Malware Forensics

DISASSEMBLY

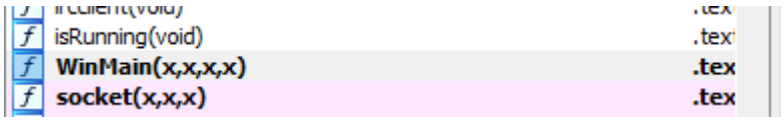

Figure 1: WinMain function

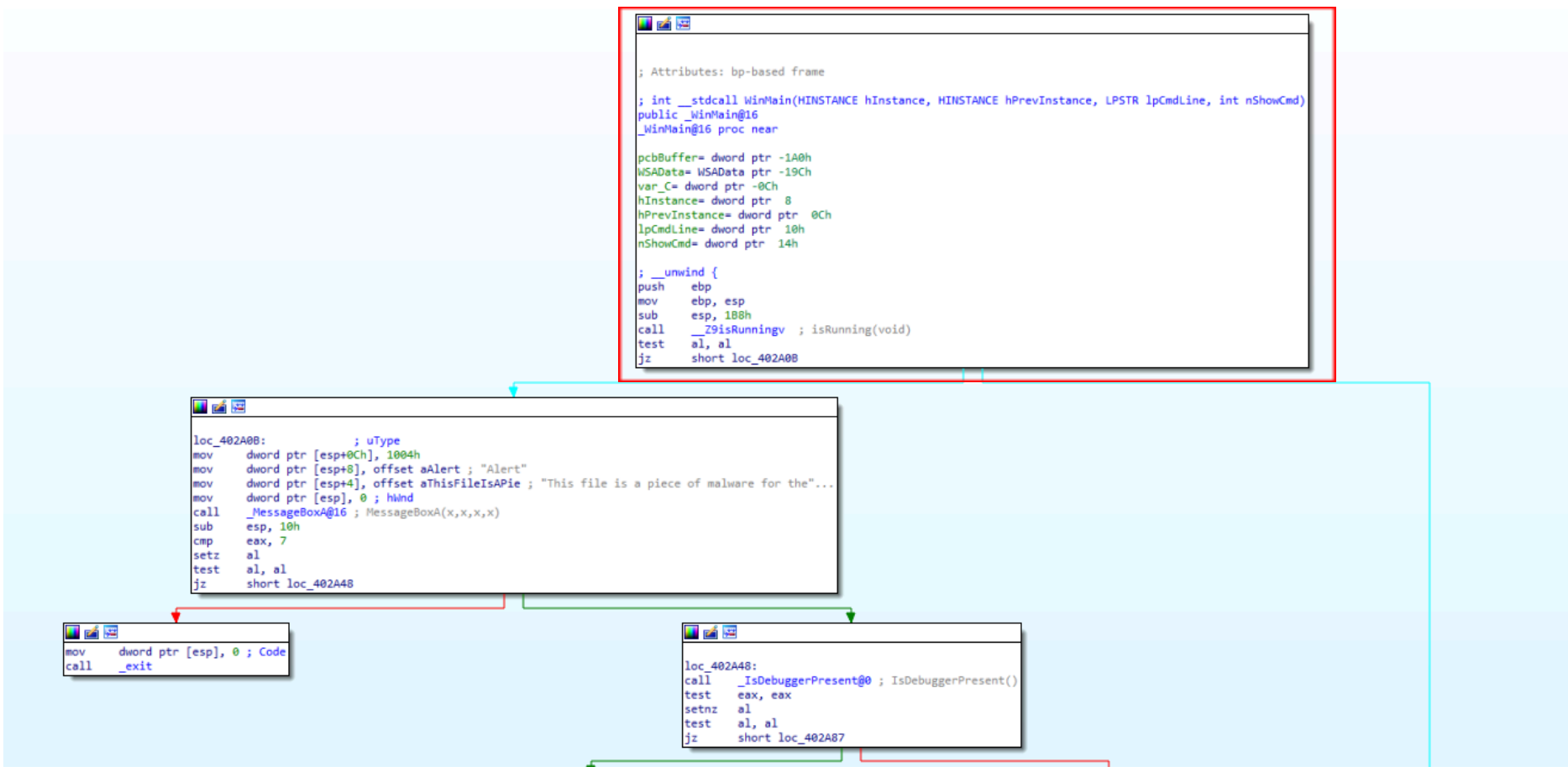

Figure 2: Inside **Winmain** function

Red: This box checks the argument that the .exe file has been double clicked, leading to the **__Z9isRunningv** Function.

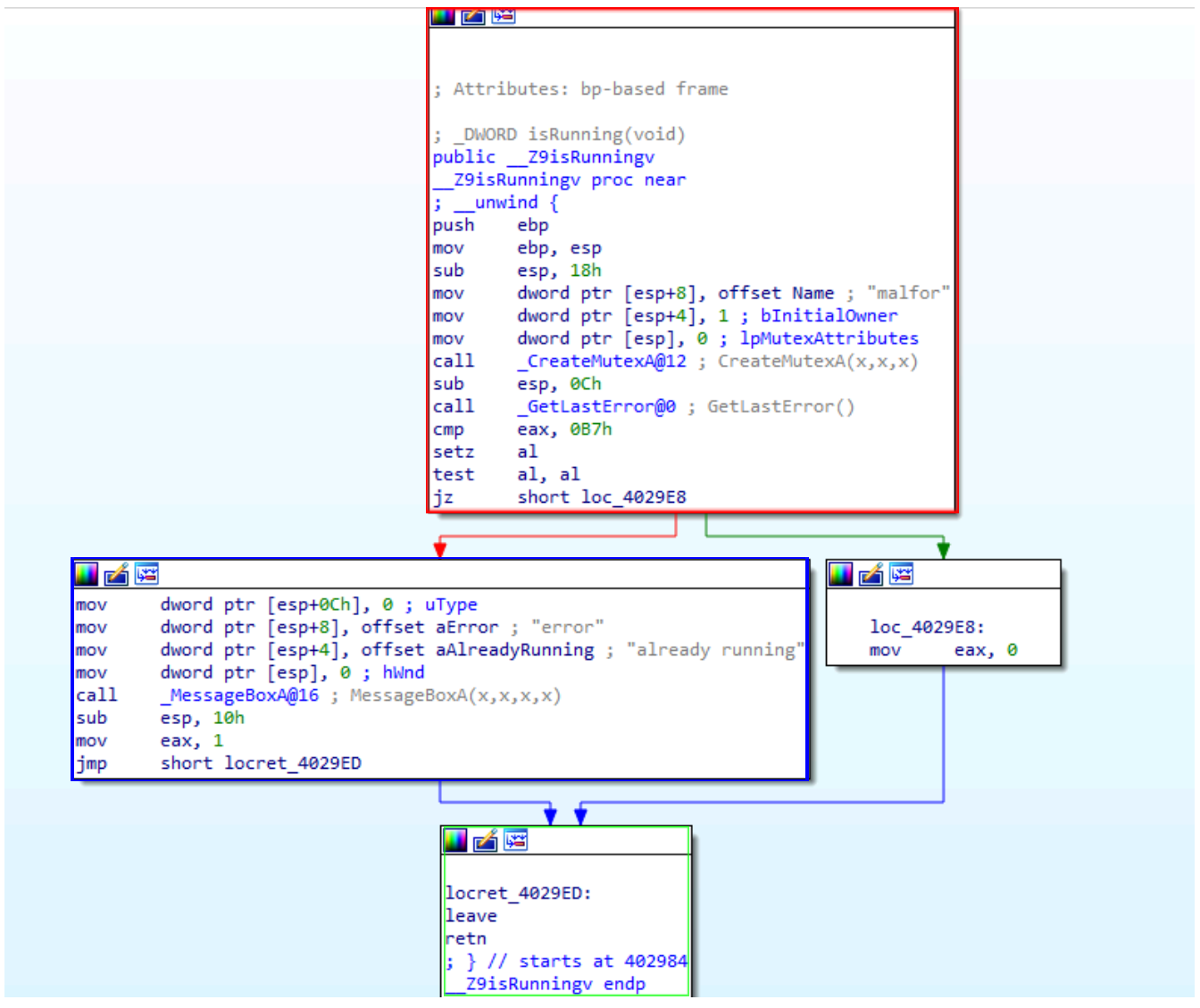

Figure 2.1: inside **__Z9isRunningv** function

Red: an offsetname is referenced as "malfor". I assume this is the name of the mutex being stored as [esp+8]. **CreateMutexA(x,x,x)** is referenced in this box. This API function is used for creating named or unnamed mutex objects; also, **GetLastError()** is referenced to get the calling threats' last error code value, which could be referenced if the malware is already running on the system.
Blue: This box is for the **call _GetLastError@0.** Further along with this, there is an if statement; if the error is that it is running, it goes to this box and presents the message box saying it is already running and allows the user to click okay.
Green: This represents the end of this function.

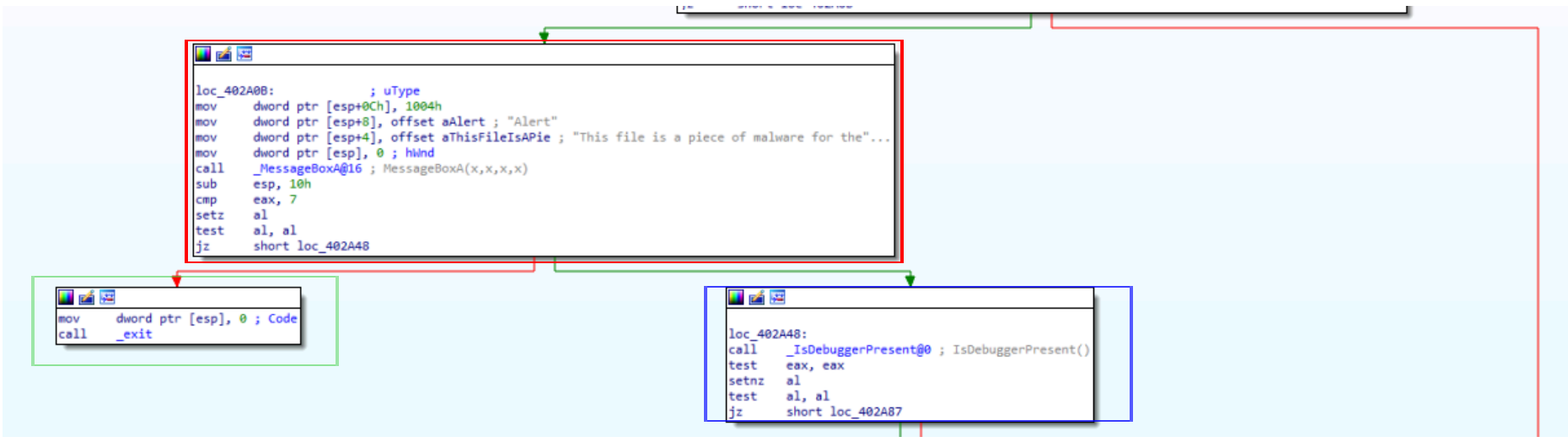

Figure 3: Alert debugger

Red: After the __Z9isRunningv there is a jz operand. This alerts the user, "This file is a piece of malware for the University malware analysis", as a message box. There is then another operand. One carries on with the infection and the other checks for debuggers. Blue: This is the beginning of the function that checks if a debugger is present, springing off the if statement. The green section represents what happens when the user clicks no since this message box allows the user to run it.

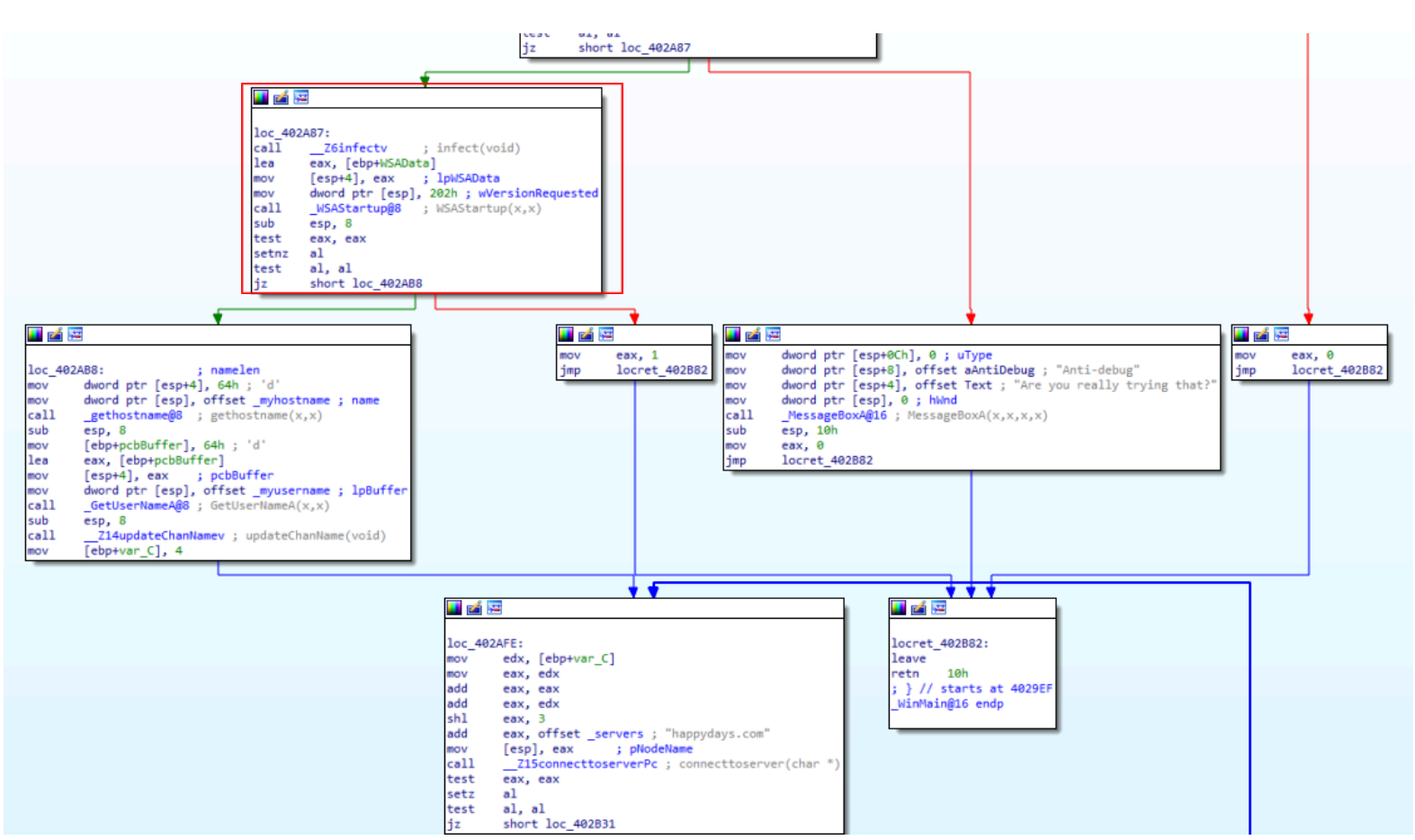

Figure 3.1: Infect (anti debugger no)

Red: If there is no debugger present it goes to this box, this segment starts the infection of the system through **__z6infectv** function

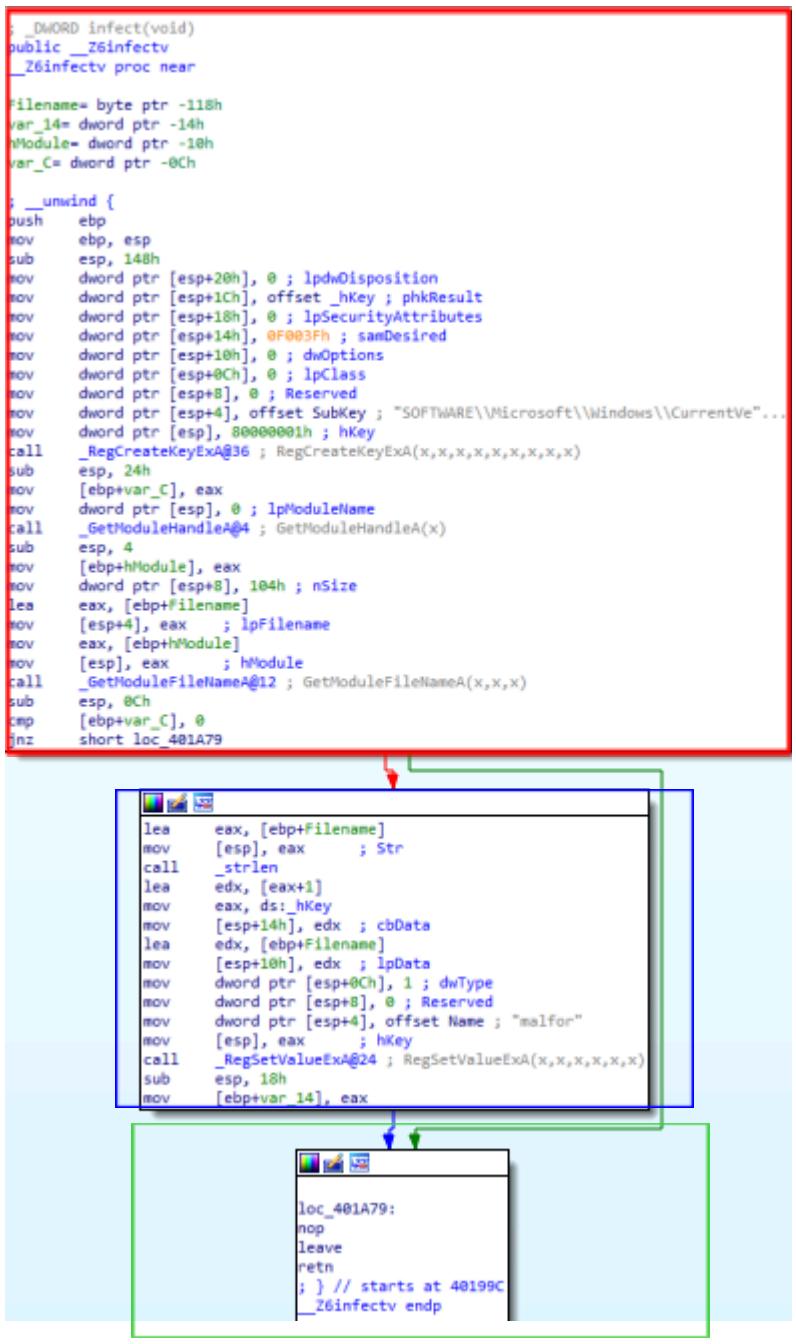

Figure 3.2: Inside **__z6infectv**

Red: This section is where persistence for the malware is created; it looks for the section in the “Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run” where it leaves a persistent so that when the computer starts, it automatically starts the program. This part also references the **GetModuleHandleA(x)** grabs a handle “malfor” shown in the blue: section, which is then saved the registry **RegSetValueExA(x,x,x,x,x,x)** implying its set value. The green box represents its end point to which it goes back to the function before.

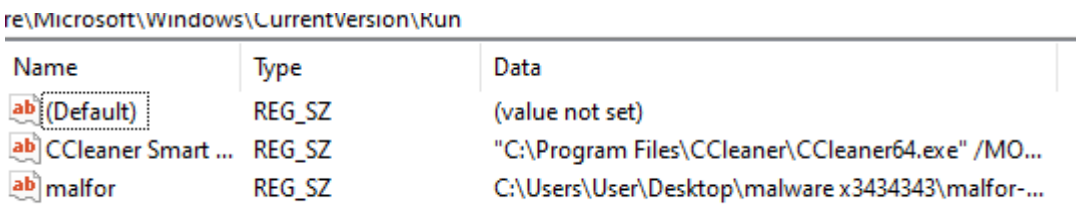

Figure 3.3: The registry

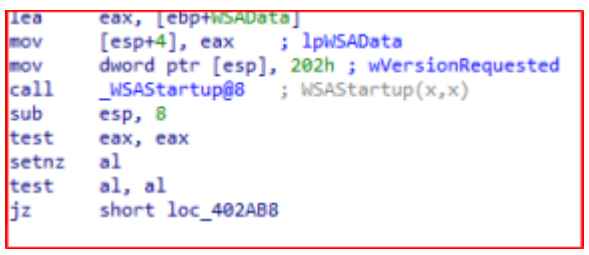

Figure 3.4: **WSAStartup(x,x)**

Red: This starts the Winsock dll process referencing its two parameters that are not specified. If this call is successful **al** should be set to 1 if not it's set to 0 indicating a failure.

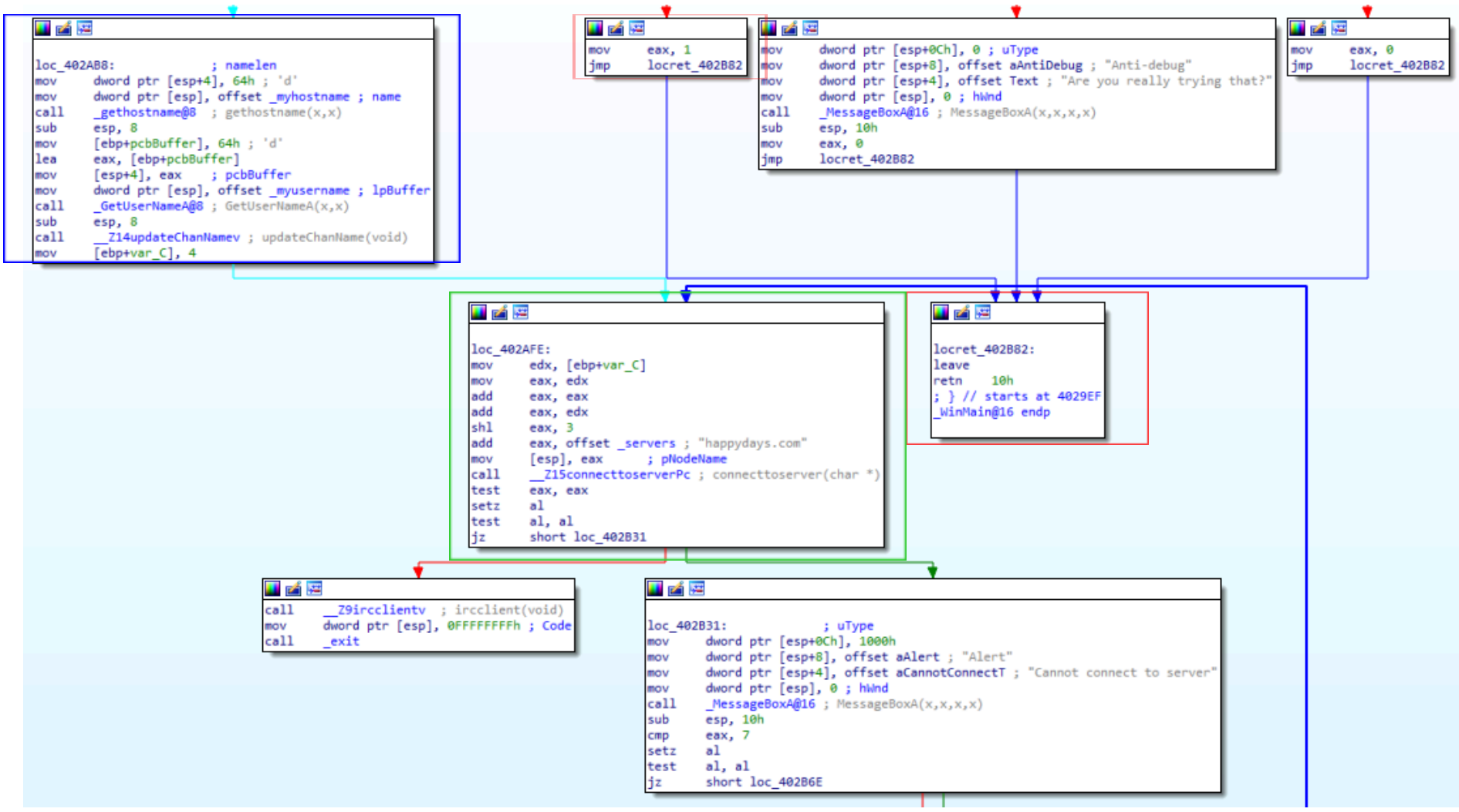

Figure 3.5: **WSAStartup** failure and success

Red: If al for WSAStartup is 0, it returns to the beginning of _winmain@16 = 4029EF, retrying this connection process.
The Blue boxes represent what happens when the al for WSAStartup is 1; it goes down a route that focuses on connecting this device to the botnet. gethostname(x,x) retrieves the hostname for the local machine, and I assume the device it's trying to connect to. From what I can find, GetUserNameA(x,x) gets the user a username. It has it automatically set as _nick referenced in the _myusername. After this, it goes into the __z14updateChanNamev.

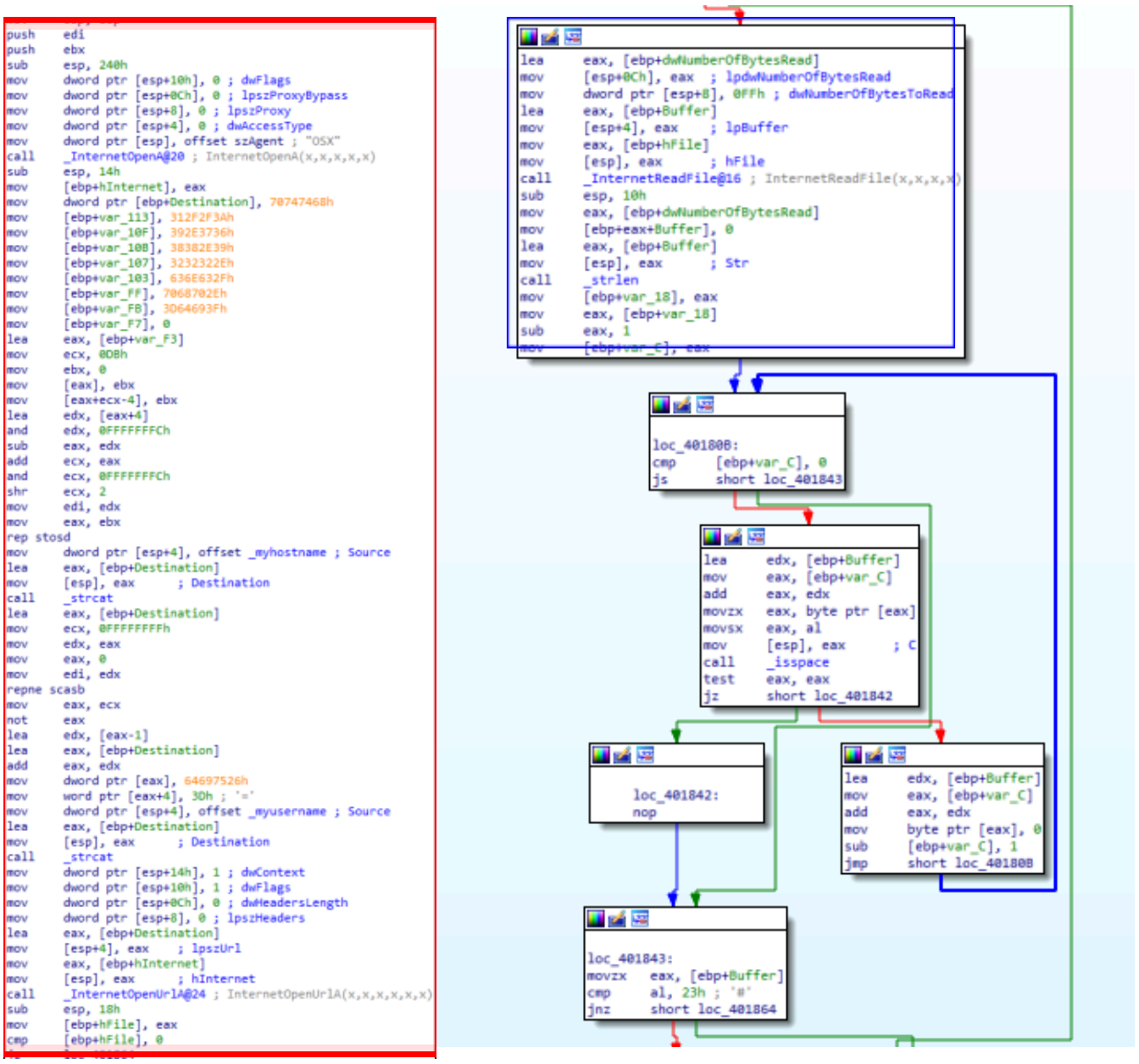

Figure 3.6: **__z14updateChanNamev** function

Red: This section of the InternetOpenA(x,x,x,x) function starts a session with the WinINet functions. In this part, OSX is one of the parameters that can be referenced for this function. The specifications of the destinations are shown with the vars and their specific values. This function slowly generates a URL and then opens it with the InternetOpenUrlA(x,x,x,x) with its represented parameters.
Blue: This represents the InternetReadFile(x,x,x,x), which is the section that reads a file following the buffer represented for its parameter, which then reads a file and follows the path.

Figure 3.7: InternetCloseHandle

Red: This part references a “#” from what I know about this malware; this relates to the next part, the **__Z9ircclientv** for the chat channels.
Blue: This section has a tag that is used for closing the **__z14updateChanNamev** function using the **InternetCloseHandle(x)**, which is used to close internet file handlers. Then, move on to the next segment of the overall function

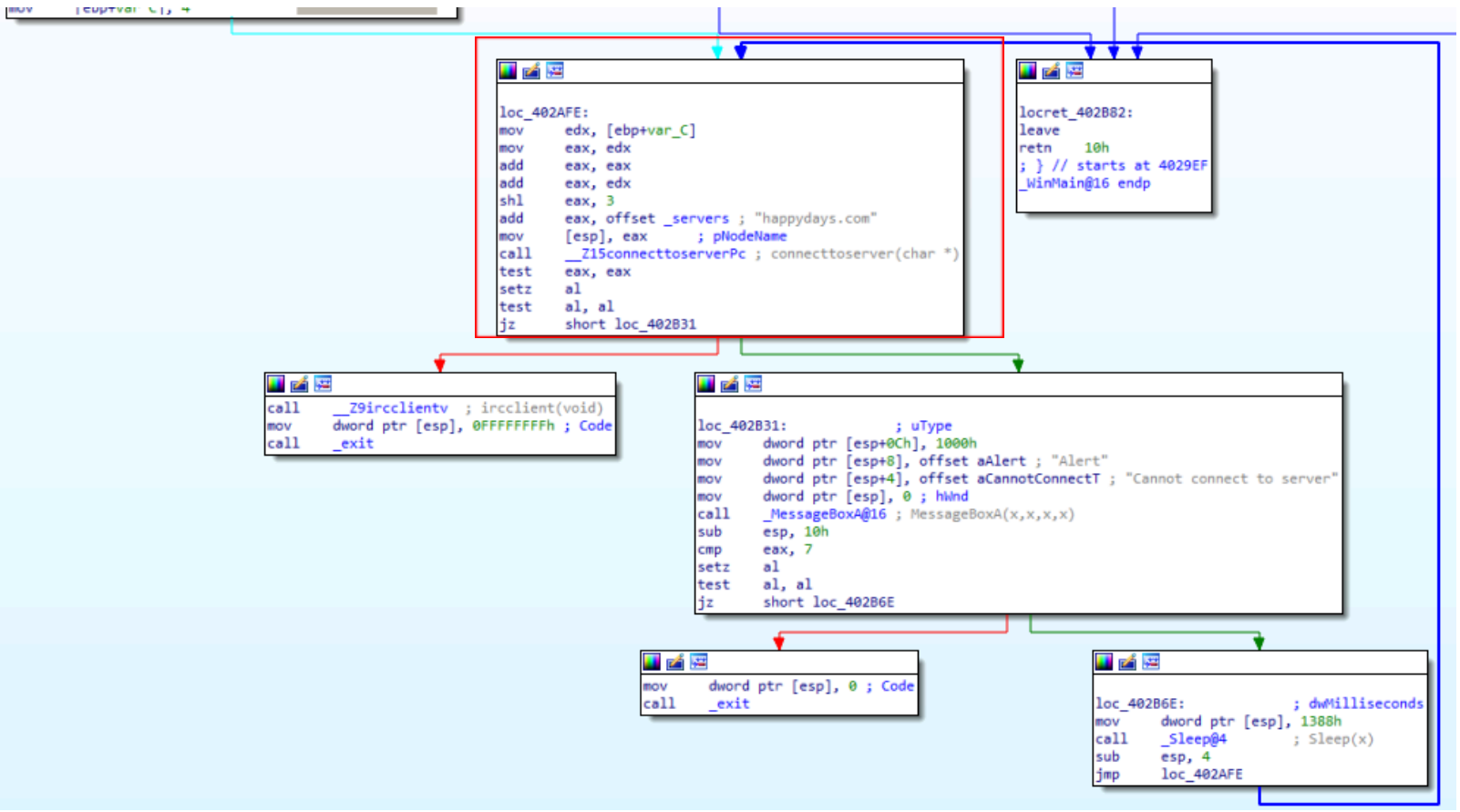

Figure 3.8: server connect functions

Red: It represents the connection to the host; I can see the name of the host's server, "happydays.com." This is what is running the websites that connect to the IRC client.

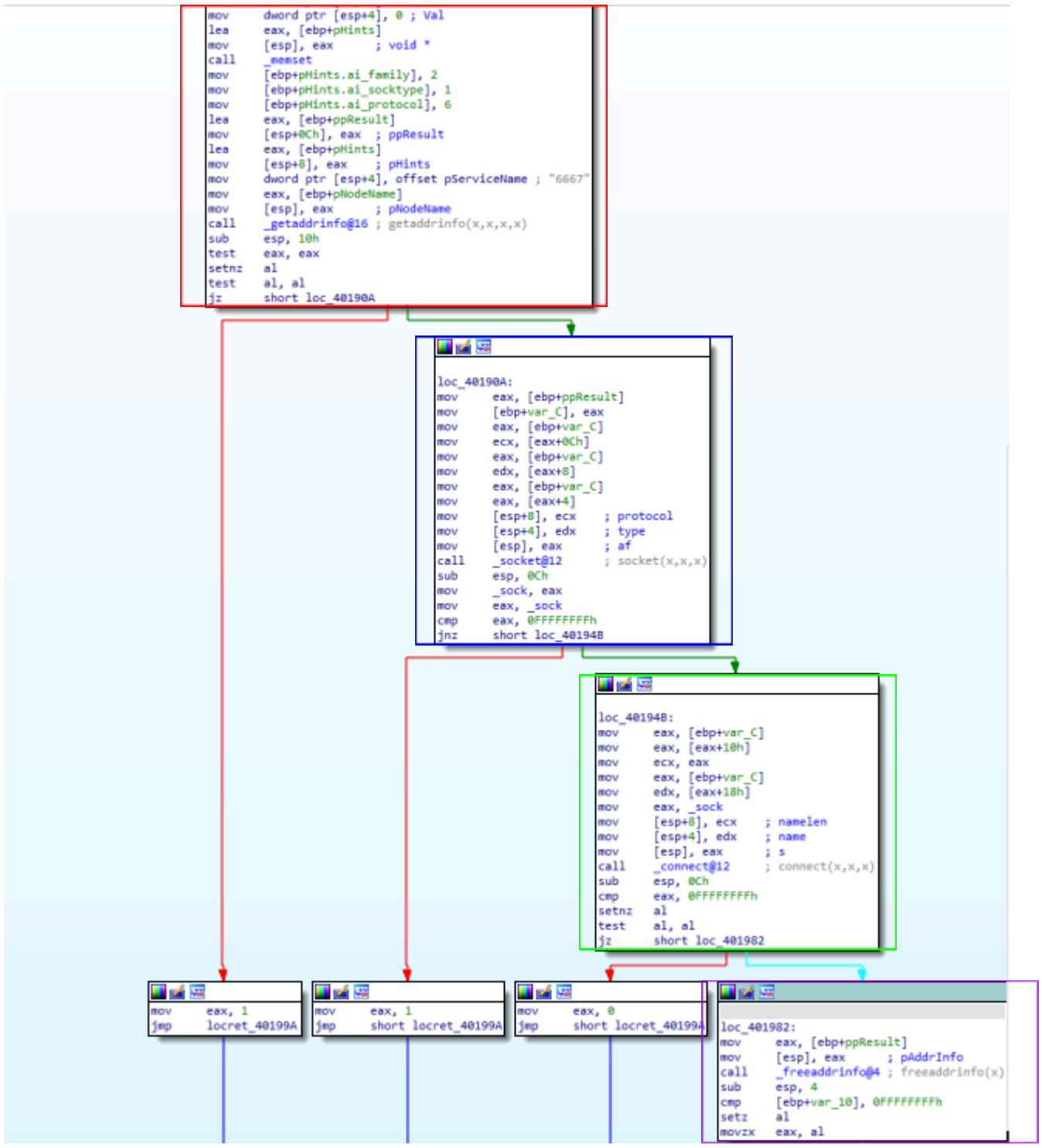

Figure 3.9: **__Z15connecttoserverPC** function

Red: This shows the port number that the IRC is running off. **pServiceName** is that specific port “6667”; these are TCP/UDP ports known for IRC traffic. **getaddrinfo(x, x,x,x,x)** I can tell this is getting the address information of the IR client, possibly its IP address.
Blue: Sockets are being created in this block. The parameters for the sockets are gained from the ppResult structure in red. If successful, it will proceed to the next loc.
Green: This section tries connecting the remote host and the user using this connect function.
Purple: freeaddrinfo(x) is the freeing of memory for the address information. After this, it then carries on to the next segment.

Three boxes run off these functions; if one fails, it will end the attempt to connect and leave the function, probably leading to a restart or an end to the attempts.

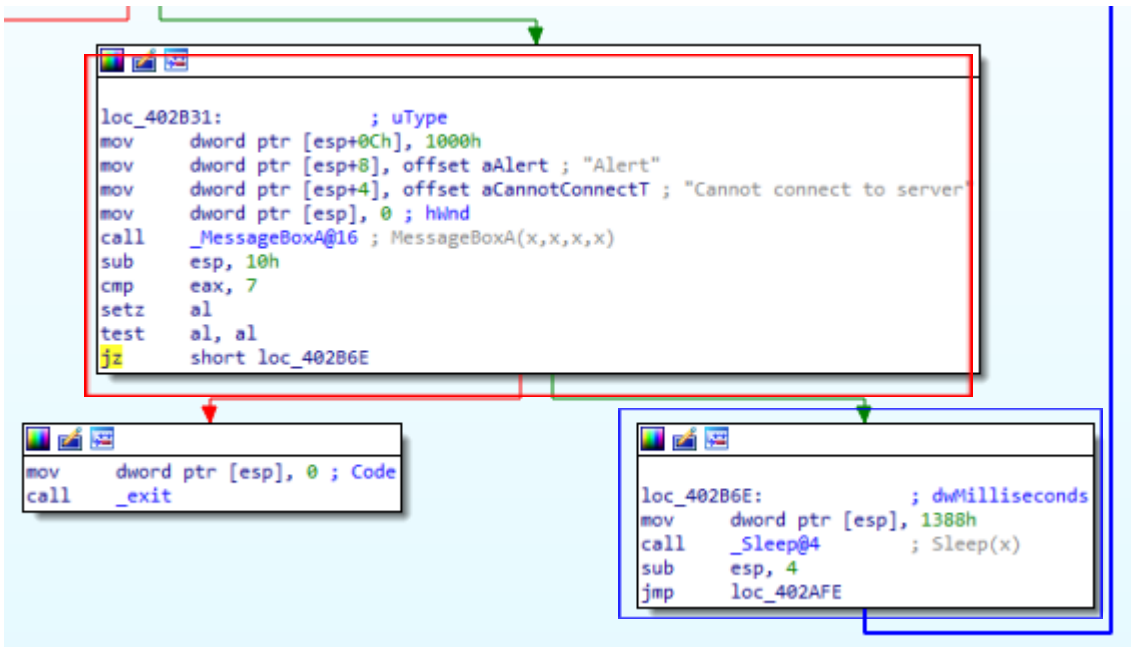

Figure 3.10: cannot connect to server

Red: This section presents an alert box for when the user cannot connect to the server; there is an if statement with two options.
Blue: This box is the response from when the user clicks okay to the cannot connect to server prompt. It will sleep for a couple ms then return to the beginning of the code

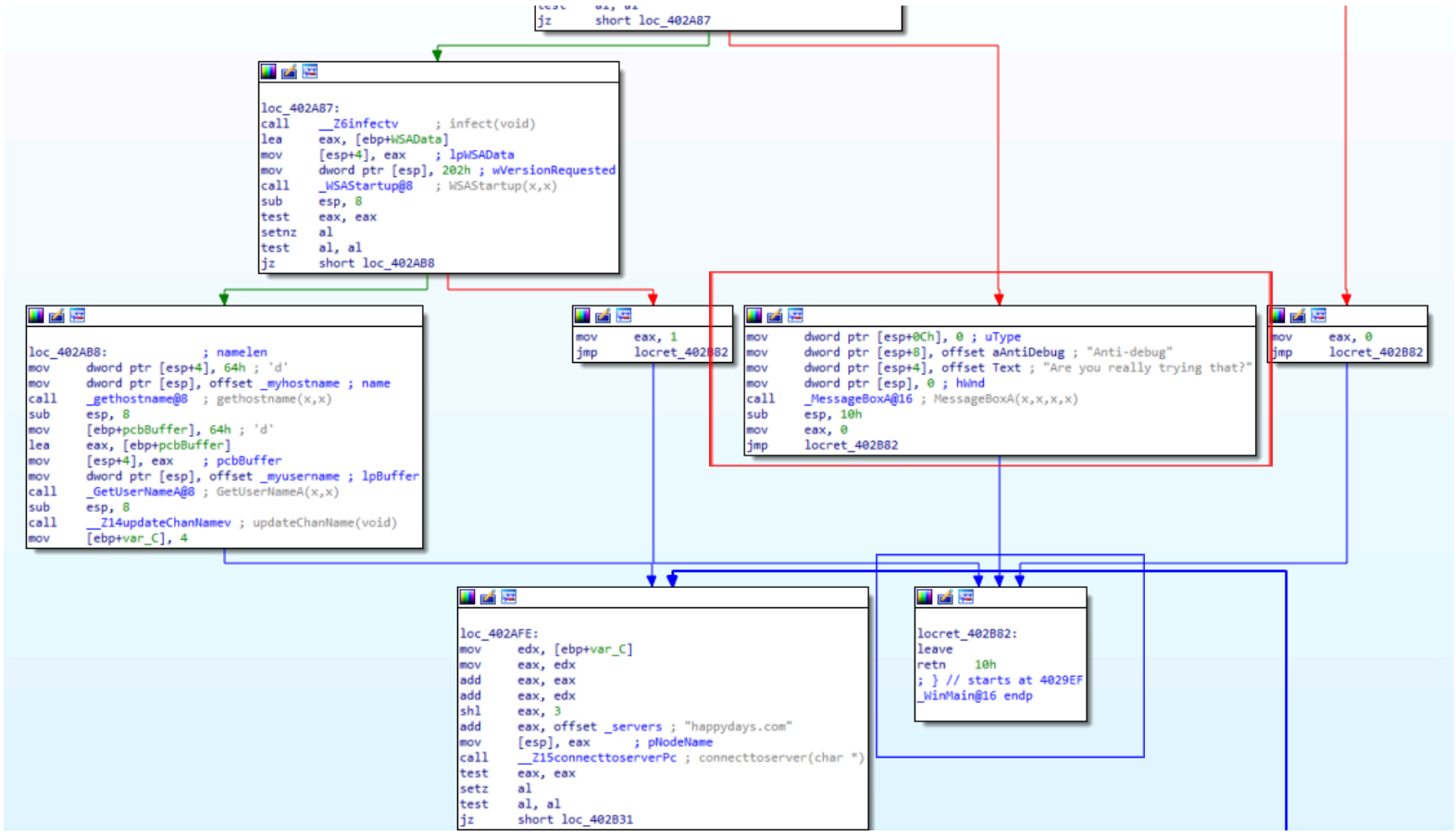

Figure 3.11: Anti-debug (yes)

Red: If an anti-debug is present, the code recognizes a debugger; it will show a message box titled “Anti-debug” with the text “Are you trying that?”

Blue: This set of functions goes and takes the user back to the beginning of the program to make sure a debugger is not present

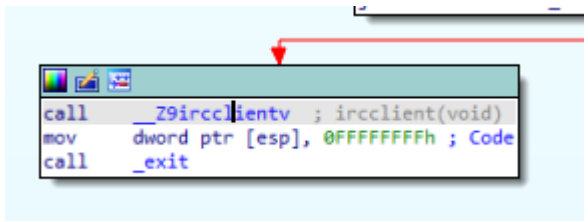

Figure 4: **__Z9ircclientv** function

This function relates to the use of an IRC client.

```
var_18= dword ptr -18h
var_14= dword ptr -14h
Str1= dword ptr -10h
Source= dword ptr -0Ch

; __unwind {
push    ebp
mov     ebp, esp
mov     eax, 1468h
call    ___chkstk_ms
sub     esp, eax
mov     dword ptr [esp], 0 ; Time
call    _time
mov     [esp], eax      ; Seed
call    _srand
call    _rand
mov     [esp+0Ch], eax
mov     dword ptr [esp+8], offset aBot ; "bot"
mov     dword ptr [esp+4], offset aSD ; "%s%d"
mov     dword ptr [esp], offset _nick ; Buffer
call    _sprintf
mov     ds:byte_409128, 0
lea     eax, [ebp+Buffer]
mov     dword ptr [eax], 53534150h
mov     dword ptr [eax+4], 6E616620h
mov     dword ptr [eax+8], 65627963h
mov     dword ptr [eax+0Ch], 0A0D7261h
mov     byte ptr [eax+10h], 0
mov     eax, _sock
lea     edx, [ebp+Buffer]
mov     [esp+4], edx    ; Str
mov     [esp], eax      ; s
call    __Z9writeLinejPc ; writeLine(uint,char *)
mov     dword ptr [esp+0Ch], offset aUnivCoursework ; "Univ Coursework Exercise"
mov     dword ptr [esp+8], offset _nick
mov     dword ptr [esp+4], offset aUserS0S ; "USER %s 0 * :%s\r\n"
lea     eax, [ebp+Buffer]
mov     [esp], eax      ; Buffer
call    _sprintf
mov     eax, _sock
lea     edx, [ebp+Buffer]
mov     [esp+4], edx    ; Str
mov     [esp], eax      ; s
call    __Z9writeLinejPc ; writeLine(uint,char *)
mov     dword ptr [esp+8], offset _nick
mov     dword ptr [esp+4], offset aNickS ; "NICK %s\r\n"
lea     eax, [ebp+Buffer]
mov     [esp], eax      ; Buffer
call    _sprintf
mov     eax, _sock
lea     edx, [ebp+Buffer]
mov     [esp+4], edx    ; Str
mov     [esp], eax      ; s
call    __Z9writeLinejPc ; writeLine(uint,char *)
call    _IsDebuggerPresent@0 ; IsDebuggerPresent()
test    eax, eax
setnz   al
test    al, al
jz      short loc_402098
```

Figure 4.1: inside beginning of the **__Z9ircclientv** function

This whale code block implements an internet relay chat (IRC) client. The section in red has the IRC generate a random nickname using the **_strand** and **_rand** functions, then use the "bot" with a randomly generated number stored in the **_nick** buffer. The section in blue initiates a buffer with IRC commands. The green section writes the buffer's content to the IRC server using the **__Z9writelineJPc** function.
The purple section checks for a debugger shown with the if statement.

```
; Attributes: bp-based frame

; _DWORD __cdecl writeLine(SOCKET s, char *Str)
public __Z9writeLinejPc
__Z9writeLinejPc proc near

s= dword ptr  8
Str= dword ptr  0Ch

; __unwind {
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+Str]
mov     [esp], eax      ; Str
call    _strlen
mov     dword ptr [esp+0Ch], 0 ; flags
mov     [esp+8], eax    ; len
mov     eax, [ebp+Str]
mov     [esp+4], eax    ; buf
mov     eax, [ebp+s]
mov     [esp], eax      ; s
call    _send@16        ; send(x,x,x,x)
sub     esp, 10h
leave
retn
; } // starts at 401460
__Z9writeLinejPc endp
```

Figure 4.2: **__Z9writelineJPc** function

This function is designed to send strings to specified sockets using the send function. It calculates the string length with the correct parameters and then calls the send function.

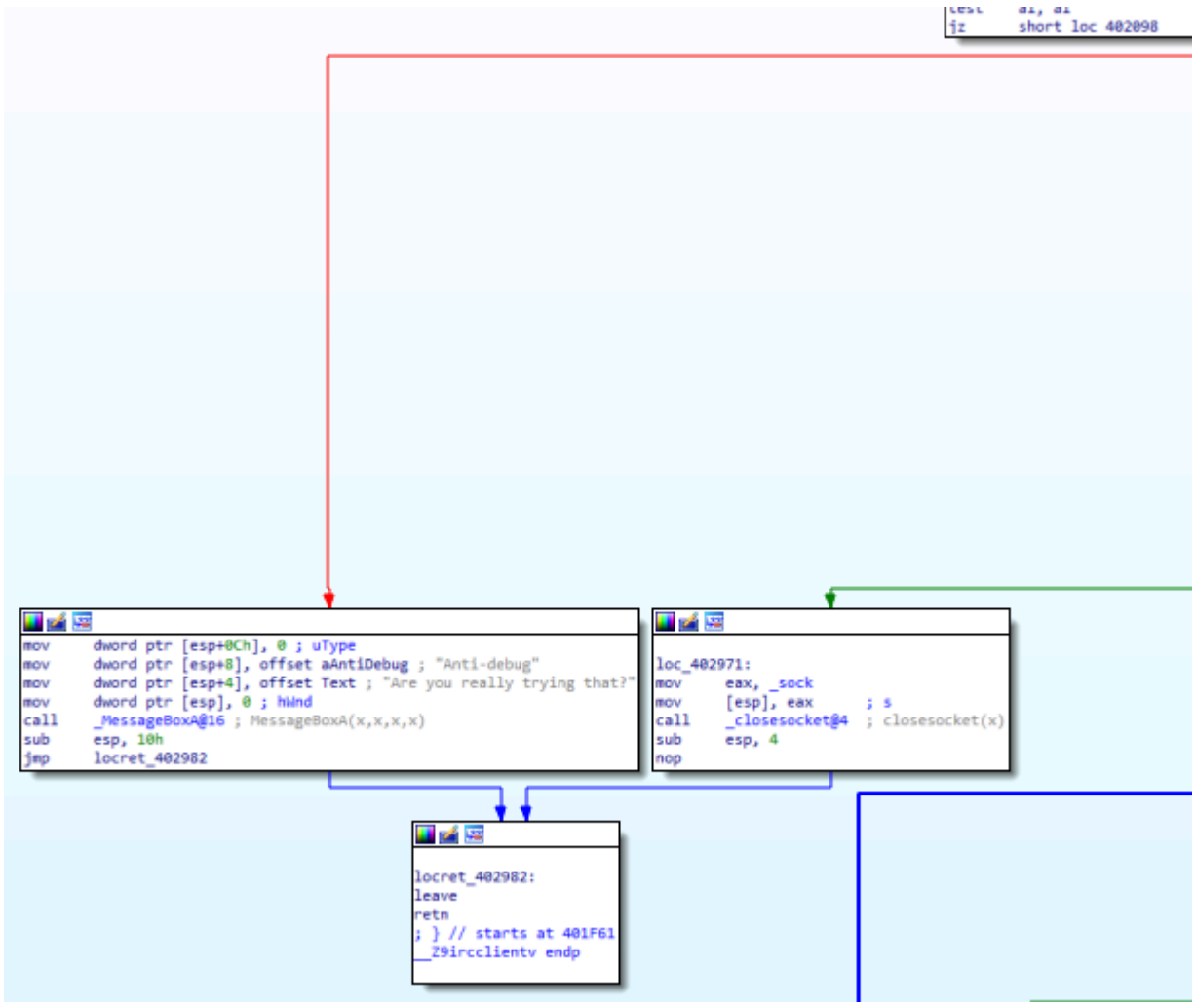

Figure 4.3: Anti-debug and close socket

This section shows another anti-debug for if you are able to bypass the first one, this one is used to make sure you can't make it past the second one for the irc client

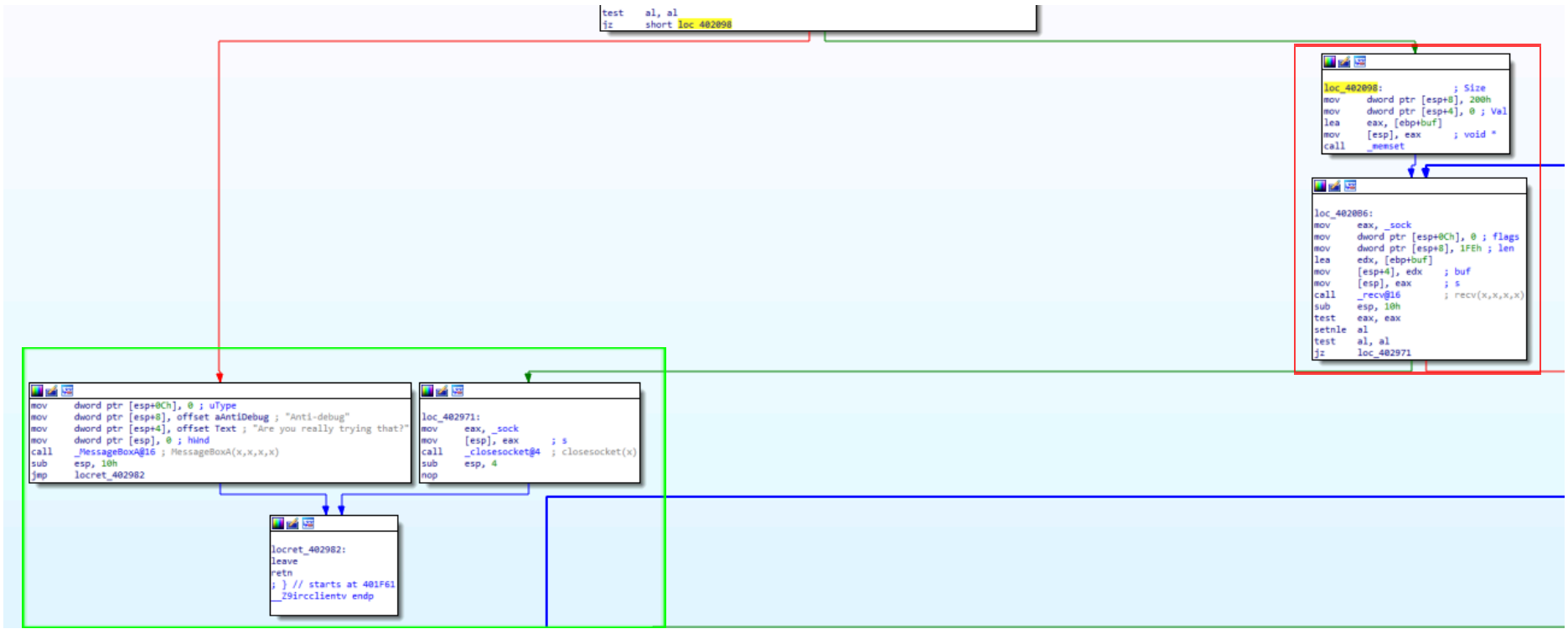

Figure 4.4: socket recv

Red: This code section receives data from a socket and stores it in a buffer before checking whether the "recv" operation was successful. The result determines the behaviour following the accepted process, and the programme will take various actions based on the conditions tested.
Green: This code segment is again the anti-debugger blocker. If you want a more detailed description, go to the previous one.

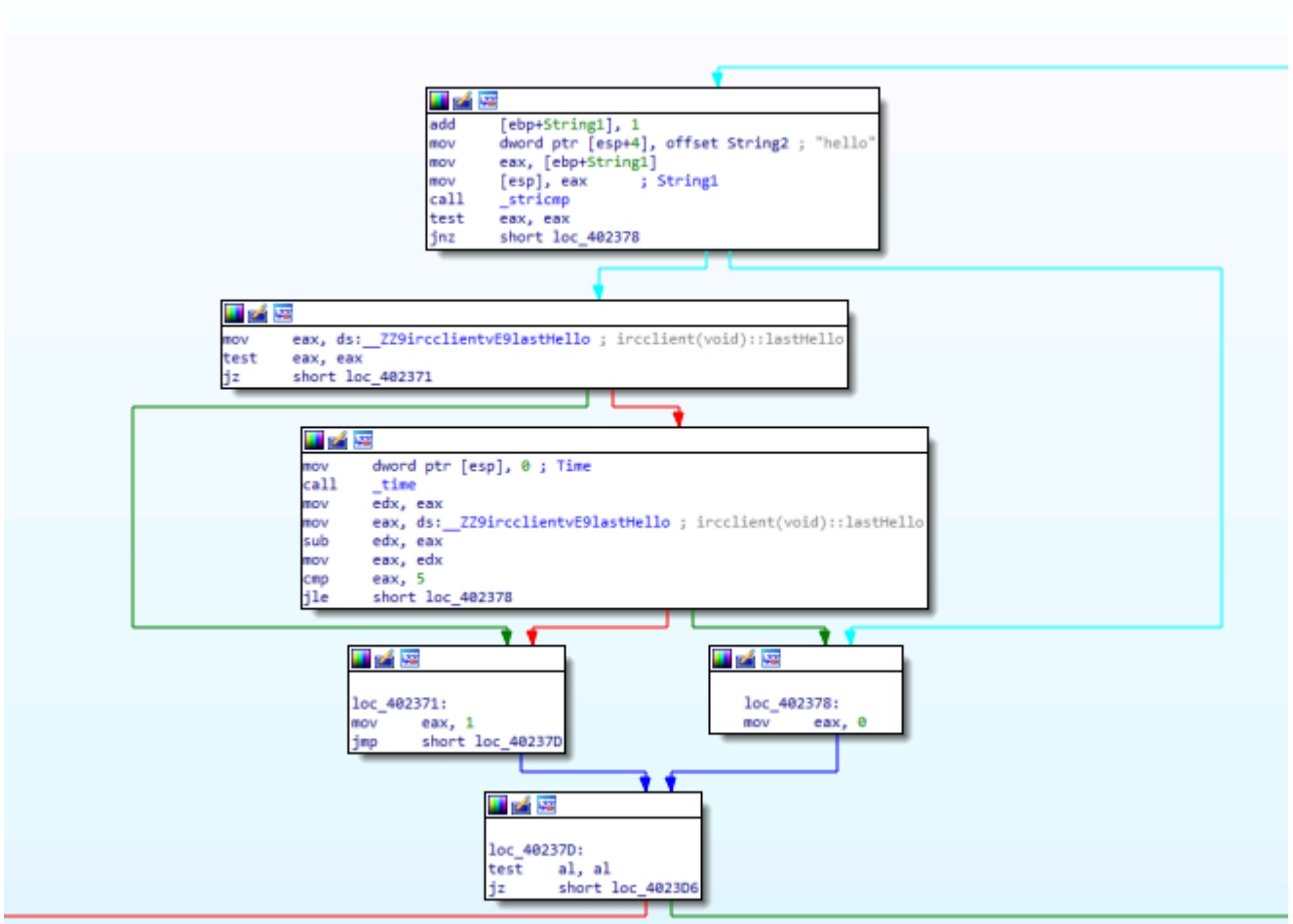

Figure 4.5: Hello

The “hello” command is for the IRC client when ran it prints a private message to the user shown in (figure 4.5.1)

Figure 4.5.1:

This code segment shows the response you get when you type hello in the irc client. The full response is “PRIVMSG %s :Hi %s, I',27h,'m a bot that', 27h,'s part of the UoP MALFOR Coursework.”

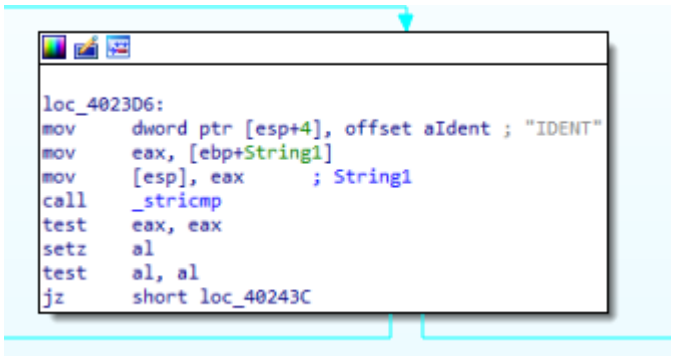

Figure 4.6: IDENT

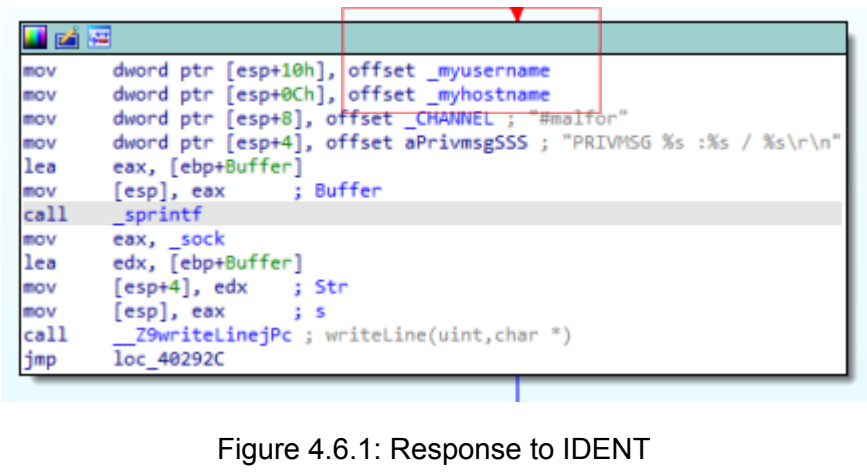

Figure 4.6.1: Response to IDENT

The IDENT command “identify”, this will scan the user system and check for the name of there machine and the users account name shown in red by the 2 calls of **_myusername** and **_myhostname**

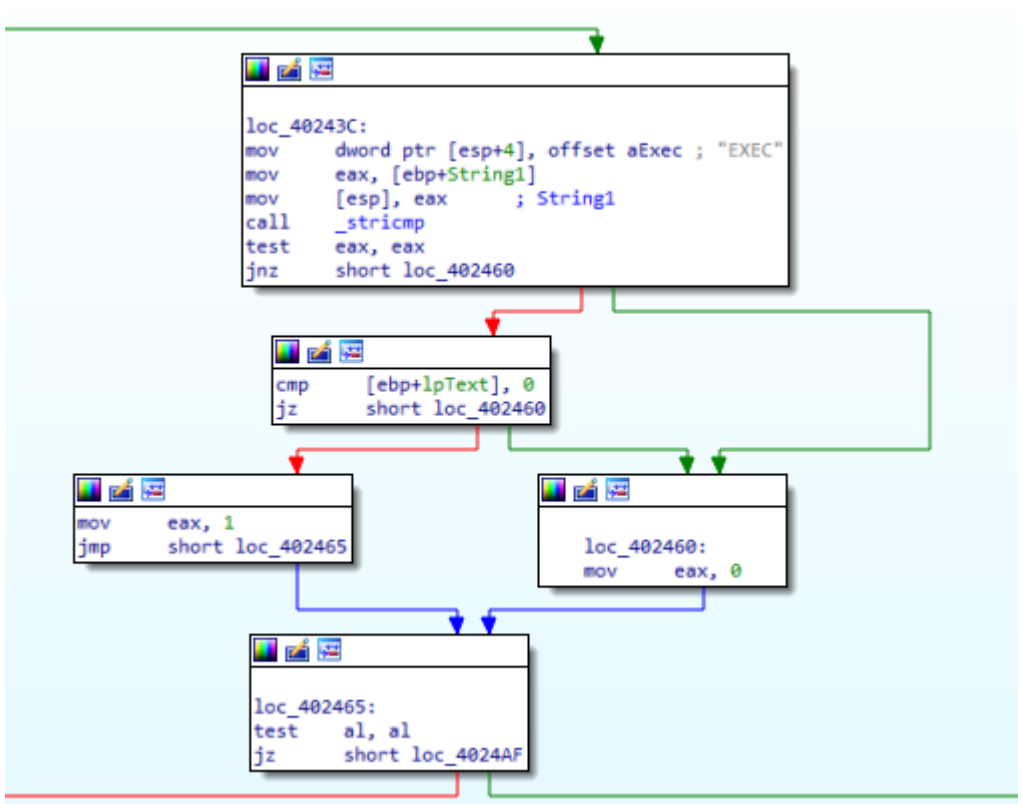

Figure 4.7: EXEC command

The EXEC command means execute, In figure 4.6.1 you can see a box in red Showing a “#”, this is because when using the command you need to put a hashtag before any executable. Something like “exec #notepad.exe” will open the notepad program.

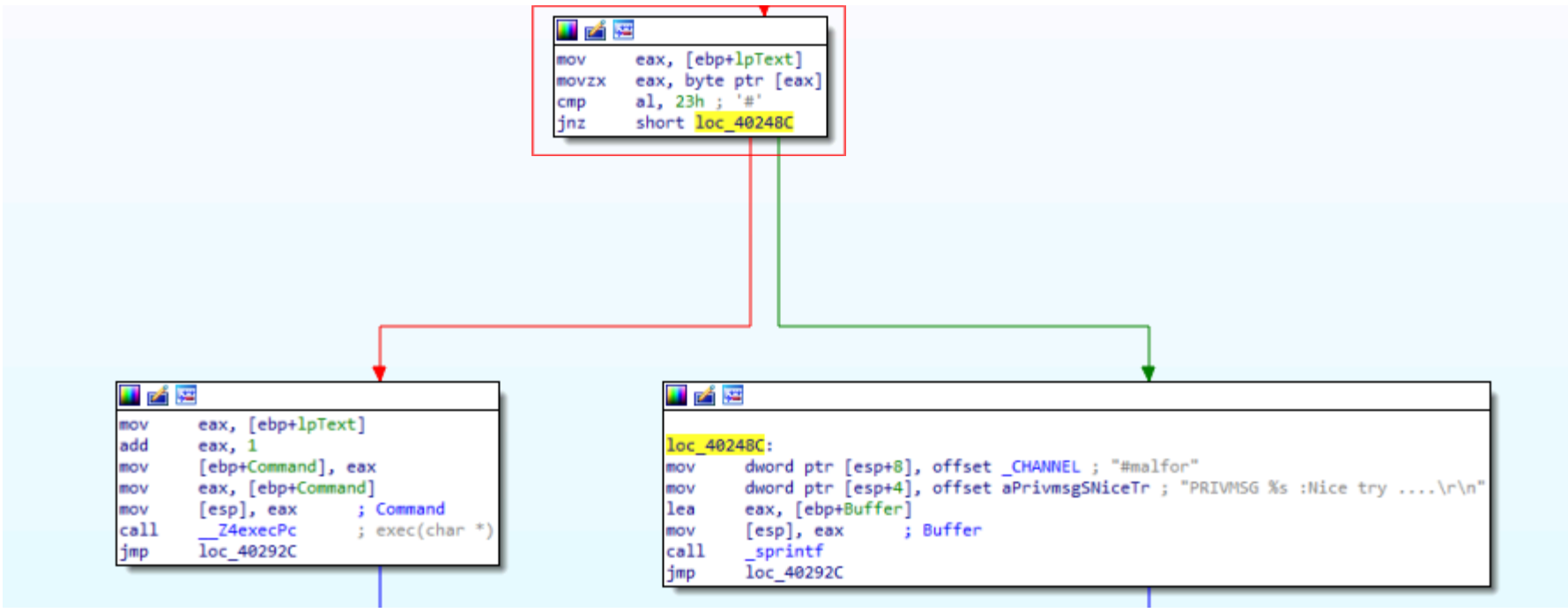

Figure 4.6.1: Response from EXEC

This box is explain in the explanation above

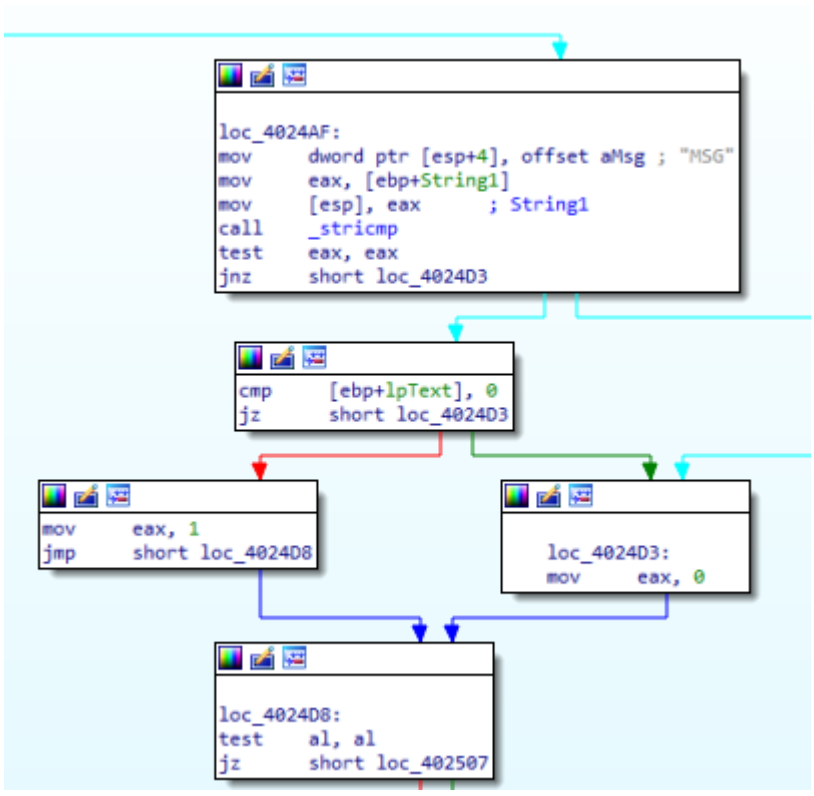

Figure 4.7: MSG command

This command is a system message for windows, it allows an user to write whatever message they want to send to a bot and have it appear on the bots screen.

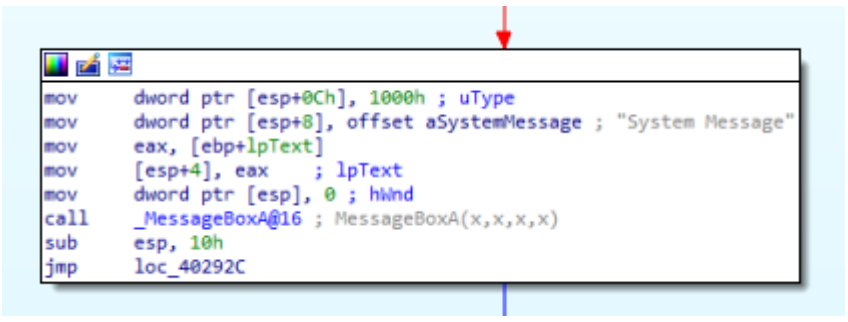

Figure 4.7.1: Exit code for MSG

This shows how the MSG command displays a message box, this box will be called “system message” with the given parameters given with the msg command.

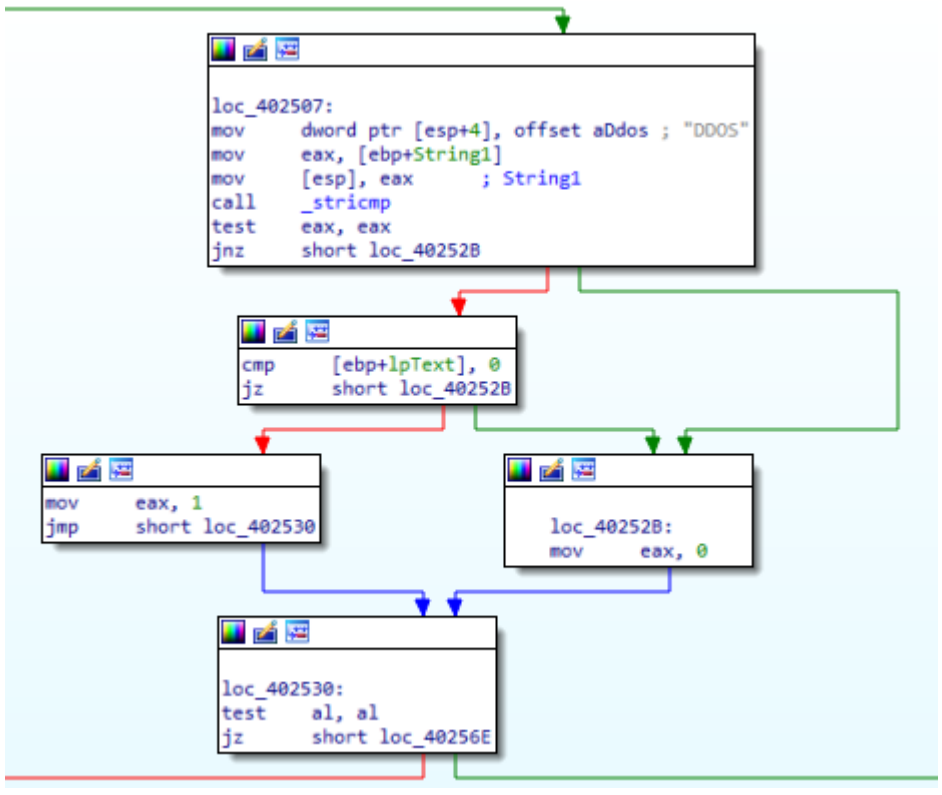

Figure 4.8: DDOS command

With botnets they can do DDoS attacks since you can use them to spam a server to create massive loads so that it slows the server and can bring down networks. With this malware you can see there is an option for the potential type of attack.

Figure 4.8.1: DDOS command response

Although this is a command that is on display as being usable, it seems that the command is not actually usable since the creator of this malware has removed the functionality for it to be a thing.

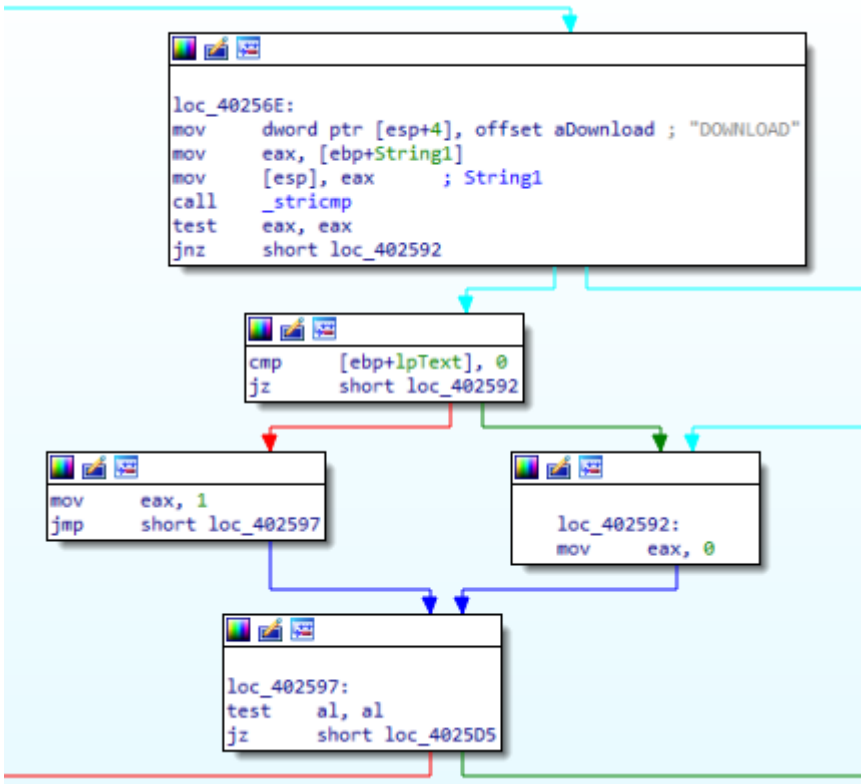

Figure 4.9: DOWNLOAD command

The download command seems to be able to download a requested item.

```
mov     dword ptr [esp+8], offset _CHANNEL ; "#malfor"
mov     dword ptr [esp+4], offset aPrivmsgSReally ; "PRIVMSG %s :Really? ....\r\n"
lea     eax, [ebp+Buffer]
mov     [esp], eax      ; Buffer
call    _sprintf
mov     eax, _sock
lea     edx, [ebp+Buffer]
mov     [esp+4], edx    ; Str
mov     [esp], eax      ; s
call    __Z9writeLinejPc ; writeLine(uint,char *)
jmp     loc_40292C
```

Figure 4.9.1: response for DOWNLOAD

If the user tries to download something it can give a response for that being: “PRIVMSG %s :Really? ....”

```
loc_4025D5:
mov     dword ptr [esp+4], offset aPic ; "PIC"
mov     eax, [ebp+String1]
mov     [esp], eax      ; String1
call    _stricmp
test    eax, eax
setz    al
test    al, al
jz      short loc_4025FB
```

```
call    __Z17CaptureScreenshotv ; CaptureScreenshot(void)
jmp     loc_40292C
```

Figure 4.10: Pic command and response

This command captures a screenshot of the users screen and saves it into a folder on the users device, elsewhere in the assembly we can see it saves the picture as “screenshot.bmp”

```
loc_4025FB:
mov     dword ptr [esp+4], offset aOpen ; "OPEN"
mov     eax, [ebp+String1]
mov     [esp], eax      ; String1
call    _stricmp
test    eax, eax
jnz     short loc_40261F
```

```
cmp     [ebp+lpText], 0
jz      short loc_40261F
```

```
mov     eax, 1
jmp     short loc_402624
```

```
loc_40261F:
mov     eax, 0
```

```
loc_402624:
test    al, al
jz      short loc_402663
```

```
mov     dword ptr [esp+14h], 1 ; nShowCmd
mov     dword ptr [esp+10h], 0 ; lpDirectory
mov     dword ptr [esp+0Ch], 0 ; lpParameters
mov     eax, [ebp+lpText]
mov     [esp+8], eax    ; lpFile
mov     dword ptr [esp+4], offset Operation ; "open"
mov     dword ptr [esp], 0 ; hwnd
call    _ShellExecuteA@24 ; ShellExecuteA(x,x,x,x,x,x)
sub     esp, 18h
jmp     loc_40292C
```

Figure 4.11: OPEN command

This command can be use to perform on executables, it will open the executable as the commands says, in the second ss it shows how it will run a shell execute with the correct parameters e.g: open screenshot.bmp

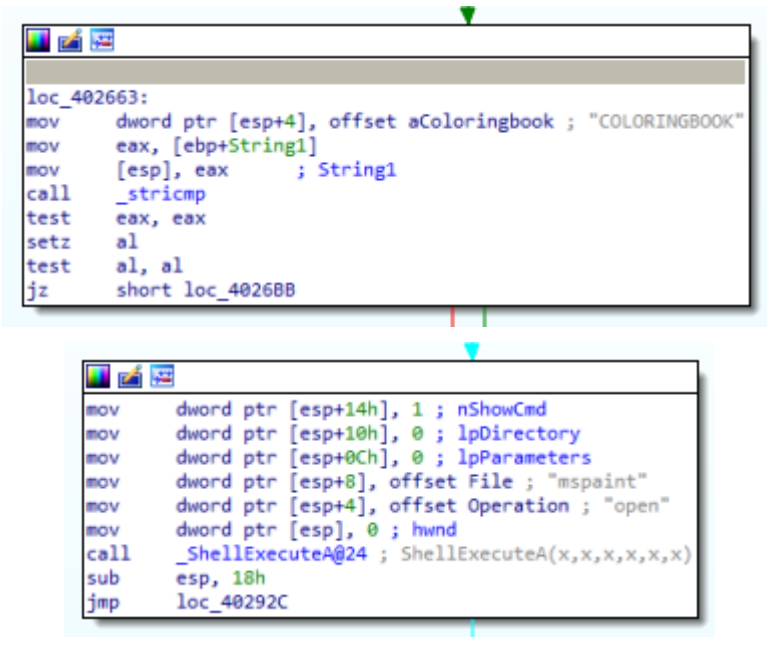

Figure 4.12: COLORINGBOOK command

In the first screenshot we can see the command coloringbook is a executable in a irc client, in the second one we can see what happens when the user runs the command it goes and opens ms paint application

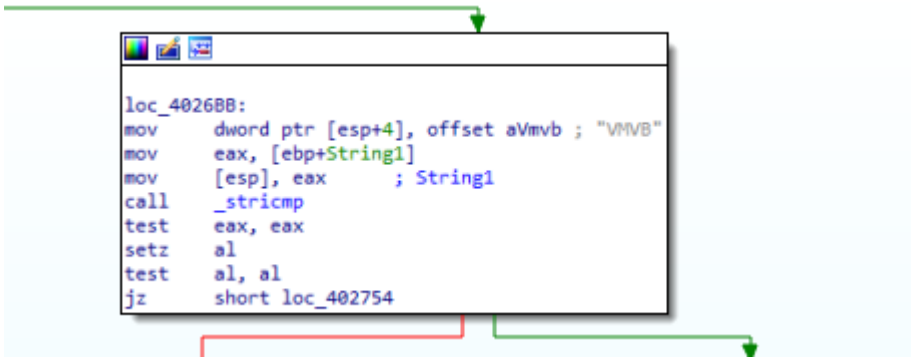

Figure 4.13: VMVB command

VMVB is a command that can be used in the irc client, In figure 4.13.1 we can see the result of the command when a user runs it

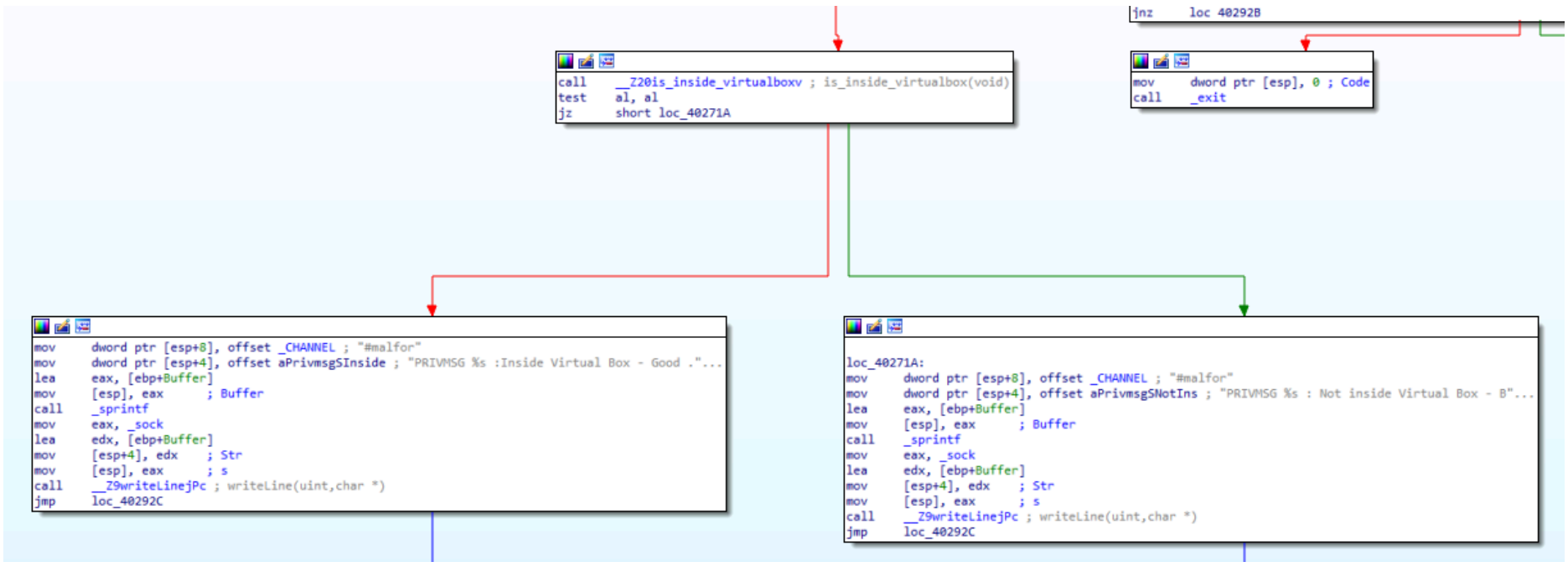

Figure 4.13.1: if statement for the command

The malware checks if the user running the command is in a virtual machine by using the **__Z20is_inside_virtualboxv** function, which goes and runs something to see if it is in a virtual box environment. After it decides if the malware is in a virtual box, it says, “inside virtualbox - good”. If not it says, “not inside virtual box - bad”

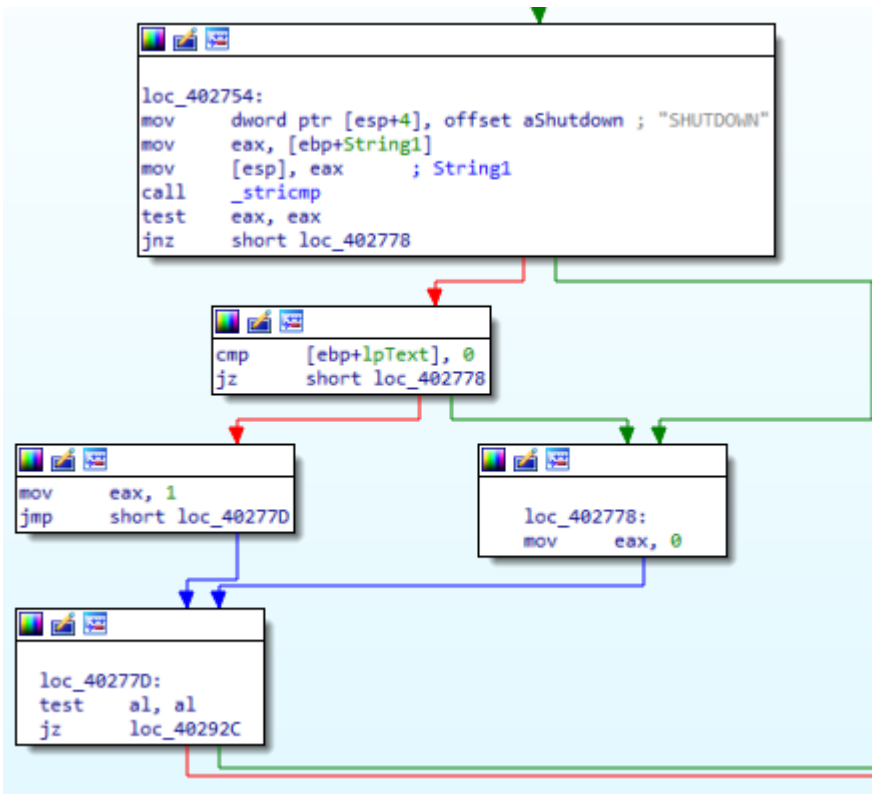

Figure 4.14: SHUTDOWN command

This command has a lot to do with it, this command from what we can see is a shutdown command for the botnet server.

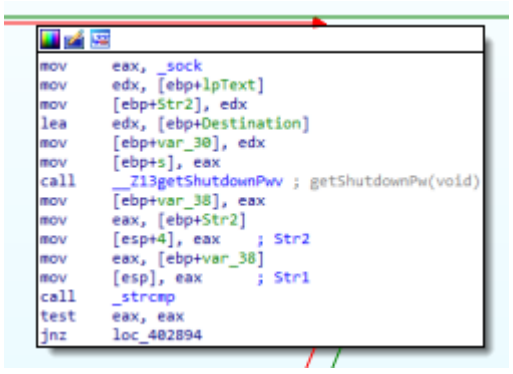

Figure 4.15: SHUTDOWN command 2nd part

This section of the command goes into more functions, this start it is called __Z13getShutdownPwv, the main point of this function is to make sure the user has entered the correct password.

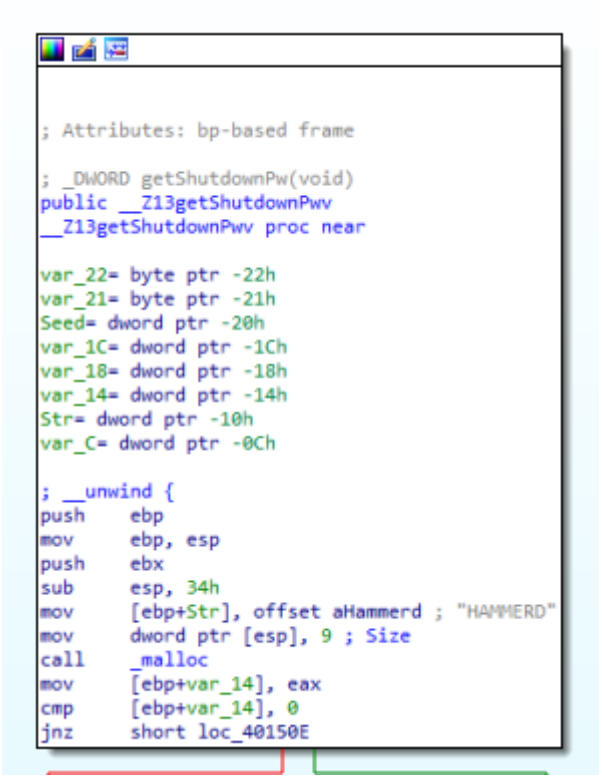

Figure 4.15.1: Inside **__Z13getShutdownPwv**

This is the beginning of the **getShutdownPw** function. I can see many different “var” lines, indicating many other variables with many different sizes. We can see a “HAMMERED”. This encryption key is used to ensure that the actual password's string size has reached the correct string length. Also, the storing of the string in the memory address of [ebp+var_14]

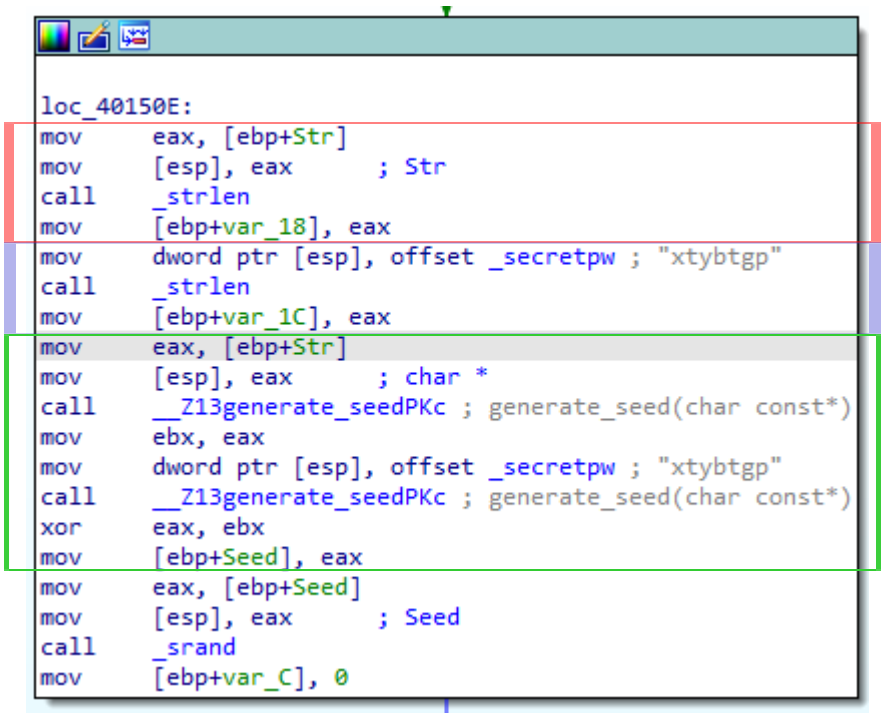

Figure 4.15.2: generate_seed

In red, I can see a section that calculates the length of the string in "[ebp+Str]" and then goes and stores this variable in [ebp+var_18].
In blue, it's similar to the previous step. It calculates the length of the string "xtybtgp" and then stores it in the variable [ebp+var_1C].
In green, this section utilises a generate_seed function with other strings and XOR the string to get a different value.

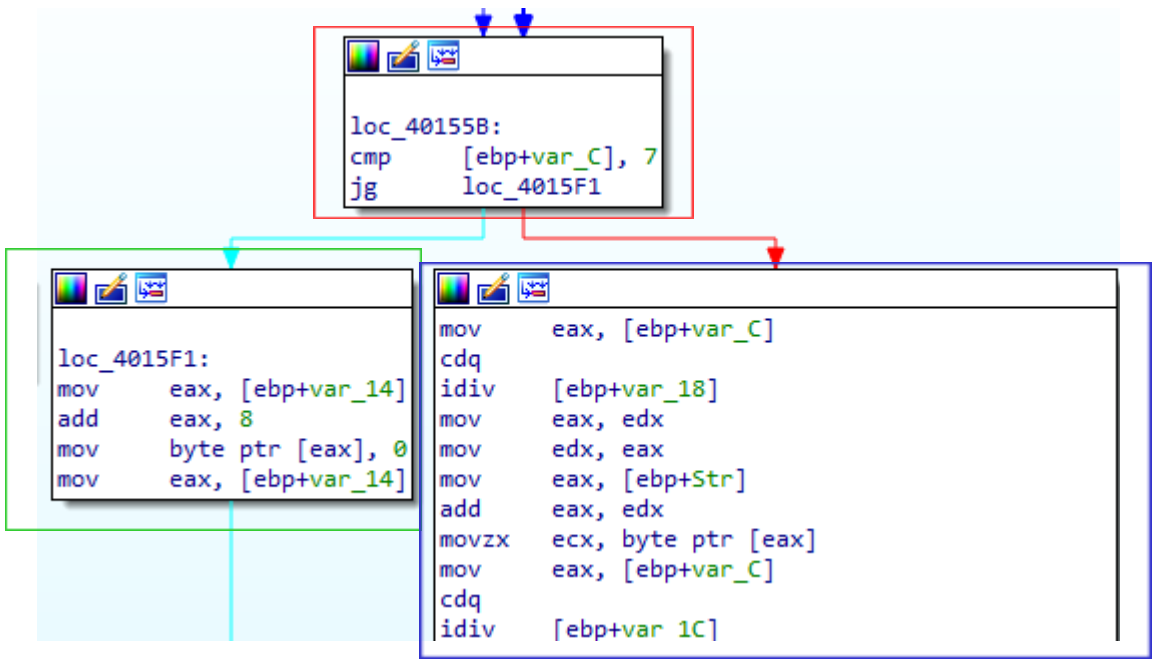

Figure 4.15.3:

The section in red compares the strings of the value stored in [ebp+var_C] to 7. It then jumps to the next lock if the comparison exceeds 7. For this function, it goes to the box in Blue and goes through that box seven times. I'll explain why in the following figure. The green section checks if this has been completed; if it has, it lets it through.

```
mov     eax, [ebp+var_C]
cdq
idiv    [ebp+var_18]
mov     eax, edx
mov     edx, eax
mov     eax, [ebp+Str]
add     eax, edx
movzx   ecx, byte ptr [eax]
mov     eax, [ebp+var_C]
cdq
idiv    [ebp+var_1C]
mov     eax, edx
movzx   eax, byte ptr _secretpw[eax] ; "xtybtgp"
xor     eax, ecx
mov     [ebp+var_21], al
call    _rand
mov     ecx, eax
mov     edx, 4EC4EC4Fh
mov     eax, ecx
imul    edx
sar     edx, 3
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
imul    eax, 1Ah
sub     ecx, eax
mov     eax, ecx
mov     [ebp+var_22], al
mov     edx, [ebp+var_C]
mov     eax, [ebp+var_14]
lea     ebx, [edx+eax]
movsx   edx, [ebp+var_21]
movsx   eax, [ebp+var_22]
lea     ecx, [edx+eax]
mov     edx, 4EC4EC4Fh
mov     eax, ecx
imul    edx
sar     edx, 3
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
imul    eax, 1Ah
sub     ecx, eax
mov     eax, ecx
add     eax, 41h ; 'A'
mov     [ebx], al
add     [ebp+var_C], 1
jmp     loc_40155B
```

Figure 4.15.4: loop function

This is a loop, and this loop runs around seven times. It relates to the two different strings we had seen before, “HAMMERED” and “xtybtgp”. The section in red divides the values relating to [ebp+var_C] by the value in [ebp+var_18], which we know are the two strings mentioned before. This calculation is then loaded into the eax. The blue section again shows more division, but this time, it takes the same and relates it to the value [ebp+var_1C]. Green demonstrates the generation of a random number using the _rand function. Purple is performing operations on this random number that has been generated. Overall, this function modified the characters in the memory, and it does this seven times until the desired string is needed, to which it will go down the other route in Figure 4.15.3. After all this, it then leaves the get shutdown pw function.

```
mov     eax, _sock
mov     edx, [ebp+lpText]
mov     [ebp+Str2], edx
lea     edx, [ebp+Destination]
mov     [ebp+var_30], edx
mov     [ebp+s], eax
call    __Z13getShutdownPwv ; getShutdownPw(void)
mov     [ebp+var_38], eax
mov     eax, [ebp+Str2]
mov     [esp+4], eax    ; Str2
mov     eax, [ebp+var_38]
mov     [esp], eax      ; Str1
call    _strcmp
test    eax, eax
jnz     loc_402894
```

```
mov     eax, [ebp+var_30]
mov     [esp+8], eax
mov     dword ptr [esp+4], offset Format ; "PRIVMSG %s :Shutdown password entered -"...
lea     eax, [ebp+Text]
mov     [esp], eax      ; Buffer
call    _sprintf
lea     eax, [ebp+Text]
mov     [esp+4], eax    ; Str
mov     eax, [ebp+s]
mov     [esp], eax      ; s
call    __Z9writeLinejPc ; writeLine(uint,char *)
```

```
loc_402894:
mov     dword ptr [esp+4], offset _secretpw ; "xtybtgp"
mov     eax, [ebp+Str2]
mov     [esp], eax      ; Str1
call    _strcmp
test    eax, eax
jnz     short loc_4028DF
```

Figure 4.16: Password authentication

In red, we can see a section that goes on to the authentication of the strings entered by the user after it leaves that previous function. I can see two “str” referenced: “Str2” “Str1” from what we can see in the blue section, we can see one of the strings relates to the string “xtybtgp”. So from

this, we can determine that the correct string to the password is “Str2”, which is connected to the section in green which, when inputted, displays the result: “PRIVMSG %s :Shutdown password entered - botnet shutting down”, then shows this message “Botnet shutdown by user %s. Would you like to restart it?”

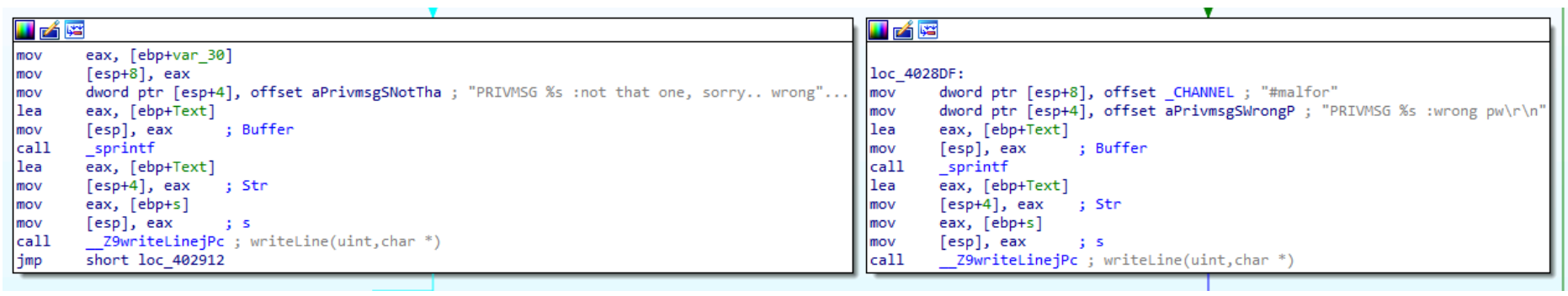

Figure 4.16.1: Password failure

These two sections relate to if a user enters an incorrect password. The right side shows what the user sees if they enter any password that doesn't relate to "xtybtgp". If that condition is met, it uses the bots to send a private message saying, "PRIVMSG %s :wrong pw". If a password that does not meet those conditions is entered, it will output this: "PRIVMSG %s :not that one, sorry.. wrong pw!"

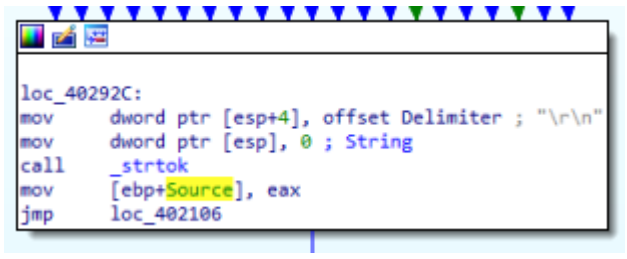

Figure 4.17: final function in IRC client

This is the final function in the ircclient function. After a user has entered a condition from the list of different strings/commands they can use, it goes back to Figure 4.18. This function is set just before the commands. This function checks if the malware is running to determine its usability. If it is, it goes to the beginning of the command IF statement.

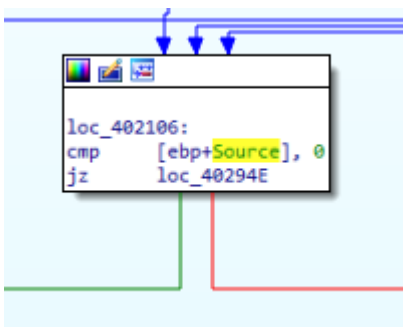

Figure 4.18: Function before commands

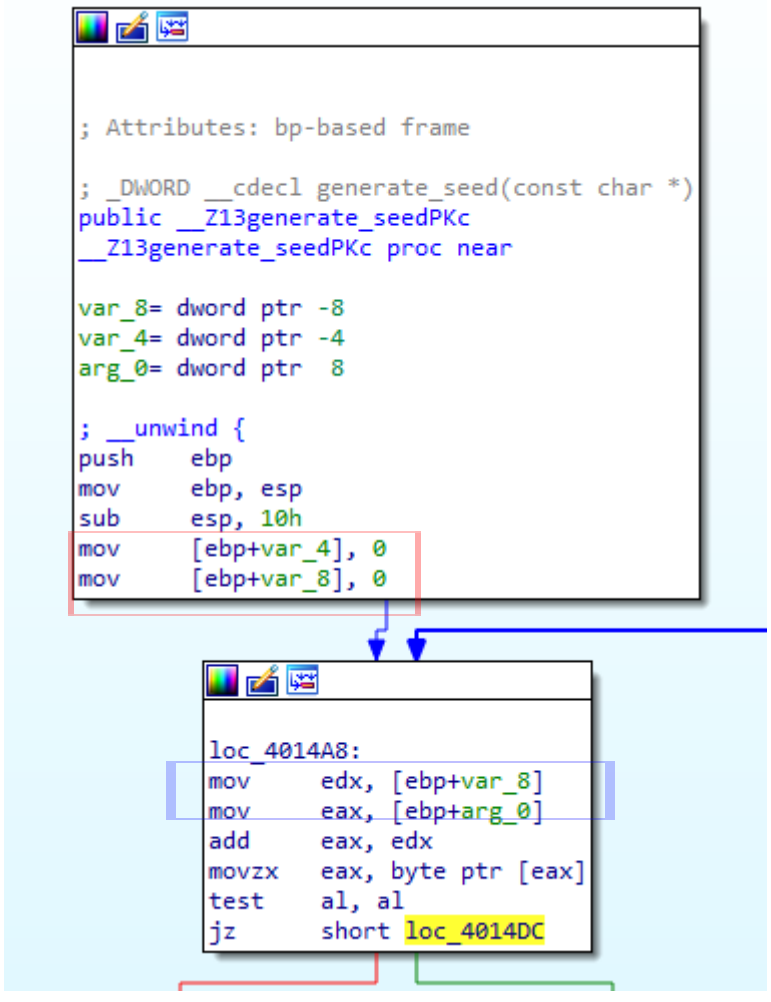

Figure 5: generate seed function

This function is called the **__Z13generate_seedPKc**. It starts with variables and sets them to 0, shown in red. It then moves those values to different parts of the memory, “edx” and “eax”, shown in blue. After that bit, it adds the variables together and stores them under “eax”, which the test function seeing if this reference variable is 0, will then complete the function. However, there is a section for if it is not 0 Figure 5.1

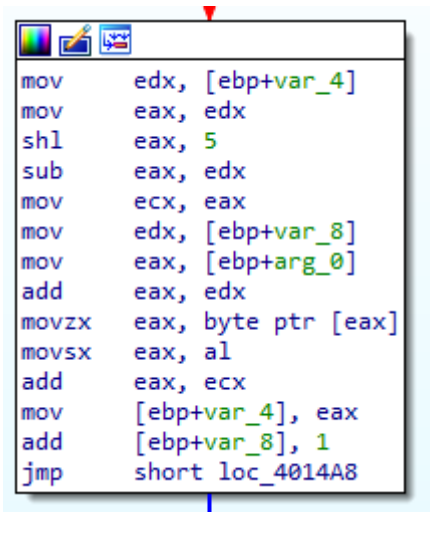

Figure 5.1: loop

When the variable from before is not a 0, it goes through this loop until it has reached 0 to which the function ends and carries on in the previous function which for this was getshutdownpw (Figure 4.15.1).

References

karl-bridge-microsoft. (n.d.). *GetModuleHandleA function (libloaderapi.h) - Win32 apps*. Learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea

karl-bridge-microsoft. (2021, June 29). *GetLastError function (errhandlingapi.h) - Win32 apps*. Learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror

karl-bridge-microsoft. (2023, February 9). *CreateMutexA function (synchapi.h) - Win32 apps*. Learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa

stevewhims. (n.d.). *WSAStartup function (winsock.h) - Win32 apps*. Learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-wsastartup

stevewhims. (2022, August 19). *gethostname function (winsock.h) - Win32 apps*. Learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-gethostname