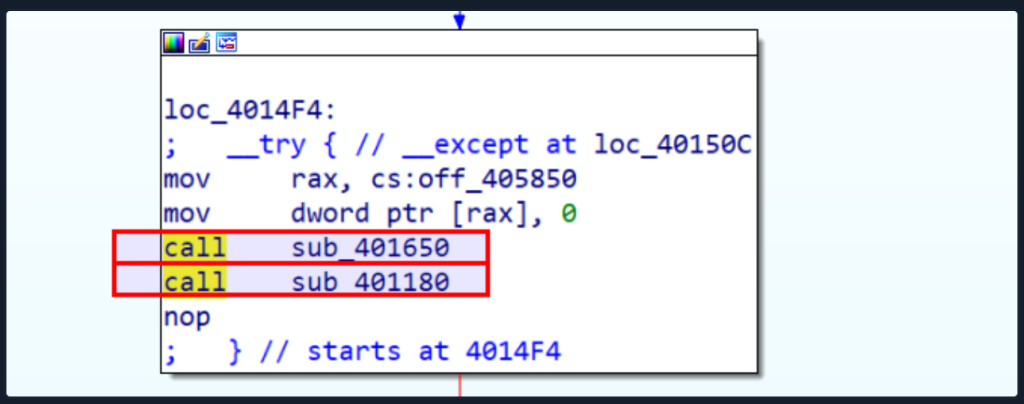
```
loc_4014F4:
;   __try { // __except at loc_40150C
mov     rax, cs:off_405850
mov     dword ptr [rax], 0
call    sub_401650          ; Will inspect this function
call    sub_401180          ; Will inspect this function
nop
;   } // starts at 4014F4

-----------------------------------------------

loc_40150C:
;   __except(TopLevelExceptionFilter) // owned by 4014F4
nop
add     rsp, 28h
retn
; } // starts at 4014F0
start endp
```

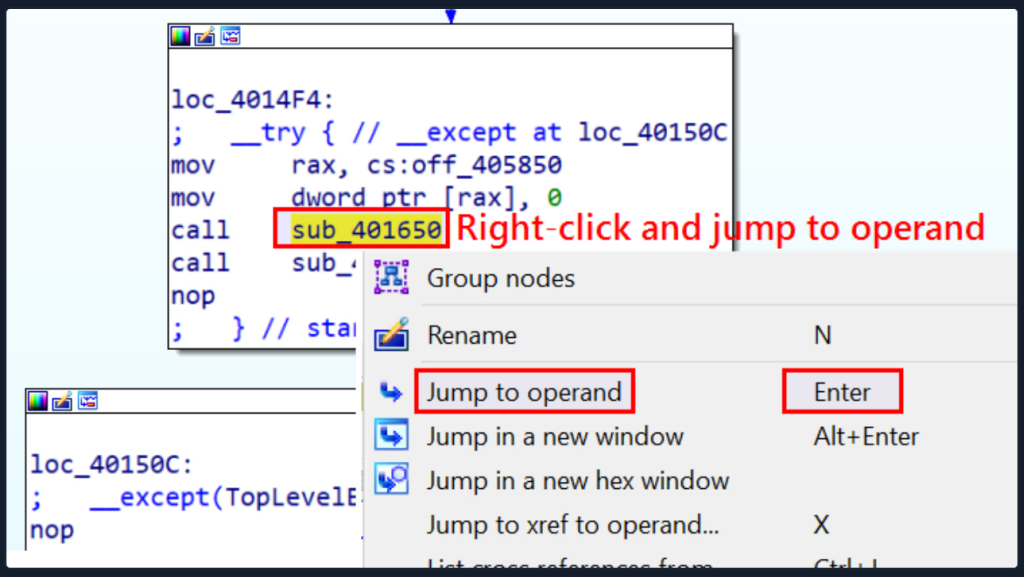
Navigating Through Functions in IDA

Let's inspect the contents of these two functions `sub_401650` and `sub_401180` by navigating within each function to peruse the disassembled code.



We will initially open the first function/subroutine `sub_401650`. To enter a function in `IDA`'s disassembly view, place the cursor on the instruction that represents the function call (or jump instruction) we want to follow, then right-click on the

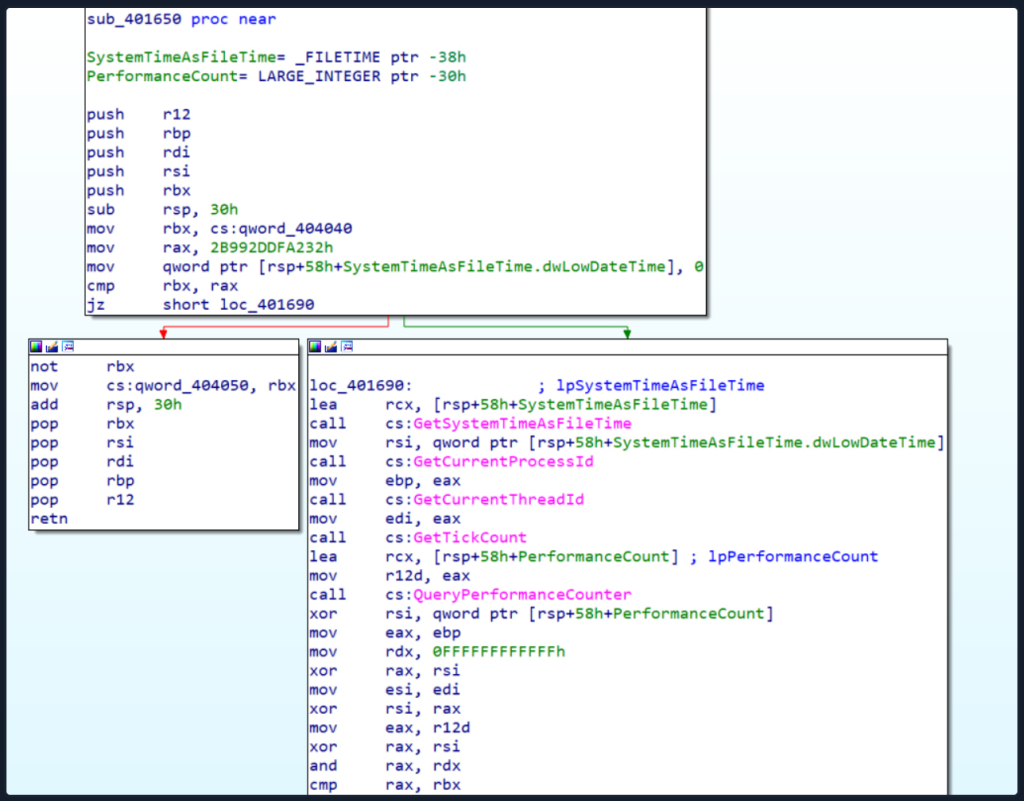
instruction and select `Jump to Operand` from the context menu. Alternatively, we can press the `Enter` key on our keyboard.



Then, `IDA` will guide us to the target location of the jump or function call, taking us to the start of the called function or the destination of the jump.

Now that we're inside the first function/subroutine `sub_401650`, let's strive to understand it in order to determine if it's the `main` function. If not, we'll navigate through other functions and discern the call to the `main` function.

In this subroutine `sub_401650`, we can see call instructions to the functions such as `GetSystemTimeAsFileTime`, `GetCurrentProcessId`, `GetCurrentThreadId`, `GetTickCount`, and `QueryPerformanceCounter`. This pattern is frequently observed at the beginning of disassembled executable code and typically consists of setting up the initial stack frame and carrying out some system-related initialization tasks.

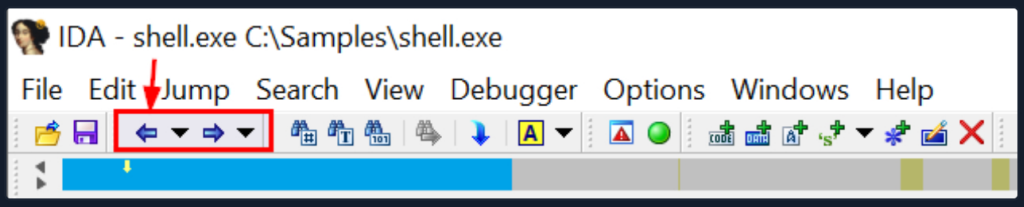


The type of instructions detailed here are typically found in the executable code produced by compilers targeting the x86/x64 architecture. When an executable is loaded and run by the operating system, it falls to the operating system to ready the execution environment for the program. This process involves tasks such as stack setup, register initialization, and preparation of system-relevant data structures.

Broadly speaking, this section of code is part of the initial execution environment setup, carrying out necessary system-related initialization tasks before the program's main logic executes. The goal here is to guarantee that the program launches in a consistent state, with access to necessary system resources and information. To clarify, this isn't where the program's main logic resides, and so we need to explore other function calls to pinpoint the `main` function.

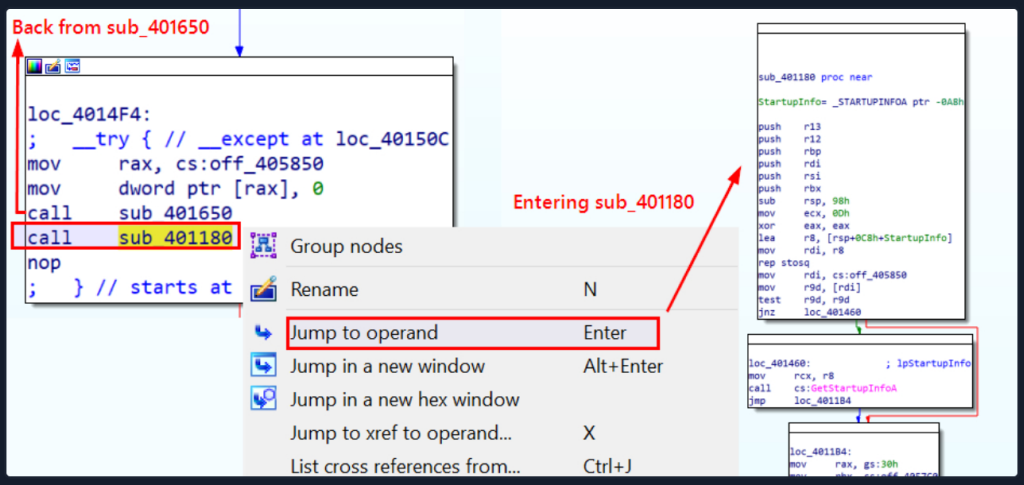
Let's return to and open the second subroutine, `sub_401180`, to examine its contents.

To backtrack to the previous function we were scrutinizing, we can press the `Esc` key on our keyboard, or alternatively, we can click the `Jump Back` button in the toolbar.

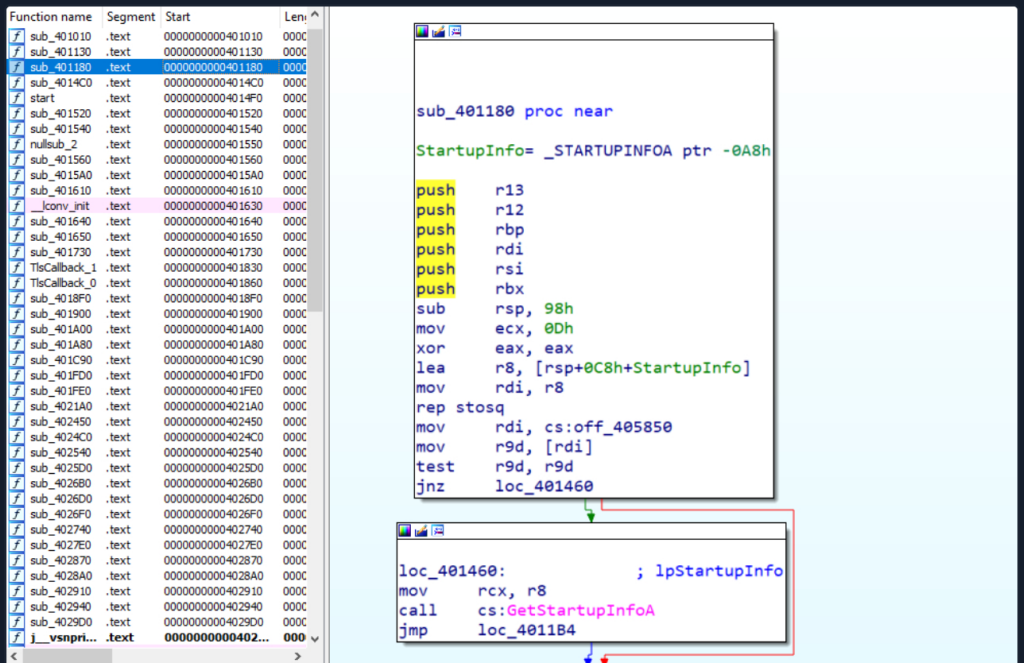


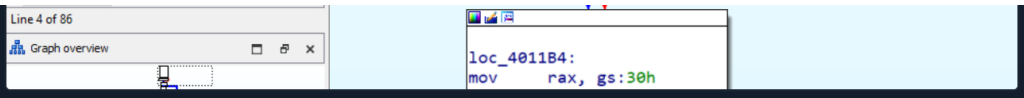
`IDA` will transport us back to the previous function we were inspecting (`loc_4014F4`), taking us to where we were prior to shifting to the current function or location. We're now back at the preceding location, which contains the call instructions to the current function, `sub_401650`, as well as another function, `sub_401180`.

From here, we can position the cursor on the instruction to call `sub_401180` and press `Enter`.



This will guide us into the function `sub_401180`, where we will endeavor to identify the `main` function in which the program logic is situated.

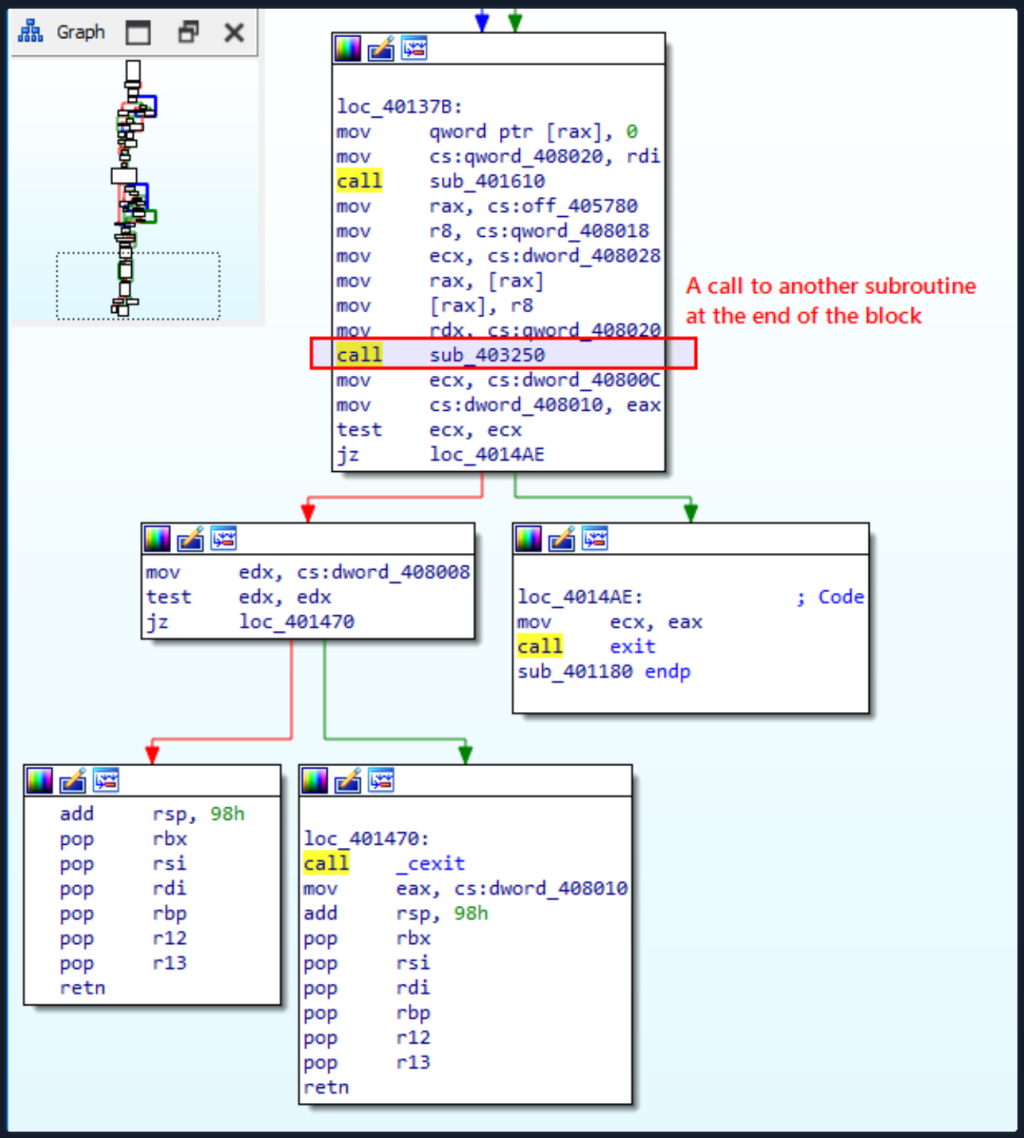




Upon examination, we can observe that this function seems to be implicated in initializing the `StartupInfo` structure and performing certain checks relative to its value. The `rep stosq` instruction nullifies a block of memory, while subsequent instructions modify the contents of registers and execute conditional jumps based on register values. This does not seem to be the `main` function in which the program logic resides, but it does contain a few `call` instructions which could potentially lead us to the `main` function. We will investigate all the `call` instructions prior to the return of this function.

We need to scroll to this function's endpoint and begin searching for `call` instructions from the bottommost one.

On scrolling upwards from the endpoint of this block (where the function returns), we observe a call to another subroutine, `sub_403250`, prior to this function's return.



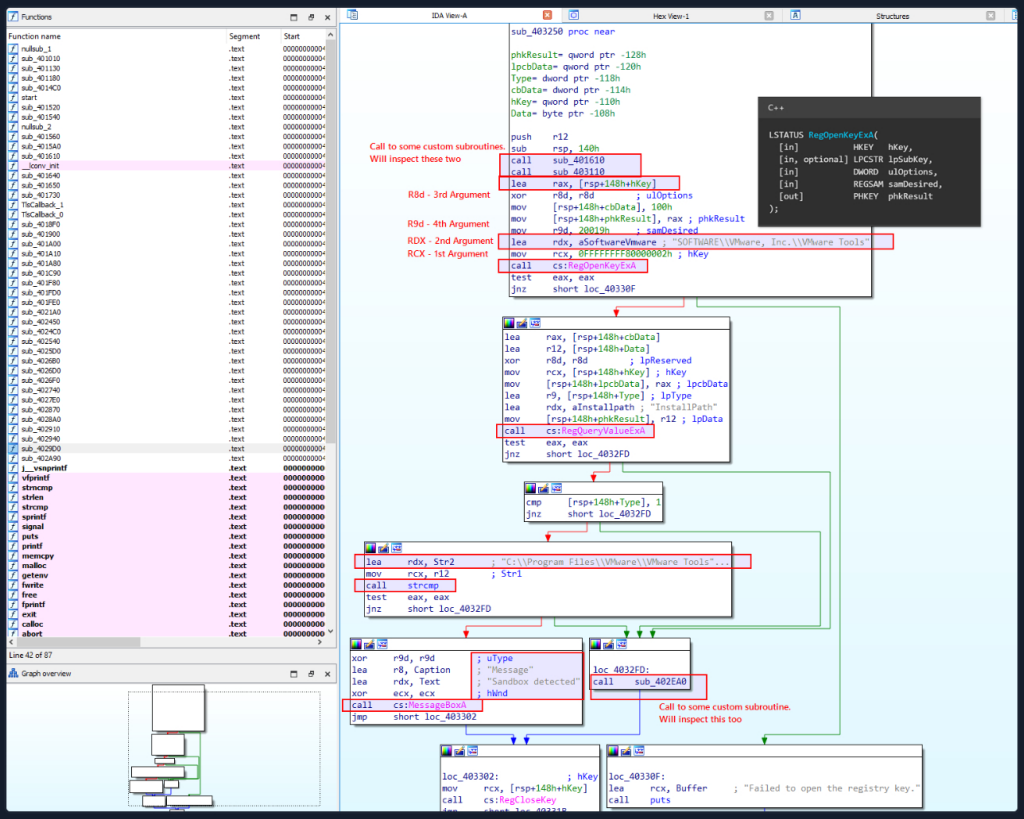
Our objective is to traverse the function calls preceding the program's exit in order to locate the main function, which might contain the initial code for registry check (sandbox detection) we witnessed in process monitor and strings.

We must now navigate to the function `sub_403250` to investigate its contents. To enter this function, we should position the cursor on the call instruction below:

Code: ida

```
call    sub_403250
```

We can right-click on the instruction and select `Jump to Operand` from the context menu, or alternatively, we can press the `Enter` key. This action will reveal the disassembled function for `sub_403250`.



Upon reviewing the instructions, it appears that the function is querying the registry for the value associated with the `SOFTWARE\\VMware, Inc.\\VMware Tools` path and performing a comparison to discern whether VMWare Tools is installed on the machine. Generally speaking, it seems probable that this is the `main` function, which was referenced in the process monitor and strings.

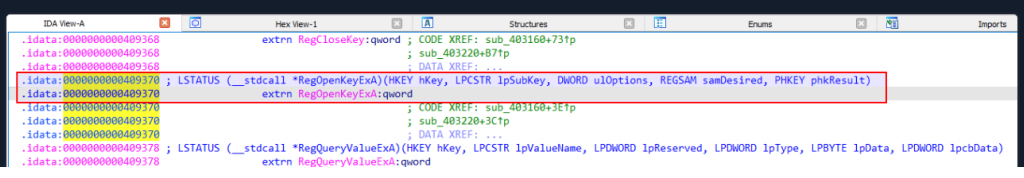
We can observe that the registry query is performed using the function `RegOpenKeyExA`, as shown in the instruction `call cs:RegOpenKeyExA` in the disassembled code that follows:

Code: ida

```
xor     r8d, r8d        ; ulOptions
mov     [rsp+148h+cbData], 100h
mov     [rsp+148h+phkResult], rax ; phkResult
mov     r9d, 20019h     ; samDesired
lea     rdx, aSoftwareVmware ; "SOFTWARE\\VMware, Inc.\\VMware Tools"
mov     rcx, 0FFFFFFFF80000002h ; hKey
call    cs:RegOpenKeyExA
```

In the code block above, the final instruction, `call cs:RegOpenKeyExA`, is presumably a representation of the `RegOpenKeyExA` function call, prefaced by `cs`. The function `RegOpenKeyExA` is a part of the Windows Registry API and is utilized to open a handle to a specified registry key. This function enables access to the Windows registry. The `A` in the function name signifies that it is the `ANSI version` of the function, which operates on ANSI-encoded strings.

In `IDA`, `cs` is a segment register that usually refers to the code segment. When we click on `cs:RegOpenKeyExA` and press `Enter`, this action takes us to the `.idata` section, which includes import-related data and the import address of the function `RegOpenKeyExA`. In this scenario, the `RegOpenKeyExA` function is imported from an external library (advapi32.dll), with its address stored in the `.idata` section for future use.



```
.idata:0000000000409370 ; LSTATUS (__stdcall *RegOpenKeyExA)(HKEY hKey, LPCSTR lpSubKey, DWORD ulOp
.idata:0000000000409370                 extrn RegOpenKeyExA:qword
.idata:0000000000409370                                         ; CODE XREF: sub_403160+3E↑p
.idata:0000000000409370                                         ; sub_403220+3C↑p
.idata:0000000000409370                                         ; DATA XREF: ...
```

This is not the actual address of the `RegOpenKeyExA` function, but rather the address of the entry in the `IAT (Import Address Table)` for `RegOpenKeyExA`. The IAT entry houses the address that will be dynamically resolved at runtime to point to the actual function implementation in the respective DLL (in this case, advapi32.dll).

The line `extrn RegOpenKeyExA:qword` indicates that `RegOpenKeyExA` is an external symbol to be resolved at runtime. This alerts the assembler that the function is defined in another module or library, and the linker will handle the resolution of its address during the linking process.

Reference: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#import-address-table

In actuality, `cs:RegOpenKeyExA` is a means of accessing the `IAT` entry for `RegOpenKeyExA` in the code segment using a relative reference. The actual address of `RegOpenKeyExA` will be resolved and stored in the IAT during runtime by the operating system's dynamic linker/loader.

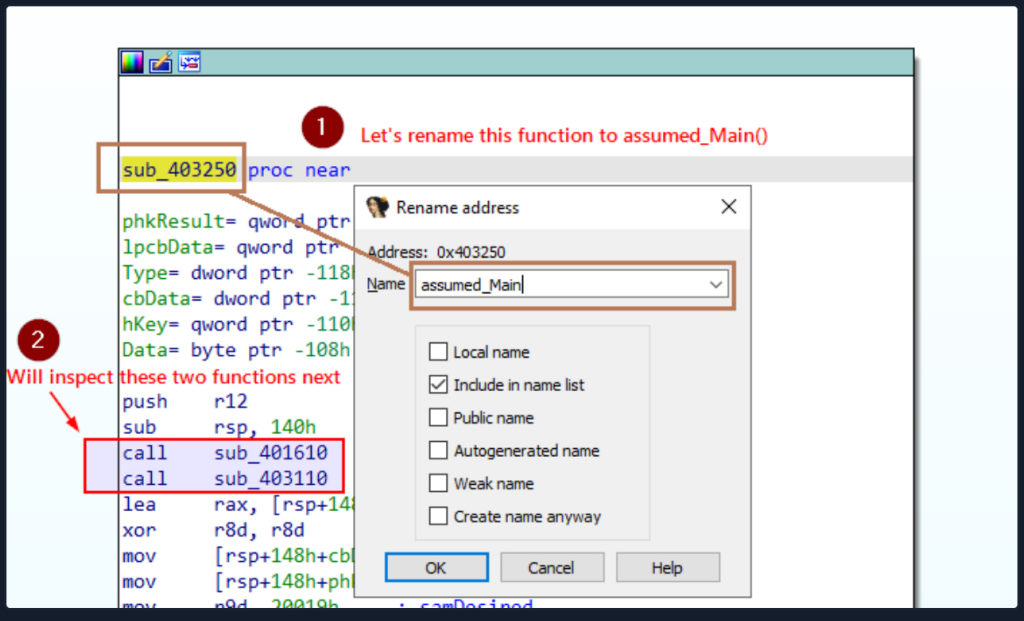
Based on the overall structure of this function, we can conjecture that this is the possible `main` function. Let's rename it to `assumed_Main` for easy recollection in the event we come across references to this function in the future.

To rename a function in `IDA`, we should proceed as follows:

- `Position the cursor on the function name (sub_403250) or the line containing the function definition. Then, press the N key on the keyboard, or right-click and select Rename from the context menu.`
- `Input the new name for the function and press Enter.`

IDA will update the function name throughout the disassembly view and any references to the function within the binary.

Note: Renaming a function in `IDA` does not modify the actual binary file. It only alters the representation within `IDA`'s analysis.

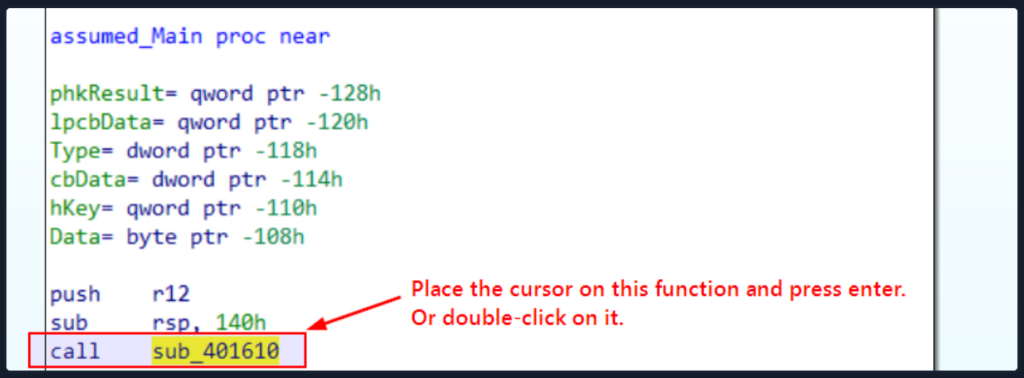


Let's now delve into the instructions present in this block of code.

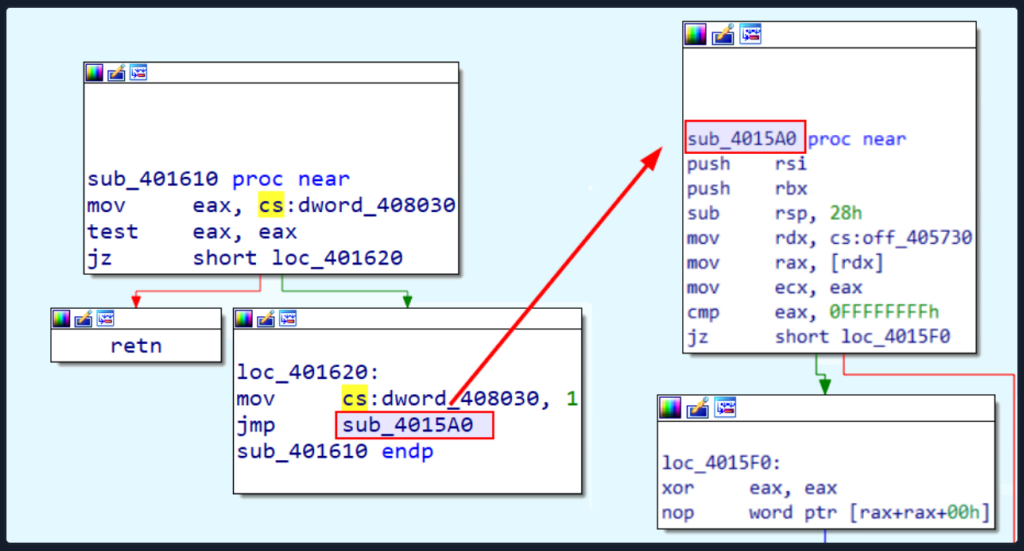
We can identify two function calls emanating from this function (`sub_401610` and `sub_403110`) prior to calling the Windows API function `RegOpenKeyExA`. Let's examine both of these before we advance to the WINAPI functions.

Let's delve into these functions by directing the cursor to their respective `call` instructions and tapping `Enter` to glimpse within.

Begin by examining the disassembled code for the first subroutine `sub_401610`. Initiate the journey into the subroutine by pressing `Enter` on the call instruction for `sub_401610`.



We find ourselves in the first subroutine `sub_401610`, which examines the value of a variable (`cs:dword_408030`). If its value is zero, it is redefined as one. It subsequently redirects to `sub_4015A0`.



The following instructions detail `sub_401610`. Let's strive to comprehend its nuances.

Code: ida

```
sub_401610 proc near

mov     eax, cs:dword_408030
test    eax, eax
jz      short loc_401620

loc_401620:
mov     cs:dword_408030, 1
jmp     sub_4015A0
sub_401610 endp
```

It initiates by transferring the value of the variable `dword_408030` into the `eax` register. It then conducts a `bitwise AND` operation with `eax` and itself, essentially evaluating whether the value is `zero`. If the result of the preceding test instruction deems `eax` as `zero`, it redirects to `sub_4015A0`. Let's dissect its code further.

Code: ida

```
sub_4015A0 proc near
```

```
push    rsi
push    rbx
sub     rsp, 28h
mov     rdx, cs:off_405730
mov     rax, [rdx]
mov     ecx, eax
cmp     eax, 0FFFFFFFFh
jz      short loc_4015F0
```

By pressing `Enter` while the cursor is on the function name `sub_4015A0`, we navigate to the disassembled code, revealing that the function commences by pushing the values of the `rsi` and `rbx` registers onto the stack, preserving the register values. Subsequently, it allots space on the stack by subtracting `28h (40 decimal)` bytes from the stack pointer (`rsp`). It then retrieves a function pointer from the address encapsulated in `off_405730` and stashes it in the `rax` register.

In essence, they seem to execute initialization checks and operations related to function pointers before the program proceeds to call the second subroutine `sub_403110` and the WINAPI function for registry operations. This isn't the actual main function hosting the program logic, so we'll scrutinize other function calls to pinpoint the `main` function.

We can rename this function as `initCheck` for our remembrance by pressing `N` and typing in the new function name.

At this point, we either press the `Esc` key or select the `Jump Back` button in the toolbar to revert to the second subroutine `sub_403110` and explore its inner workings.

Once we've navigated back to the previous function (`assumed_Main`), we should position the cursor on the `call sub_403110` instruction and hit `Enter`.

```
assumed_Main proc near

phkResult= qword ptr -128h
lpcbData= qword ptr -120h
Type= dword ptr -118h
cbData= dword ptr -114h
hKey= qword ptr -110h
Data= byte ptr -108h

push    r12
sub     rsp, 140h
call    sub_401610
call    sub_403110
lea     rax, [rsp+148h+hKey]
xor     r8d, r8d        ; ulOptions
mov     [rsp+148h+cbData], 100h
mov     [rsp+148h+phkResult], rax ; phkResult
mov     r9d, 20019h     ; samDesired
lea     rdx, aSoftwareVmware ; "SOFTWARE\\VMware, Inc.\\VMware Tools"
mov     rcx, 0FFFFFFFF80000002h ; hKey
call    cs:RegOpenKeyExA
```

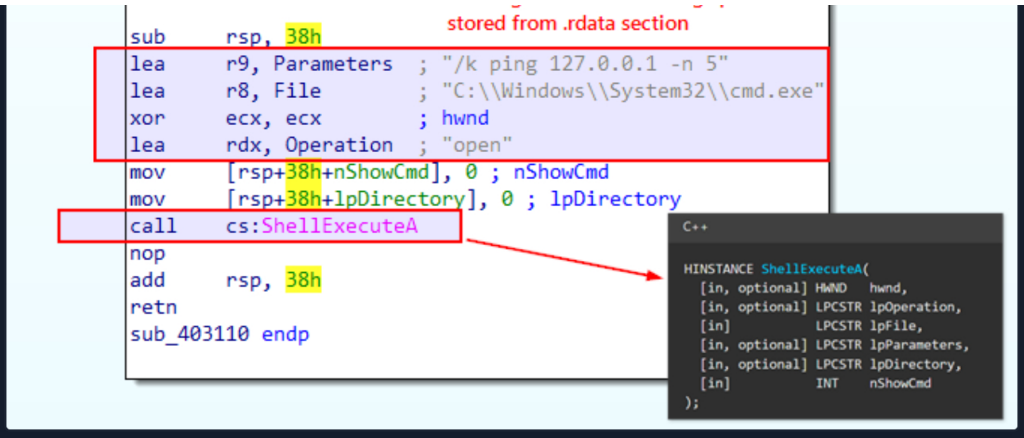
This transition lands us in the disassembled code for this function. Let's examine it to determine its operation.

```
.rdata:0000000000405266 ; const CHAR Operation[]
.rdata:0000000000405266 Operation       db 'open',0             ; DATA XREF: sub_403110+14to
.rdata:0000000000405266                                         ; sub_403150+Bto
.rdata:00000000004052A8 ; const CHAR Parameters[]
.rdata:00000000004052A8 Parameters      db '/k ping 127.0.0.1 -n 5',0
.rdata:00000000004052A8                                         ; DATA XREF: sub_403110+4to
.rdata:00000000004052BF ; const CHAR File[]
.rdata:00000000004052BF File            db 'C:\Windows\System32\cmd.exe',0

sub_403110 proc near

lpDirectory= qword ptr -18h
nShowCmd= dword ptr -10h
```

Loading addresses of strings parameters



The variables `Parameters`, `File` and `Operation` are string variables stowed in the `.rdata` section of the executable. The `lea` instructions are utilized to obtain the memory addresses of these strings, which are subsequently passed as arguments to the `ShellExecuteA` function. This block of code is accountable for a `sleep` duration of `5 seconds`. Following that, it reverts to the preceding function. Having understood the code, we can rename this function as `pingSleep` by right-clicking and choosing rename.

Now that we've encountered some references for Windows API functions, let's elucidate how WINAPI functions are interpreted in the disassembled code.

After investigating the operations within the two function calls (`sub_401610` and `sub_403110`) from this function and before invoking the Windows API function `RegOpenKeyExA`, let's inspect the calls made to WINAPI function `RegOpenKeyExA`. In this `IDA` disassembly view, the arguments passed to the WINAPI function call are depicted above the `call` instruction. This standard convention in disassemblers offers a lucid representation of the function call along with its corresponding arguments.

The Windows API function, `RegOpenKeyExA`, is utilized here to unlock a registry key. The syntax of this function, as per Microsoft documentation, is presented below.

Code: ida

```
LSTATUS RegOpenKeyExA(
  [in]           HKEY   hKey,
  [in, optional] LPCSTR lpSubKey,
  [in]           DWORD  ulOptions,
  [in]           REGSAM samDesired,
  [out]          PHKEY  phkResult
);
```

C++ Copy

```
LSTATUS RegOpenKeyExA(
  [in]           HKEY   hKey,
  [in, optional] LPCSTR lpSubKey,
  [in]           DWORD  ulOptions,
  [in]           REGSAM samDesired,
  [out]          PHKEY  phkResult
);
```

Let's deconstruct the code for this function as it appears in the `IDA` disassembled view.

Code: ida

```
lea     rax, [rsp+148h+hKey]      ; Calculate the address of hKey
xor     r8d, r8d                  ; Clear r8d register (ulOptions)
mov     [rsp+148h+phkResult], rax ; Store the calculated address of hKey in phkResult
mov     r9d, 20019h               ; Set samDesired to 0x20019h (which is KEY_READ in MS-DOCS)
lea     rdx, aSoftwareVmware      ; Load address of string "SOFTWARE\\VMware, Inc.\\VMware Tools"
mov     rcx, 0FFFFFFFF80000002h   ; Set hKey to 0xFFFFFFFF80000002h (HKEY_LOCAL_MACHINE)
call    cs:RegOpenKeyExA          ; Call the RegOpenKeyExA function
test    eax, eax                  ; Check the return value
```

```
jnz     short loc_40330F          ; Jump if the return value is not zero (error condition)
```

The `lea` instruction calculates the address of the `hKey` variable, presumably a handle to a registry key. Then, `mov rcx, 0FFFFFFFF80000002h` pushes `HKEY_LOCAL_MACHINE` as the first argument (`rcx`) to the function. The `lea rdx, aSoftwareVmware` instruction employs the `load effective address (LEA)` operation to calculate the effective address of the memory location storing the string `Software\\VMware, Inc.\\VMware Tools`. This calculated address is then stowed in the `rdx` register, the function's second argument.

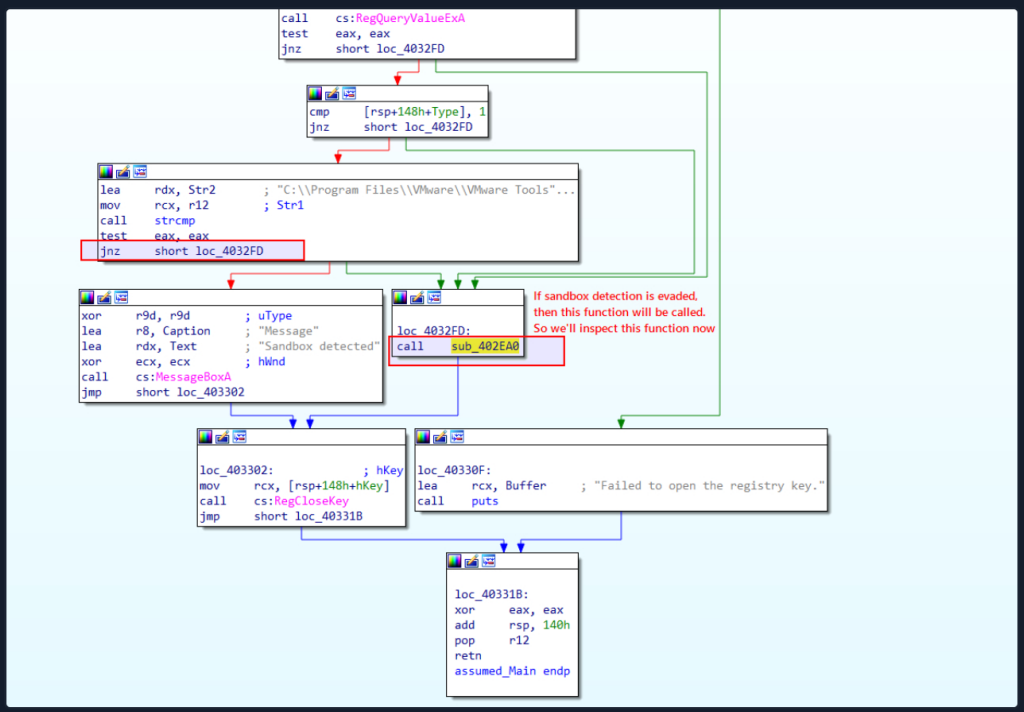
The third argument to this function is passed to the `r8d` register via the instruction `xor r8d, r8d` which empties the `r8d` register by implementing an `XOR` operation with itself, effectively resetting it to zero. In the context of this code, it indicates that the third argument (`ulOptions`) passed to the `RegOpenKeyExA` function bears a value of `0`.

The fourth argument is `mov r9d, 20019h`, corresponding to `KEY_READ` in MS-DOCS.

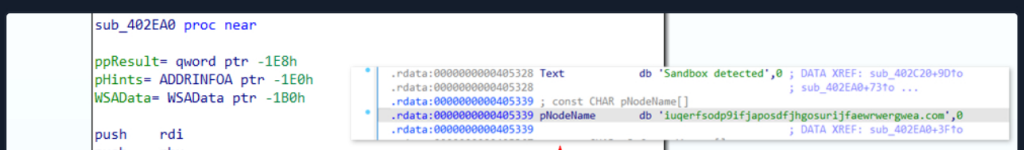
The fifth argument, `phkResult`, is on the stack. By adding `rsp+148h` to the base stack pointer `rsp`, the code accesses the memory location on the stack where the `phkResult` parameter resides. The `mov [rsp+148h+phkResult], rax` instruction duplicates the value of `rax` (which holds the address of `hKey`) to the memory location pointed to by `phkResult`, essentially storing the address of `hKey` in `phkResult` (which is passed to the next function as the first argument).

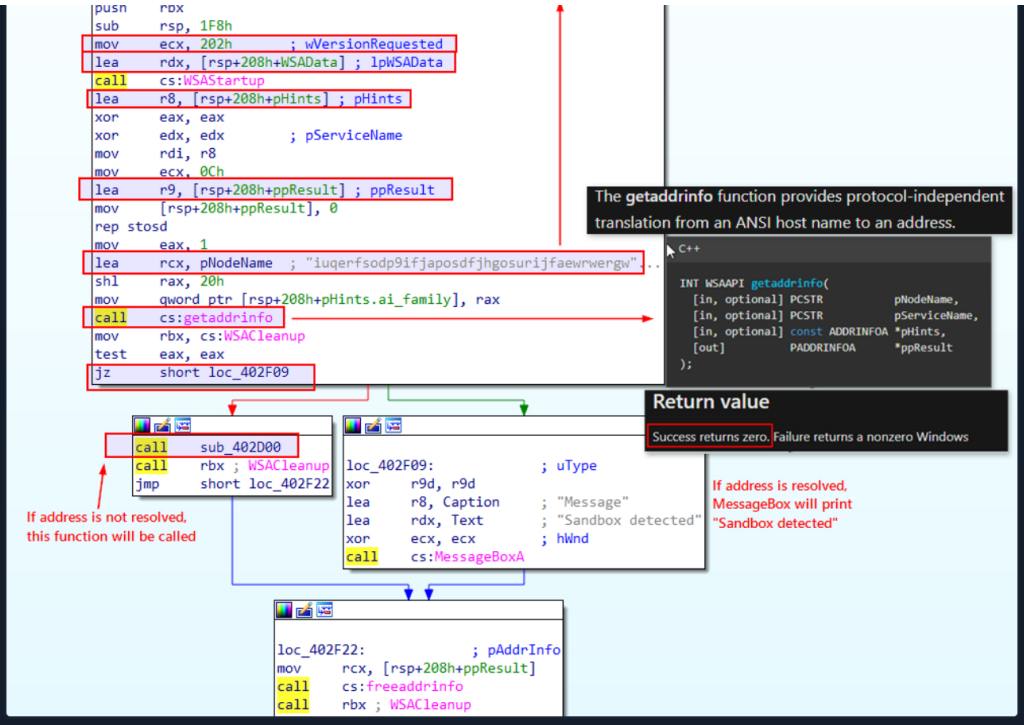
From this point onward, whenever we stumble upon a WINAPI function reference in the code, we'll resort to the Microsoft documentation for that function to grasp its syntax, parameters, and the return value. This will assist us in understanding the probable values in the registers when these functions are invoked.

Should we scroll down the graph view, we encounter the next WINAPI function `RegQueryValueExA` which retrieves the type and data for the specified value name associated with an open registry key. The key data is compared, and upon a match, a message box stating `Sandbox Detected` is displayed. If it does not match, then it redirects to another subroutine `sub_402EA0`. We'll also rectify this sandbox detection in the debugger later. The image below outlines the overall flow of this operation.



Let's press `Enter` on the upcoming call instruction for the function `sub_402EA0` to enable us to scrutinize this subroutine and figure out its operations.





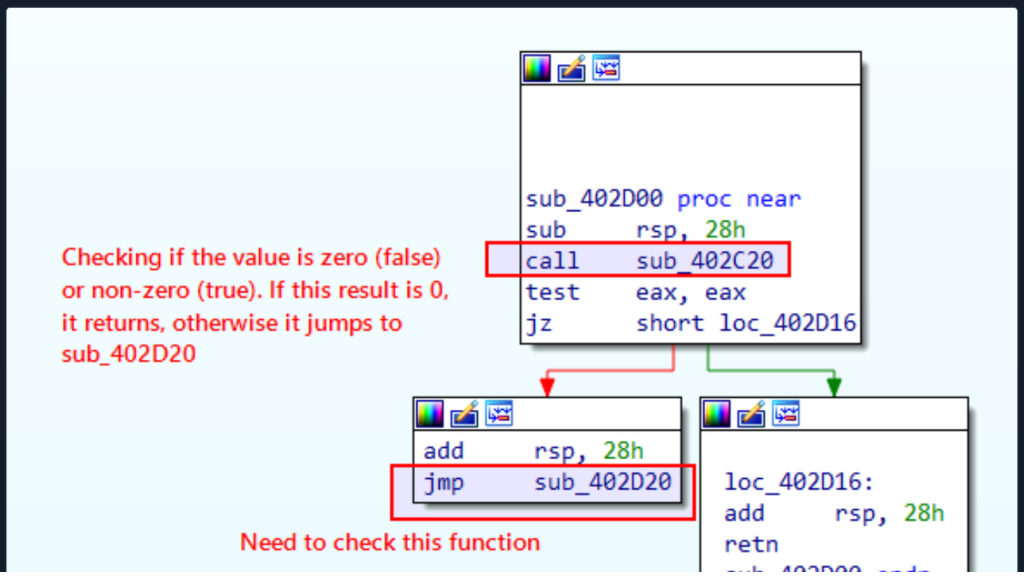
Upon pressing `Enter`, we uncover its functionality. This subroutine seems to execute network-related operations using the `Windows Sockets API (Winsock)`. It initially invokes the `WSAStartup` function to set up the `Winsock` library, then it calls the `WSAAPI` function `getaddrinfo` which is used to fetch address information for the specified node name (`pNodeName`) based on the provided hints `pHints`. The subroutine verifies the success of the address resolution using the `getaddrinfo` function.

If the `getaddrinfo` function yields a return value of `zero` (indicating success), this implies that the address has been successfully resolved to an IP. Following this event, if indeed successful, the sequence jumps to a MessageBox which displays `Sandbox detected`. If not, it directs the flow to the subroutine `sub_402D00`.

Subsequently, it prompts the invocation of the `WSACleanup` function. This action initiates the cleanup of resources related to `Winsock`, irrespective of whether the address resolution process was successful or unsuccessful. For the sake of clarity, we'll christen this function as `DomainSandboxCheck`.

Possible IOC: Kindly note the domain name `iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com` as a component of potential `IOC`s.

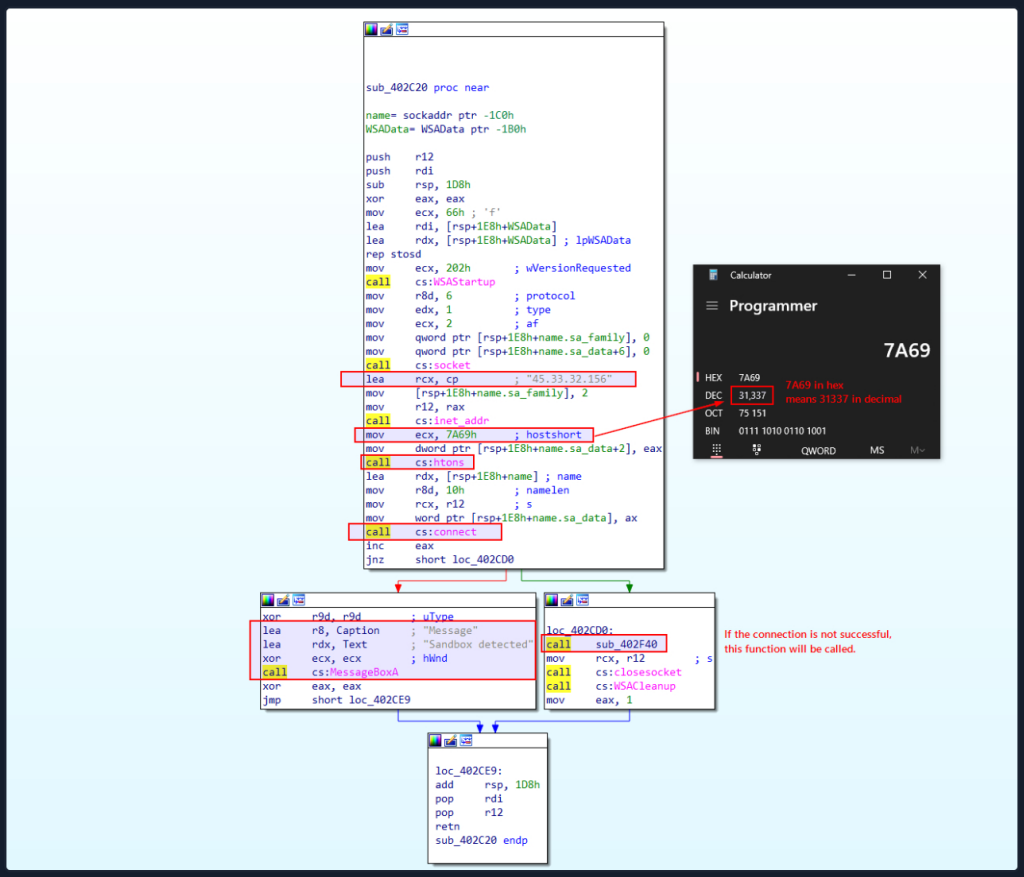
To explore the consequences of bypassing the sandbox check, we'll delve into the subroutine `sub_402D00`. We can scrutinize this subroutine by hitting `Enter` on the ensuing `call` instruction related to the `sub_402D00` function. An image attached below displays the disassembled code for this function.



```
sub_402D00 endp
```

This function first reserves space on the stack for local variables before calling `sub_402C20`, a distinct function. The output of this function is then stored within the `eax` register. Depending on the results derived from the `sub_402C20` function, the sequence either returns (`retn`) or leaps to `sub_402D20`.

Consequently, we'll select the first highlighted function, `sub_402C20`, by pressing `Enter` to examine its instructions. Upon thorough analysis of `sub_402C20`, we'll loop back to this block to evaluate the second highlighted function, `sub_402D20`.



Upon hitting `Enter`, we are greeted with its instructions as portrayed in the image above. This function initiates the `Winsock` library, generates a socket, and connects to IP address `45.33.32.156` via port `31337`. It evaluates the return value (`eax`) to ascertain if the connection was successful. However, there is a twist; post-function invocation, the instruction `inc eax` increments the `eax` register's value by `1`. Subsequent to the `inc eax` instruction, the code appraises the value of `eax` using the `jnz (jump if not zero)` instruction.

Should the connection to the aforementioned port and IP address fail, this function should return `-1`, as specified in the documentation.

Handling Winsock Errors

Article • 01/08/2021 • 3 contributors

Feedback

In this article

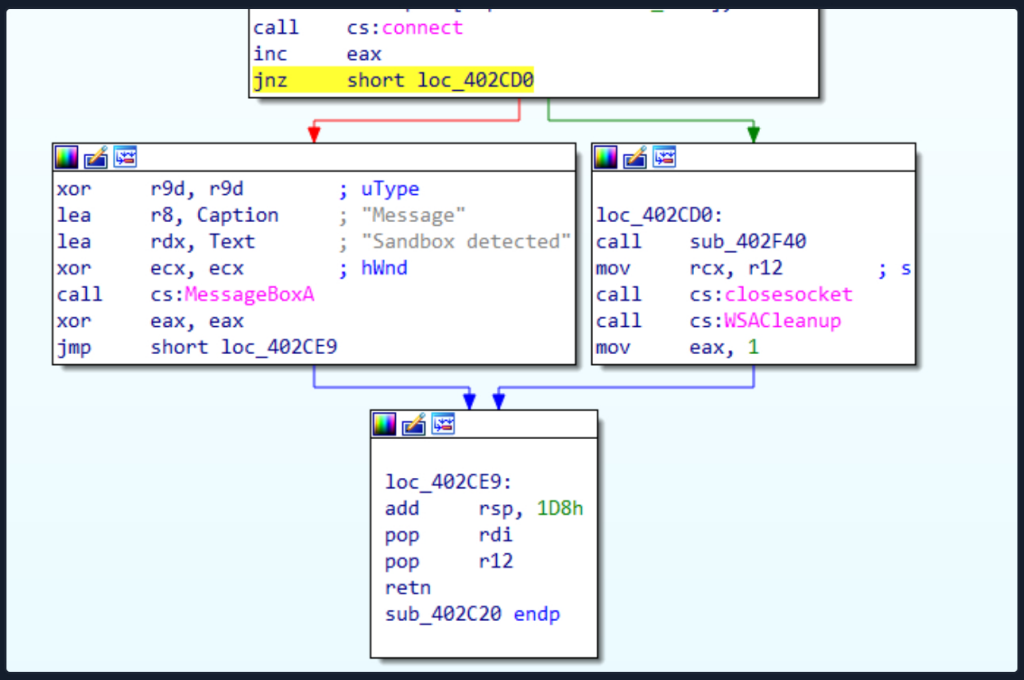
Related topics

Most Windows Sockets 2 functions do not return the specific cause of an error when the function returns. Some Winsock functions return a value of zero if successful. Otherwise, the value SOCKET_ERROR (-1) is returned and a specific error number can be retrieved by calling the WSAGetLastError function. For Winsock functions that return a handle, a return value of INVALID_SOCKET (0xffff) indicates an error and a specific error number can be retrieved by calling **WSAGetLastError**. For Winsock functions that return a pointer, a return value of **NULL** indicates an error and a specific error number can be retrieved by calling the **WSAGetLastError** function.

Code: ida

```
call    cs:connect
inc     eax
jnz     short loc_402CD0
```

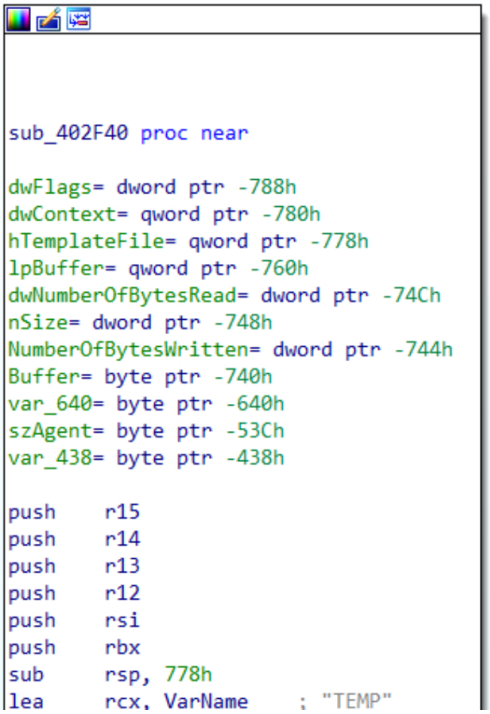
Given that `eax` is incremented by `1` post-function call, this should reduce to `0`. Consequently, the `MessageBox` will print `Sandbox detected`. This implies that the function is examining the state of the internet connection.



If, on the other hand, the connection is successful, it will produce a `non-zero` value, prompting the code to leap to `loc_402CD0`. This location houses a call to another function, `sub_402F40`. With a clear understanding of this function's operations, we'll rename it as `InternetSandboxCheck`.

Possible IOC: Remember to note this IP address `45.33.32.156` and port `31337` as components of potential `IOC`s.

Next, we'll proceed to function `sub_402F40` to decipher its operations. We can do this by right-clicking and selecting `Jump to Operand`, or by pressing `Enter` on its `call` instruction.



```
call    getenv
lea     r15, [rsp+7A8h+Buffer]
lea     r9, aSvchostExe ; "svchost.exe"
mov     r8, rax
lea     rdx, aSS        ; "%s\\%s"
mov     rcx, r15        ; Buffer
call    sprintf
lea     r9, [rsp+7A8h+var_640]
lea     rdx, [rsp+7A8h+nSize] ; nSize
mov     [rsp+7A8h+nSize], 104h
mov     [rsp+7A8h+lpBuffer], r9
mov     rcx, r9         ; lpBuffer
call    cs:GetComputerNameA
mov     r9, [rsp+7A8h+lpBuffer]
test    eax, eax
jz      loc_4030F8
```

This function calls upon the `getenv` function (with `rcx` acting as the argument passer for `TEMP`) and saves its result in the `eax` register. This action retrieves the `TEMP` environment variable's value.

Code: ida

```
lea     rcx, VarName    ; "TEMP"
call    getenv
```

To verify the output, we can use powershell to print the `TEMP` environment variable's value.

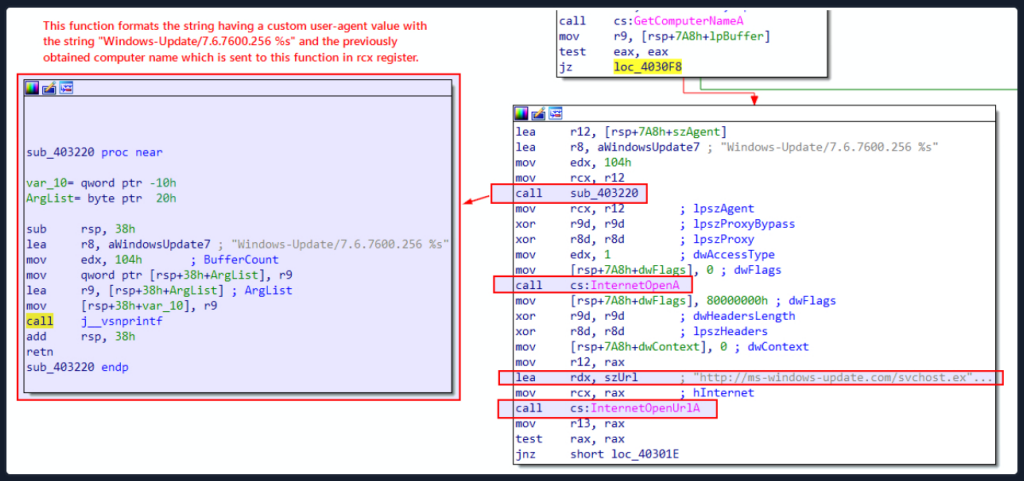
Code Analysis

```
PS C:\> Get-ChildItem env:TEMP

Name                           Value
----                           -----
TEMP                           C:\Users\htb-student\AppData\Local\Temp
```

It then employs the `sprintf` function to append the obtained `TEMP` path to the string `svchost.exe`, yielding a complete file path. Thereafter, the `GetComputerNameA` function is called to retrieve the computer's name, which is then stored in a buffer.

If the computer name is non-existent, it skips to the label `loc_4030F8` (which houses instructions for returning). Conversely, if the computer name is not empty (`non-zero` value), the code progresses to the subsequent instruction as displayed on the left side of the image.



In subsequent instructions, we find a call to the function `sub_403220`. We can access it by double-clicking on the function name.

The left side of the attached image above displays the function `sub_403220`, which formats a string housing a custom `user-agent` value with the string `Windows-Update/7.6.7600.256 %s`. The `%s` placeholder is replaced with the previously

obtained computer name, which is transmitted to this function in the `rcx` register.

```
sub_403220 proc near

var_10= qword ptr -10h
ArgList= byte ptr  20h

sub     rsp, 38h
lea     r8, aWindowsUpdate7 ; "Windows-Update/7.6.7600.256 %s"
mov     edx, 104h       ; BufferCount
mov     qword ptr [rsp+38h+ArgList], r9
lea     r9, [rsp+38h+ArgList] ; ArgList
mov     [rsp+38h+var_10], r9
call    j__vsnprintf
add     rsp, 38h
retn
sub_403220 endp
```

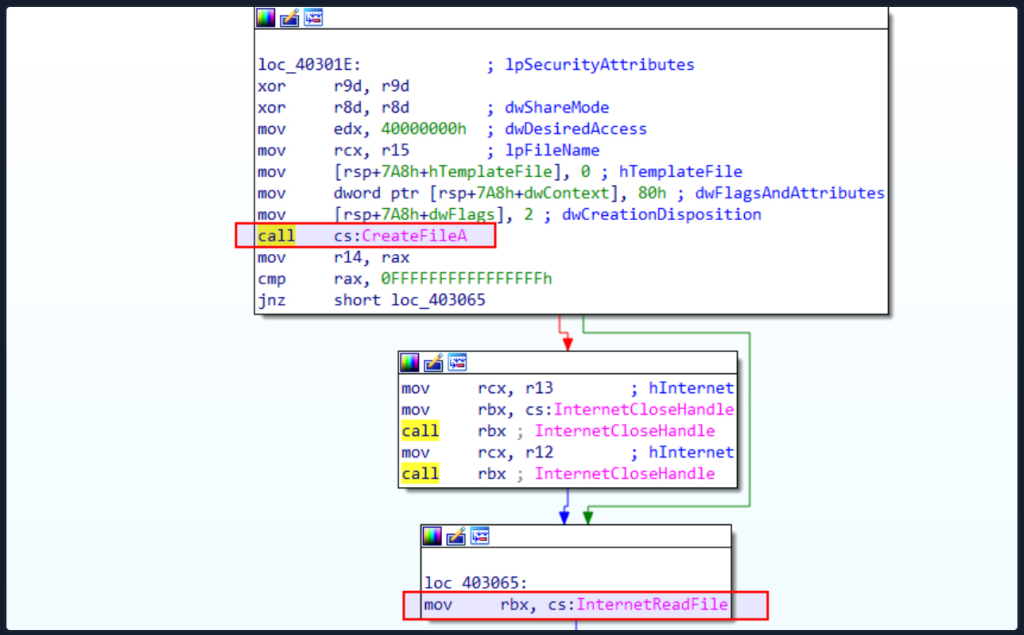
Now, the complete value reads `Windows-Update/7.6.7600.256 HOSTNAME`, where `HOSTNAME` is the result of `GetComputerNameA` (the computer's name).

It's crucial to note this unique custom `user-agent`, wherein the hostname is also transmitted in the request when the malware initiates a network connection.

Back to the previous function, it subsequently calls the `InternetOpenA` WINAPI function to commence an internet access session and configure the parameters for the `InternetOpenUrlA` function. It then proceeds to call the latter to open the URL `http://ms-windows-update.com/svchost.exe`.

Possible IOC: Do note this URL `http[:]//ms-windows-update[.]com/svchost[.]exe` as potential `IOC`. The malware is downloading an additional executable from this location.

If the URL opens successfully, the code leaps to the label `loc_40301E`. Let's probe the instructions at `loc_40301E` by double-clicking on it.

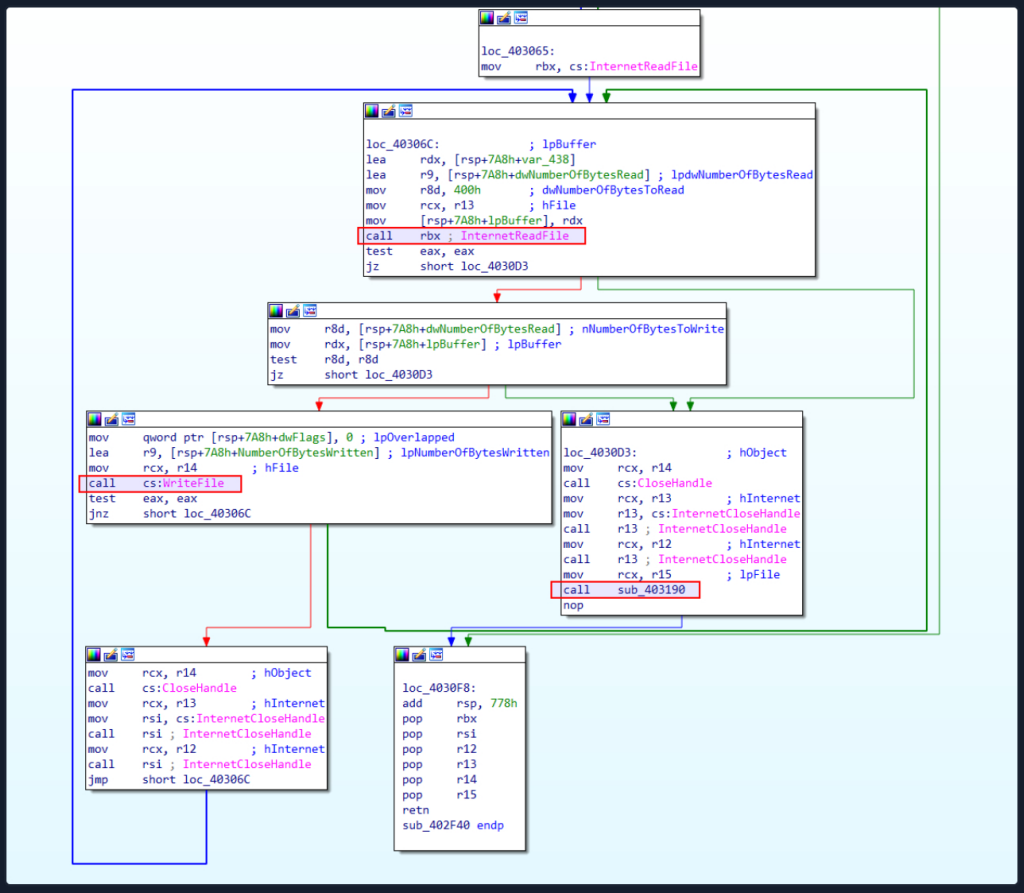


Upon opening the function, we observe a call to the Windows API function `CreateFileA`, which is used to generate a file on the local system, designating the previously obtained file path.

The code then enters a loop, repeatedly invoking the `InternetReadFile` function to pull data from the opened URL `http[:]//ms-windows-update[.]com/svchost[.]exe`. If the data reading operation proves successful, the code advances to write the received data to the created file (`svchost.exe` located in the `TEMP` directory) using the `WriteFile` function.

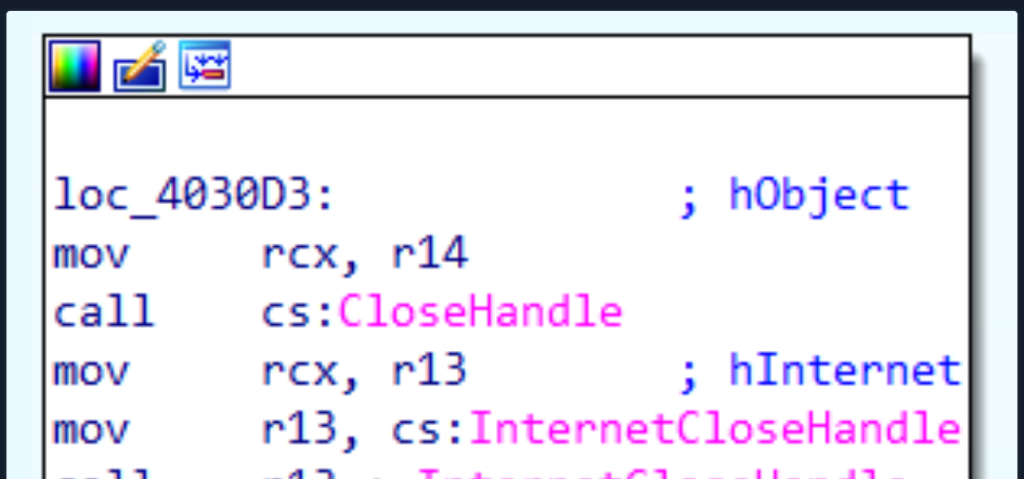
Note this unique technique, where the malware downloads and deposits an executable file `svchost.exe` in the `temp` directory.

The aforementioned loop is illustrated in the image below.



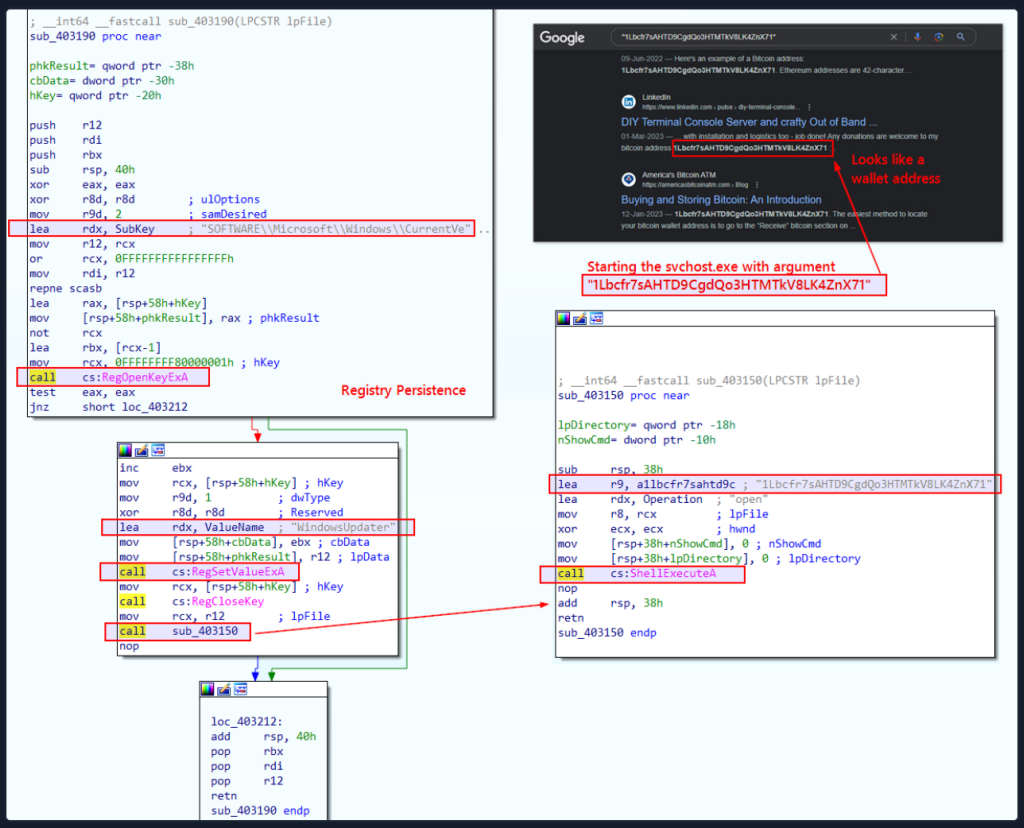
After the data writing operation, the code cycles back to read more data until the `InternetReadFile` function returns a value that indicates the end of the data stream. Once all data has been read and written, the opened file and the internet handles are closed using the appropriate functions (`CloseHandle` and `InternetCloseHandle`). Subsequently, the code leaps to `loc_4030D3`, where it calls upon the function `sub_403190`.

We'll double-click on `sub_403190` to unveil its contents.



```
call    r13 ; InternetCloseHandle
mov     rcx, r12        ; hInternet
call    r13 ; InternetCloseHandle
mov     rcx, r15        ; lpFile
call    sub_403190
nop
```

The function `sub_403190` is now exposed, revealing a series of WINAPI calls related to registry modifications, such as `RegOpenKeyExA` and `RegSetValueExA`.



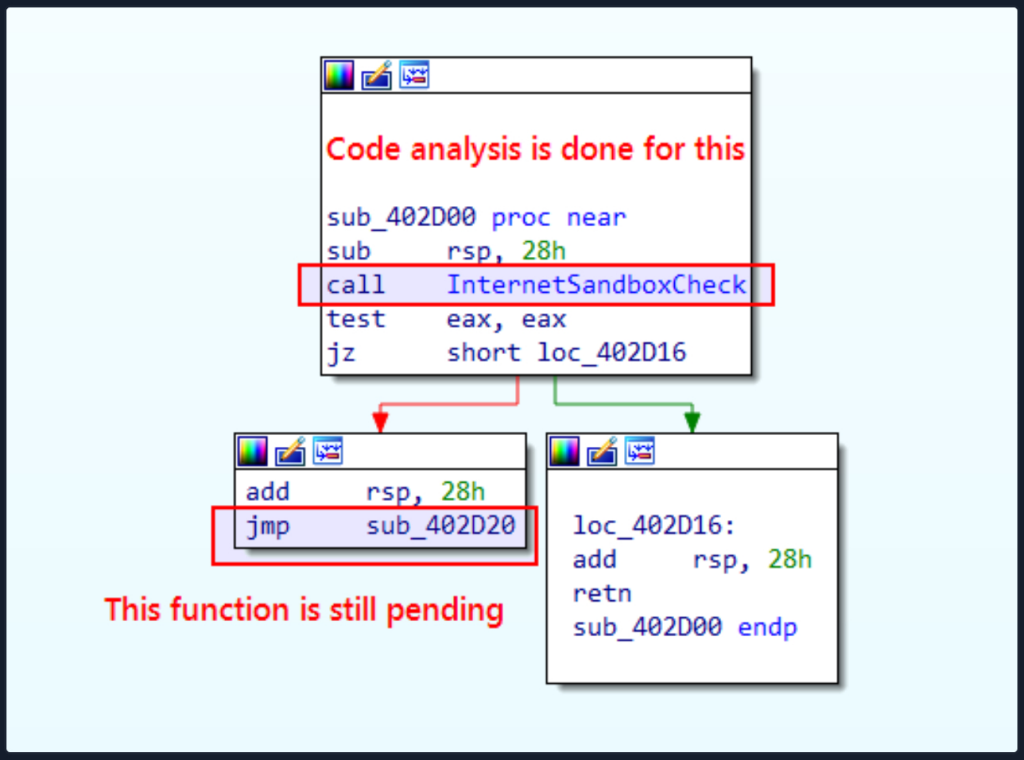
It appears that this function places the file (`svchost.exe` located in the `TEMP` directory) into the registry key path `SOFTWARE\Microsoft\Windows\CurrentVersion\Run` with the value name `WindowsUpdater`, then seals the registry key. This technique is frequently employed by both malware and legitimate applications to maintain their grip on the system across reboots, ensuring automatic operation each time the system initiates or a user logs in. We've taken the liberty of renaming this function in `IDA` to `persistence_registry` for the sake of clarity.

```
.rdata:000000000040526B ; const CHAR SubKey[]
.rdata:000000000040526B SubKey          db 'SOFTWARE\Microsoft\Windows\CurrentVersion\Run',0
.rdata:000000000040526B                                         ; DATA XREF: sub_403190+13↑o
.rdata:0000000000405299 ; const CHAR ValueName[]
.rdata:0000000000405299 ValueName       db 'WindowsUpdater',0   ; DATA XREF: sub_403190+58↑o
```

Possible IOC: Highlight this technique in which the malware modifies the registry to achieve persistence. It does so by adding an entry for `svchost.exe` under the `WindowsUpdater` name in the `SOFTWARE\Microsoft\Windows\CurrentVersion\Run` registry key.

Upon establishing the registry, it initiates another function, `sub_403150`, which sets in motion the dropped file `svchost.exe` and funnels an argument into it. A rudimentary Google search suggests that this argument could potentially be a `Bitcoin wallet address`. Thus, it's reasonable to postulate that the dropped executable could be a coin miner.

By rewinding our steps and inspecting the functions systematically, we can identify any residual functions that we've not yet scrutinized. The `Esc` key or the `Jump Back` button in the toolbar facilitates this reverse tracking.



After tracing back on the analysed code, we've reached this block, where a subroutine `sub_402D20` is pending for analysis. So let's double click to open it and see what's inside it.

```
sub_402D20 proc near

bInheritHandles= dword ptr -2A8h
dwCreationFlags= dword ptr -2A0h
lpEnvironment= qword ptr -298h
lpCurrentDirectory= qword ptr -290h
lpStartupInfo= qword ptr -288h
lpProcessInformation= qword ptr -280h
ProcessInformation= _PROCESS_INFORMATION ptr -278h
StartupInfo= _STARTUPINFOA ptr -260h
Buffer= byte ptr -1F5h

push    r12
push    rdi
push    rsi
push    rbx
sub     rsp, 2A8h
xor     eax, eax
mov     ecx, 1Ah
lea     rsi, unk_405057
xor     r9d, r9d        ; lpThreadAttributes
xor     r8d, r8d        ; lpProcessAttributes
lea     rdx, CommandLine ; "C:\\Windows\\System32\\notepad.exe"
lea     rdi, [rsp+2C8h+StartupInfo]
rep stosd
lea     rdi, [rsp+2C8h+ProcessInformation]
mov     ecx, 6
mov     [rsp+2C8h+StartupInfo.cb], 68h ; 'h'
rep stosd
lea     rax, [rsp+2C8h+ProcessInformation]
lea     rdi, [rsp+2C8h+Buffer]
mov     ecx, 1CDh       ; lpApplicationName
rep movsb
mov     [rsp+2C8h+lpProcessInformation], rax ; lpProcessInformation
lea     rax, [rsp+2C8h+StartupInfo]
mov     [rsp+2C8h+lpStartupInfo], rax ; lpStartupInfo
mov     [rsp+2C8h+lpCurrentDirectory], 0 ; lpCurrentDirectory
mov     [rsp+2C8h+lpEnvironment], 0 ; lpEnvironment
mov     [rsp+2C8h+dwCreationFlags], 4 ; dwCreationFlags
mov     [rsp+2C8h+bInheritHandles], 0 ; bInheritHandles
call    cs:CreateProcessA
test    eax, eax
jz      loc_402E89
```

Upon opening the subroutine, it's clear that it's setting up the necessary parameters for the `CreateProcessA` function to generate a new process. It then proceeds to instigate a new process, `notepad.exe`, situated in the `C:\Windows\System32` directory.

Here is the syntax for the CreateProcessA function.

Code: ida

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

With `rdx` observed in the code, we see that the second argument to this function is pinpointed as `C:\\Windows\\System32\\notepad.exe`.

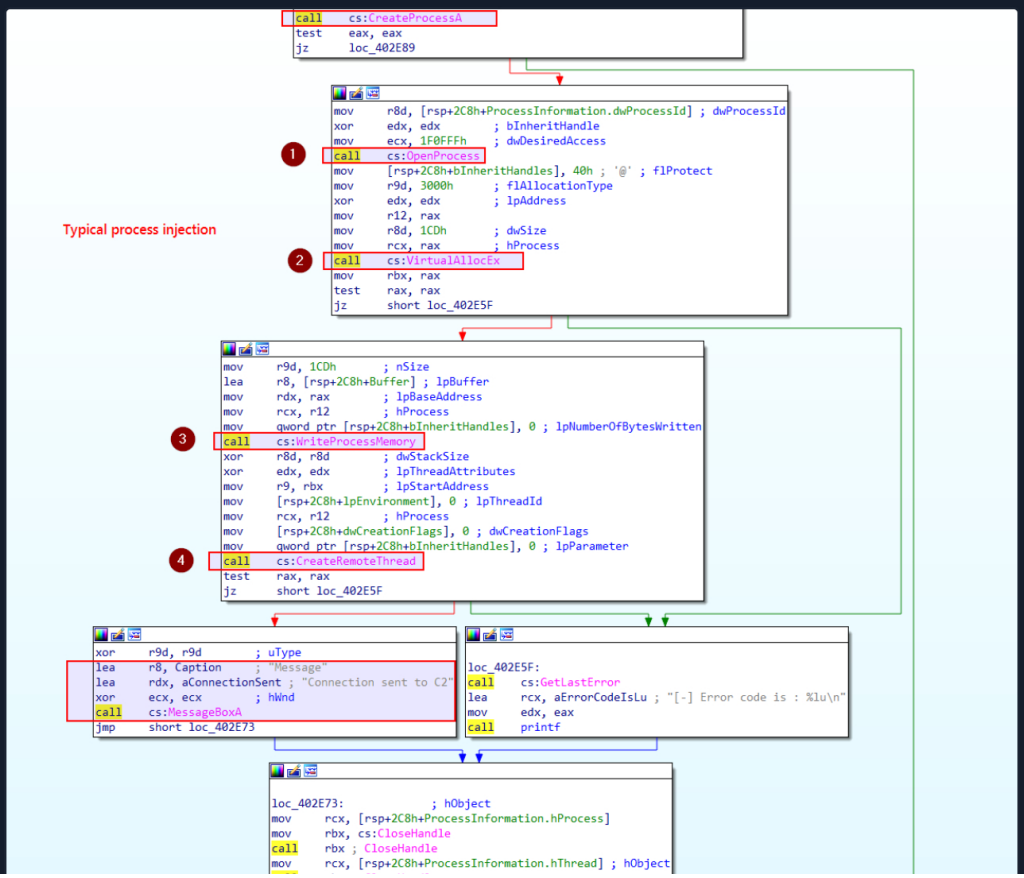
Return value

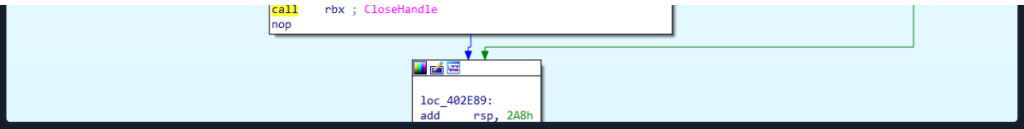
If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call GetExitCodeProcess.

We note in the `CreateProcessA` function documentation that a `nonzero` return value indicates successful function execution. Consequently, if successful, it won't jump to `loc_402E89` but will continue to the next block of instructions.



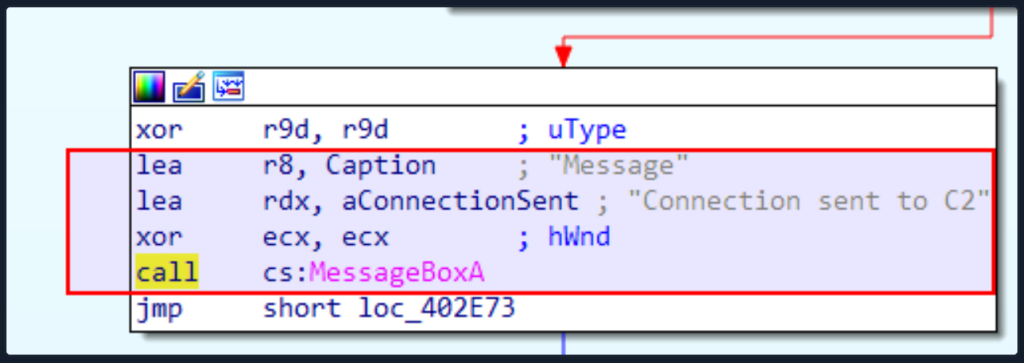


The subsequent block of instructions hints at a commonplace type of process injection, wherein `shellcode` is inserted into the newly created process using `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` functions.

Let's decipher the process injection based on our observations of the code.

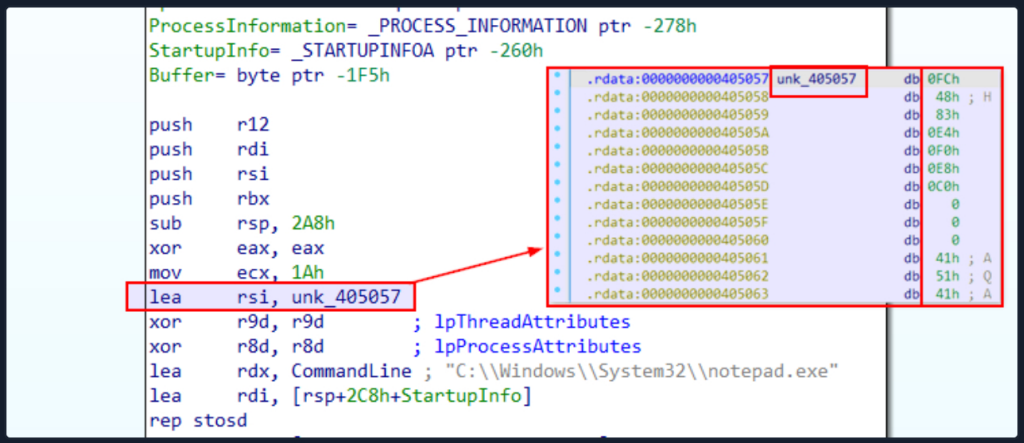
A fresh `notepad.exe` process is fabricated via the `CreateProcessA` function. Following this, memory is allocated within this process using `VirtualAllocEx`. The `shellcode` is then inscribed into the allocated memory of the remote process `notepad.exe` using the WINAPI function `WriteProcessMemory`. Lastly, a remote thread is established in `notepad.exe`, initiating the `shellcode` execution via the `CreateRemoteThread` function.

If the injection is triumphant, a message box manifests, declaring `Connection sent to C2`. Conversely, an error message surfaces in the event of failure.

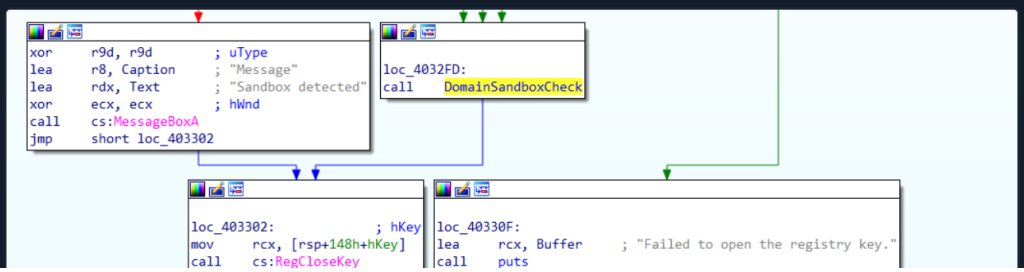


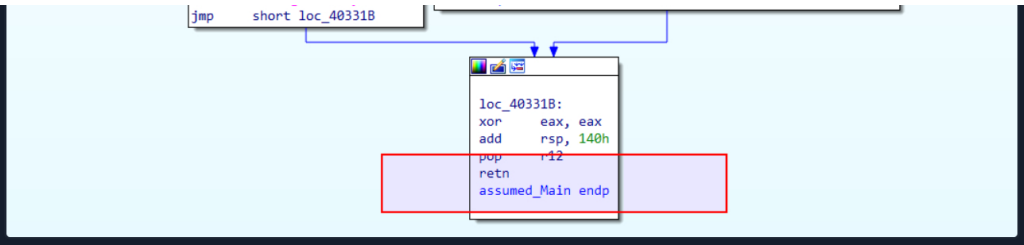
For the sake of ease, let's rename the function `sub_402D20` as `process_Injection`.

At the outset of this function, we can spot an unknown address `unk_405057`, the effective address of which is loaded into the `rsi` register via the instruction `lea rsi, unk_405057`. Executed prior to the WINAPI functions call for the process injection, the reason for loading the effective address into `rsi` could be manifold - it might function as a data-accessing pointer or as a function call argument. There is, however, the possibility that this address houses potential `shellcode`. We will verify this when `debugging` these WINAPI functions using a debugger like `x64dbg`.



Upon analyzing and renaming this process injection function, we will continue to retrace our steps to the preceding functions to ensure that no function has been overlooked.

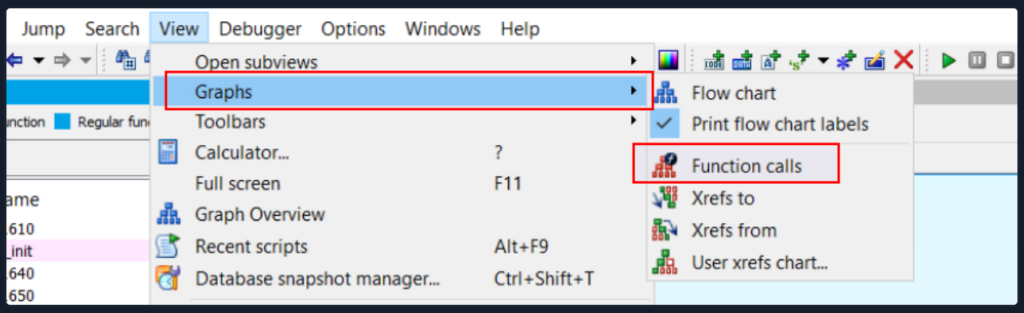




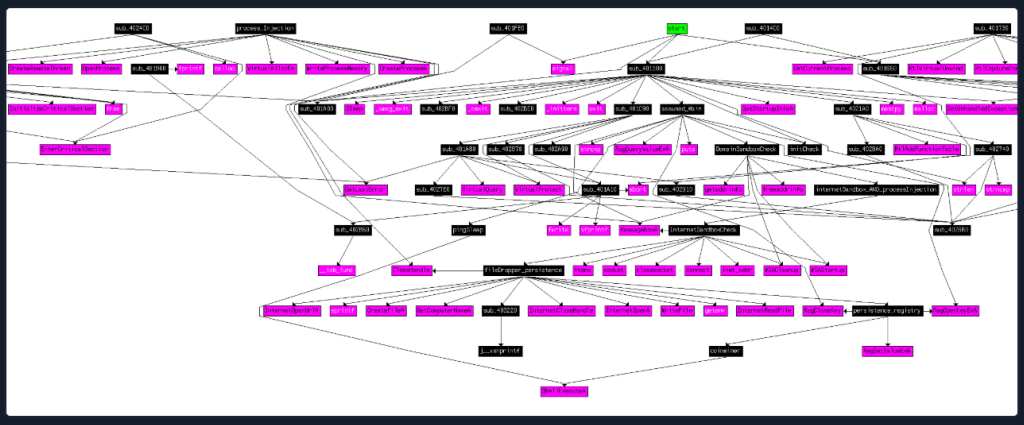
IDA also offers a feature that visualizes the execution flow between functions in an executable via a call flow graph. This potent visual tool aids analysts in navigating and understanding the control flow and the interactions among functions.

Here's how to generate and examine the graph to identify the links among different functions:

- Switch to the disassembly view.
- Locate the View menu at the top of the IDA interface.
- Hover over the Graphs option.
- From the submenu, choose Function calls.



IDA will then forge the function calls flow graph for all functions in the binary and present it in a new window. This graph offers an overview of the calls made between the various functions in the program, enabling us to scrutinize the control flow and dependencies among functions. An example of how this graph appears is shown in the screenshot below.



Contrary to viewing the relationship graph for all function calls, we can also focus on specific functions. To generate the reference graph for the function calls flow related to a specific function, these steps can be followed.

- Navigate to the function whose function call flow graph we wish to examine.
- To open the function in the disassembly view, either double-click the function name or press Enter.
- In the disassembly view, right-click anywhere and opt for either Xrefs graph to... or Xrefs graph from..., based on whether we want to observe the function calls made by the selected function or the function calls leading to the selected function.
- IDA will craft the function calls flow graph and exhibit it in a new window.

Looking ahead, we will delve into debugging in the subsequent section. There, we'll launch the executable within a debugger and establish breakpoints on the requisite instructions and select critical WINAPI functions. This strategy allows us to comprehend and manage the execution flow in real time as the program operates.

VPN Servers

⚠ **Warning:** Each time you "Switch", your connection keys are regenerated and you must re-download your VPN connection file.

All VM instances associated with the old VPN Server will be terminated when switching to a new VPN server.

Existing PwnBox instances will automatically switch to the new VPN server.

US Academy 3 — Medium Load

PROTOCOL

UDP 1337 TCP 443

DOWNLOAD VPN CONNECTION FILE

Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK — 160ms

Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!

Download VPN Connection File

Target(s): Click here to spawn the target system!

RDP to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 2 Download additional_samples.zip from this module's resources (available at the upper right corner) and transfer the .zip file to this section's target. Unzip additional_samples.zip (password: infected) and use IDA to analyze orange.exe. Enter the registry key that it modifies for persistence as your answer. Answer format: SOFTWARE\ \ \ \

Software\Microsoft\Windows\CurrentVersion\Run

Submit

RDP to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 2 Download additional_samples.zip from this module's resources (available at the upper right corner) and transfer the .zip file to this section's target. Unzip additional_samples.zip (password: infected) and use IDA to analyze orange.exe. Enter the name of the function that is holding the name of the file intrenat.exe that orange.exe drops as your answer. Answer format: sub_4XXXX3

sub_40A7A3

Submit

Previous Next

Mark Complete & Next

Powered by HACKTHEBOX