

Hands-on Learning Exercise 2 to Accompany Chapter 4:

Assembling With Cygwin/NASM and Shellcode

Here is a basic helloworld.asm program in x86 assembly.

```
;msgbox.asm
[SECTION .text]

global _start

_start:
    ;eax holds return value
    ;ebx will hold function addresses
    ;ecx will hold string pointers
    ;edx will hold NULL

    xor eax,eax
    xor ebx,ebx           ;zero out the registers
    xor ecx,ecx
    xor edx,edx

    jmp short GetLibrary
LibraryReturn:
    pop ecx               ;get the library string
    mov [ecx + 10], dl    ;insert NULL
    mov ebx, 0x7c801d7b   ;LoadLibraryA(libraryname);
    push ecx              ;beginning of user32.dll
    call ebx              ;eax will hold the module handle

    jmp short FunctionName
FunctionReturn:

    pop ecx               ;get the address of the Function string
    xor edx,edx
    mov [ecx + 11],dl     ;insert NULL
    push ecx
    push eax
    mov ebx, 0x7c80ae30   ;GetProcAddress(hmodule,functionname);
    call ebx              ;eax now holds the address of MessageBoxA

    jmp short Message
MessageReturn:
```

```

    pop ecx                ;get the message string
    xor edx,edx
    mov [ecx+3],dl        ;insert the NULL

    xor edx,edx

    push edx              ;MB_OK
    push ecx              ;title
    push ecx              ;message
    push edx              ;NULL window handle

    call eax              ;MessageBoxA(windowhandle,msg,title,type); Address

ender:
    xor edx,edx
    push eax
    mov eax, 0x7c81cafa   ;exitprocess(exitcode);
    call eax              ;exit cleanly so we don't crash the parent program

    ;the N at the end of each string signifies the location of the NULL
    ;character that needs to be inserted

GetLibrary:
    call LibraryReturn
    db 'user32.dllN'
FunctionName
    call FunctionReturn
    db 'MessageBoxAN'
Message
    call MessageReturn
    db 'HeyN'

```

Copy and paste this a text editor and name the file hello.asm

This is the same basic program as the first tutorial but is formatted using NASM directives. NASM is another assembler.

Open Cygwin and type the following commands.

```
User@malware ~/asm  
$ nasm -f elf hello.asm; ld -o hello hello.o; objdump -d hello
```

This is actually 3 commands together, each separated by a semicolon.

```
nasm -f elf hello.asm          ;    This assembles hello.asm as an elf binary  
  
ld -o hello hello.o           ;    This links hello.o and creates the .  
                               executable hello  
objdump -d hello               ;    This disassembles the binary hello
```

If all goes correctly, you should get the following output:

```
Disassembly of section .text:
00401000 <_start>:
401000: 31 c0          xor     %eax,%eax
401002: 31 db          xor     %ebx,%ebx
401004: 31 c9          xor     %ecx,%ecx
401006: 31 d2          xor     %edx,%edx
401008: eb 37          jmp     401041 <GetLibrary>
0040100a <LibraryReturn>:
40100a: 59            pop     %ecx
40100b: 88 51 0a      mov     %dl,0xa(%ecx)
40100e: bb 7b 1d 80 7c mov     $0x7c801d7b,%ebx
401013: 51            push   %ecx
401014: ff d3        call   *%ebx
401016: eb 39          jmp     401051 <FunctionName>
00401018 <FunctionReturn>:
401018: 59            pop     %ecx
401019: 31 d2          xor     %edx,%edx
40101b: 88 51 0b      mov     %dl,0xb(%ecx)
40101e: 51            push   %ecx
40101f: 50            push   %eax
401020: bb 30 ae 80 7c mov     $0x7c80ae30,%ebx
401025: ff d3        call   *%ebx
401027: eb 39          jmp     401062 <Message>
00401029 <MessageReturn>:
401029: 59            pop     %ecx
40102a: 31 d2          xor     %edx,%edx
40102c: 88 51 03      mov     %dl,0x3(%ecx)
40102f: 31 d2          xor     %edx,%edx
401031: 52            push   %edx
401032: 51            push   %ecx
401033: 51            push   %ecx
401034: 52            push   %edx
401035: ff d0        call   *%eax
00401037 <sender>:
401037: 31 d2          xor     %edx,%edx
401039: 50            push   %eax
40103a: b8 fa ca 81 7c mov     $0x7c81cafa,%eax
40103f: ff d0        call   *%eax
00401041 <GetLibrary>:
401041: e8 c4 ff ff ff call   40100a <LibraryReturn>
401046: 75 73          jne    4010bb <__DTOR_LIST__+0x48>
401048: 65            gs
401049: 72 33          jb     40107e <__DTOR_LIST__+0xb>
40104b: 32 2e          xor     (%esi),%ch
40104d: 64            fs
40104e: 6c            insb   (%dx),%es:(%edi)
40104f: 6c            insb   (%dx),%es:(%edi)
401050: 4e            dec    %esi
00401051 <FunctionName>:
401051: e8 c2 ff ff ff call   401018 <FunctionReturn>
401056: 4d            dec    %ebp
401057: 65            gs
401058: 73 73          jae    4010cd <__DTOR_LIST__+0x5a>
40105a: 61            popa
40105b: 67            addr16
40105c: 65            gs
40105d: 42            inc    %edx
40105e: 6f            outsl  %ds:(%esi),(%dx)
40105f: 78 41          js     4010a2 <__DTOR_LIST__+0x2f>
401061: 4e            dec    %esi
00401062 <Message>:
401062: e8 c2 ff ff ff call   401029 <MessageReturn>
401067: 48            dec    %eax
401068: 65            gs
401069: 79 4e          jns    4010b9 <__DTOR_LIST__+0x46>
0040106b <__CTOR_LIST__>:
40106b: ff            (bad)
40106c: ff            (bad)
40106d: ff            (bad)
40106e: ff 00        incl   (%eax)
401070: 00 00        add   %al,(%eax)
...
00401073 <__DTOR_LIST__>:
401073: ff            (bad)
401074: ff            (bad)
401075: ff            (bad)
401076: ff 00        incl   (%eax)
401078: 00 00        add   %al,(%eax)
```

We are just interested in these sections:

Disassembly of section .text:

```
00401000 <_start>:
401000: 31 c0          xor    %eax,%eax
401002: 31 db          xor    %ebx,%ebx
401004: 31 c9          xor    %ecx,%ecx
401006: 31 d2          xor    %edx,%edx
401008: eb 37          jmp   401041 <GetLibrary>

0040100a <LibraryReturn>:
40100a: 59            pop    %ecx
40100b: 88 51 0a      mov    %dl,0xa(%ecx)
40100e: bb 7b 1d 80 7c mov    $0x7c801d7b,%ebx
401013: 51            push   %ecx
401014: ff d3        call   *%ebx
401016: eb 39          jmp   401051 <FunctionName>

00401018 <FunctionReturn>:
401018: 59            pop    %ecx
401019: 31 d2          xor    %edx,%edx
40101b: 88 51 0b      mov    %dl,0xb(%ecx)
40101e: 51            push   %ecx
40101f: 50            push   %eax
401020: bb 30 ae 80 7c mov    $0x7c80ae30,%ebx
401025: ff d3        call   *%ebx
401027: eb 39          jmp   401062 <Message>

00401029 <MessageReturn>:
401029: 59            pop    %ecx
40102a: 31 d2          xor    %edx,%edx
40102c: 88 51 03      mov    %dl,0x3(%ecx)
40102f: 31 d2          xor    %edx,%edx
401031: 52            push   %edx
401032: 51            push   %ecx
401033: 51            push   %ecx
401034: 52            push   %edx
401035: ff d0        call   *%eax

00401037 <ender>:
401037: 31 d2          xor    %edx,%edx
401039: 50            push   %eax
40103a: b8 fa ca 81 7c mov    $0x7c81cafa,%eax
40103f: ff d0        call   *%eax

00401041 <GetLibrary>:
401041: e8 c4 ff ff ff call   40100a <LibraryReturn>
401046: 75 73          jne   4010bb <__DTOR_LIST__+0x48>
401048: 65            gs
401049: 72 33          jb    40107e <__DTOR_LIST__+0xb>
40104b: 32 2e          xor    (%esi),%ch
40104d: 64            fs
40104e: 6c            insb  (%dx),%es:(%edi)
40104f: 6c            insb  (%dx),%es:(%edi)
401050: 4e            dec   %esi

00401051 <FunctionName>:
401051: e8 c2 ff ff ff call   401018 <FunctionReturn>
401056: 4d            dec   %ebp
401057: 65            gs
401058: 73 73          jae   4010cd <__DTOR_LIST__+0x5a>
40105a: 61            popa
40105b: 67            addr16
40105c: 65            gs
40105d: 42            inc   %edx
40105e: 6f            outsl %ds:(%esi),(%dx)
40105f: 78 41          js    4010a2 <__DTOR_LIST__+0x2f>
401061: 4e            dec   %esi

00401062 <Message>:
401062: e8 c2 ff ff ff call   401029 <MessageReturn>
401067: 48            dec   %eax
401068: 65            gs
401069: 79 4e          jns   4010b9 <__DTOR_LIST__+0x46>

0040106b <__CTOR_LIST__>:
40106b: ff            (bad)
40106c: ff            (bad)
40106d: ff            (bad)
40106e: ff 00        incl  (%eax)
401070: 00 00        add  %al,(%eax)
...

00401073 <__DTOR_LIST__>:
401073: ff            (bad)
401074: ff            (bad)
401075: ff            (bad)
401076: ff 00        incl  (%eax)
401078: 00 00        add  %al,(%eax)
...
```

These contain the hexadecimal numbers for our assembly code.

We see it starts with the hex numbers 31 c0 31 db 31 c9 31 d2 eb 37

Disassembly of section .text:

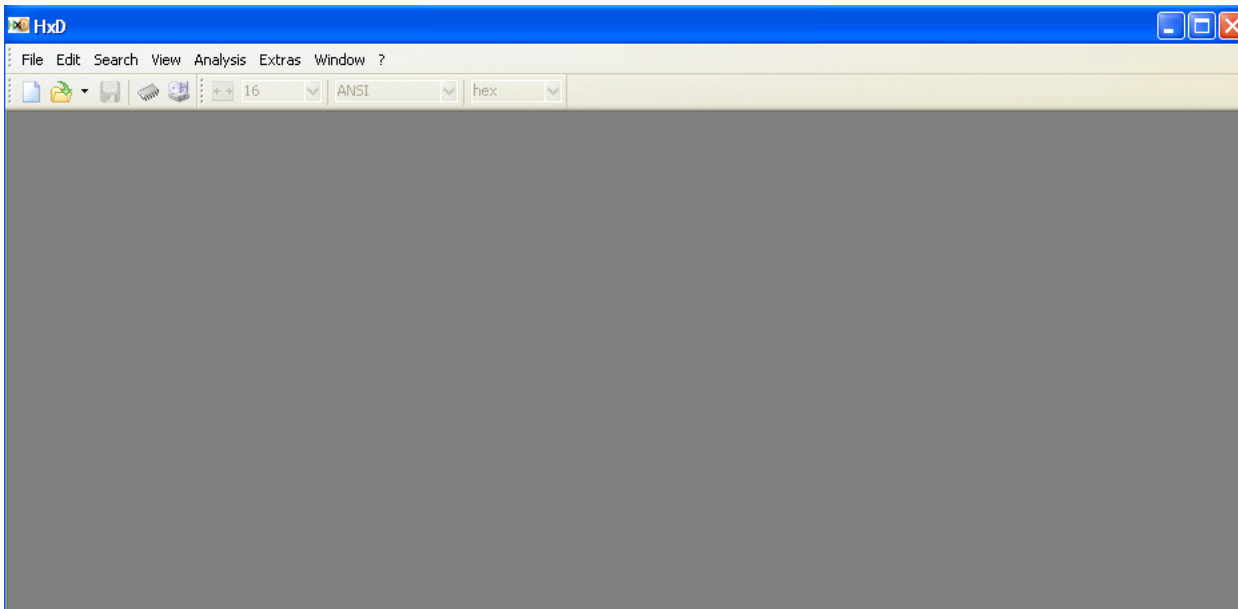
```
00401000 <_start>:
 401000:    31 c0          xor    %eax,%eax
 401002:    31 db          xor    %ebx,%ebx
 401004:    31 c9          xor    %ecx,%ecx
 401006:    31 d2          xor    %edx,%edx
 401008:    eb 37          jmp   401041 <GetLibrary>
```

...
...
...

And the last section ends with e8 c2 ff ff ff 48 65 79 4e

```
0401062 <Message>:
 401062:    e8 c2 ff ff ff  call  401029 <MessageReturn>
 401067:    48             dec   %eax
 401068:    65             gs
 401069:    79 4e          jns  4010b9 <__DTOR_LIST__+0x46>
```

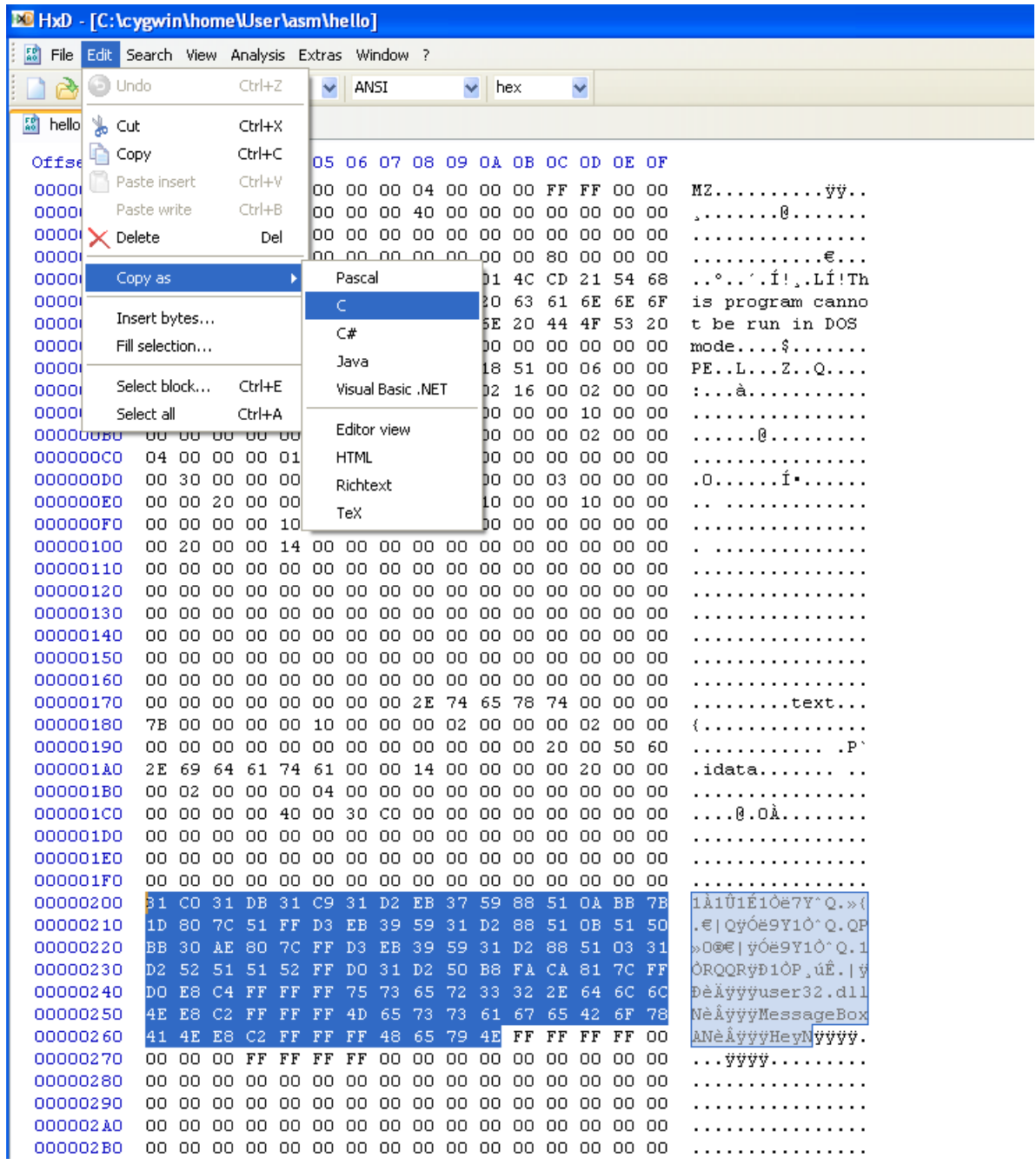
Let us now open the hello executable in the hex editor HxD. Start→Programs→HxD Hex Editor→HxD



Highlight from the 31 c0 to the 79 4e:

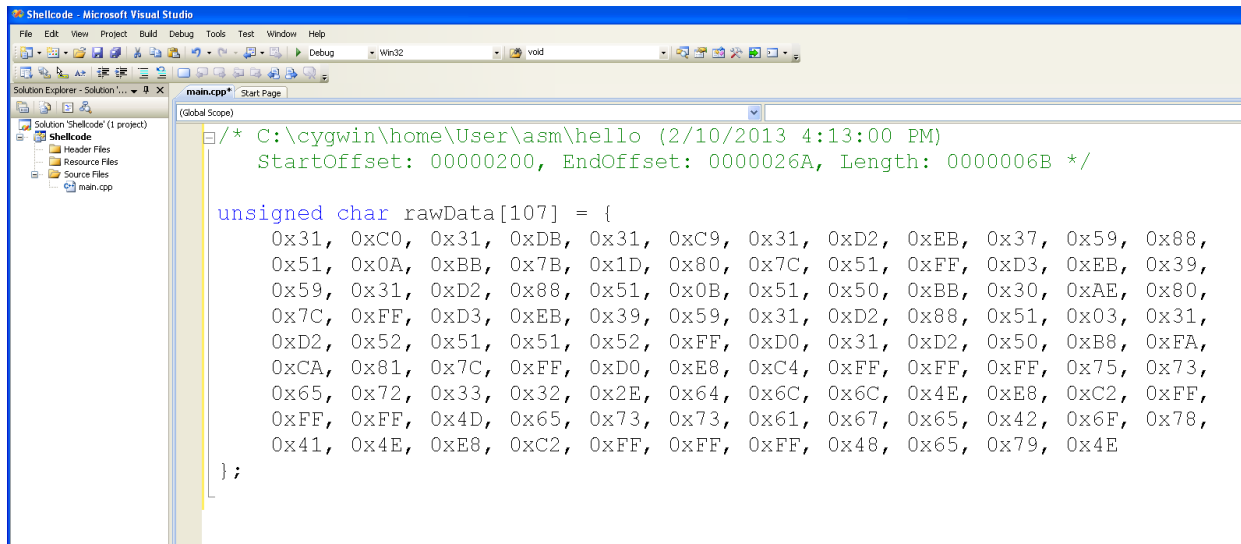
```
00000090 3A 00 00 00 E0 00 07 03 0B 01 02 16 00 02 00 00 :...à.....
000000A0 00 02 00 00 00 00 00 00 10 00 00 00 10 00 00 .....
000000B0 00 00 00 00 00 00 40 00 00 10 00 00 00 02 00 .....@.....
000000C0 04 00 00 00 01 00 00 00 04 00 00 00 00 00 00 .....
000000D0 00 30 00 00 00 02 00 00 CD 95 00 00 03 00 00 00 .0.....í*.....
000000E0 00 00 20 00 00 10 00 00 00 00 10 00 00 10 00 00 ..
000000F0 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 .....
00000100 00 20 00 00 14 00 00 00 00 00 00 00 00 00 00 .....
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000170 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 .....text...
00000180 7B 00 00 00 00 10 00 00 00 02 00 00 00 02 00 00 {...}.....
00000190 00 00 00 00 00 00 00 00 00 00 00 20 00 50 60 .....P`
000001A0 2E 69 64 61 74 61 00 00 14 00 00 00 00 20 00 00 .idata.....
000001B0 00 02 00 00 00 04 00 00 00 00 00 00 00 00 00 .....
000001C0 00 00 00 00 40 00 30 C0 00 00 00 00 00 00 00 .....@.0À.....
000001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000200 B1 C0 31 DB 31 C9 31 D2 EB 37 59 88 51 0A BB 7B 1À1Û1É1Ôè7Y^Q.»{
00000210 1D 80 7C 51 FF D3 EB 39 59 31 D2 88 51 0B 51 50 .€|Qyóè9Y1ó^Q.QP
00000220 BB 30 AE 80 7C FF D3 EB 39 59 31 D2 88 51 03 31 »00€|ýóè9Y1ó^Q.1
00000230 D2 52 51 51 52 FF D0 31 D2 50 B8 FA CA 81 7C FF ÓRQQRýD1ÓP,úÉ.|ý
00000240 D0 E8 C4 FF FF FF 75 73 65 72 33 32 2E 64 6C 6C ðèÀýýýuser32.dll
00000250 4E E8 C2 FF FF FF 4D 65 73 73 61 67 65 42 6F 78 NèÀýýýMessageBox
00000260 41 4E E8 C2 FF FF FF 48 65 79 4E FF FF FF FF 00 ANèÀýýýHeyMýýýý.
00000270 00 00 00 FF FF FF FF 00 00 00 00 00 00 00 00 00 ...ýýýý.....
00000280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000002F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Then go to the top toolbar, Edit→Copy As→C



Now create a new project in Visual Studio, make a Win32 console project, and make sure it's an empty project. If you need help doing this, see the first tutorial.

Paste the hex you copied as C from HxD. It should look like this.



Now we are going to fill in the `main()` part of our c program to call this code. It will look like this. Below the picture is the code you can copy and paste.

```

/* StartOffset: 00000200, EndOffset: 0000026A, Length: 0000006B */

unsigned char rawData[107] = {
    0x31, 0xC0, 0x31, 0xDB, 0x31, 0xC9, 0x31, 0xD2, 0xEB, 0x37, 0x59, 0x88,
    0x51, 0x0A, 0xBB, 0x7B, 0x1D, 0x80, 0x7C, 0x51, 0xFF, 0xD3, 0xEB, 0x39,
    0x59, 0x31, 0xD2, 0x88, 0x51, 0x0B, 0x51, 0x50, 0xBB, 0x30, 0xAE, 0x80,
    0x7C, 0xFF, 0xD3, 0xEB, 0x39, 0x59, 0x31, 0xD2, 0x88, 0x51, 0x03, 0x31,
    0xD2, 0x52, 0x51, 0x51, 0x52, 0xFF, 0xD0, 0x31, 0xD2, 0x50, 0xB8, 0xFA,
    0xCA, 0x81, 0x7C, 0xFF, 0xD0, 0xE8, 0xC4, 0xFF, 0xFF, 0xFF, 0x75, 0x73,
    0x65, 0x72, 0x33, 0x32, 0x2E, 0x64, 0x6C, 0x6C, 0x4E, 0xE8, 0xC2, 0xFF,
    0xFF, 0xFF, 0x4D, 0x65, 0x73, 0x73, 0x61, 0x67, 0x65, 0x42, 0x6F, 0x78,
    0x41, 0x4E, 0xE8, 0xC2, 0xFF, 0xFF, 0xFF, 0x48, 0x65, 0x79, 0x4E
};

int main()
{
    int (*func) (); //declare pointer to function that returns an int and takes a void or no args
    func = (int (*) ())&rawData; //assign func pointer to address of code, (int (*) ()) is the type cast.
    func(); //call shellcode
}

```

```

int main()
{
    int (*func) ();
    func = (int (*) ())&code;
    func();
}

```

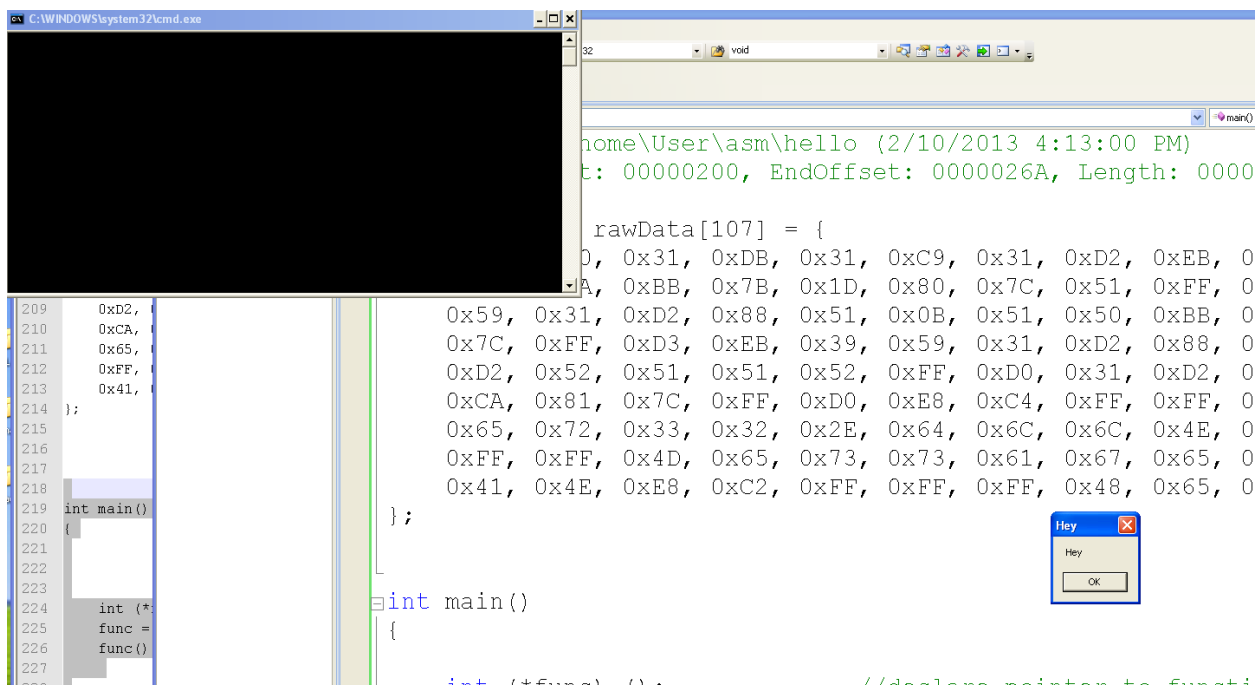
Let's go over what each line is doing.

`int (*func) ()` This declares a pointer to function that returns an int and takes a void or no args

`func = (int (*) ())&code;` This assigns a function pointer to address of code, `(int (*) ())` is the type cast.

`func();` This calls the function

Build it and run it.



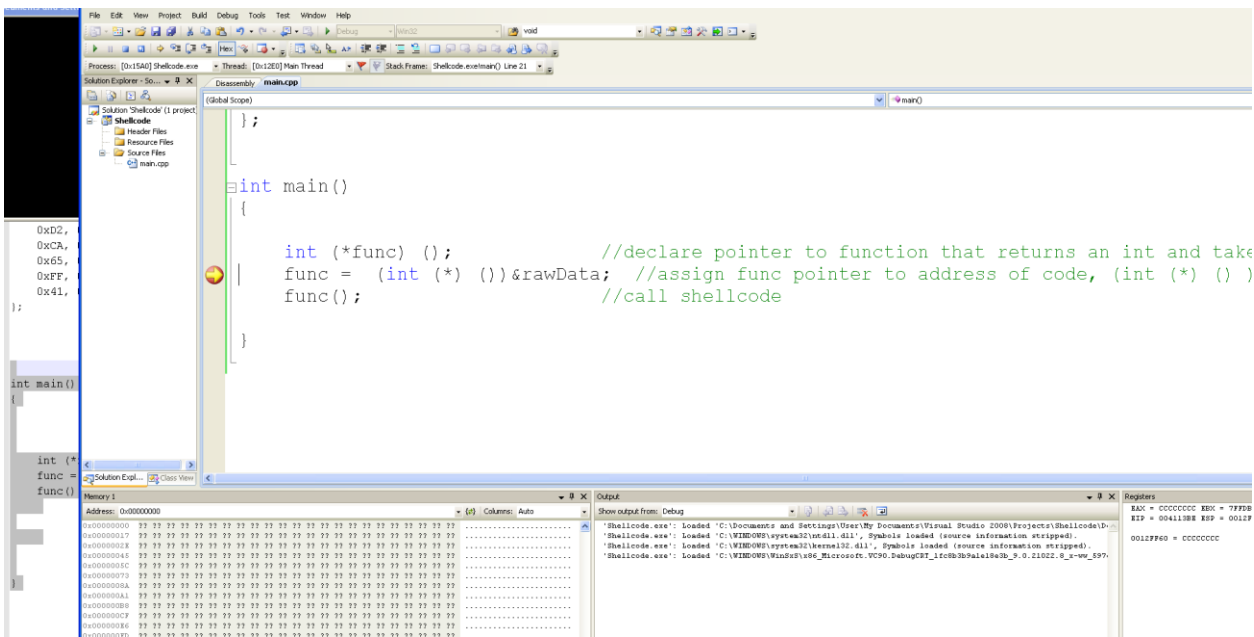
We get another popup box.

Let's use the debugger and step through the code.

Left click the side bar next to `int (*func) ();`; This will place a breakpoint which is indicated by a big red circle.

```
int main()
{
    int (*func) (); //declare pointer to function that returns an int and take
    func = (int (*) ())&rawData; //assign func pointer to address of code, (int (*) ())
    func(); //call shellcode
}
```

Go up to the top toolbar, **Debug**→**Start Debugging**



A tab should open near the top that says disassembly. If this tab does not open, press **alt+8** to open it.

Click the disassembly tab. You should now see your c code in bold black, with the assembly code in grey. The c code is acting like comments to the assembly code. The assembly code is actually being executed.

Now to move forward in the code, press **F10**. This is step over.

```
Disassembly main.cpp
Address: main(void)
};

int main()
{
004113A0 push    ebp
004113A1 mov     ebp,esp
004113A3 sub     esp,0CCh
004113A9 push    ebx
004113AA push    esi
004113AB push    edi
004113AC lea   edi,[ebp-0CCh]
004113E2 mov     ecx,33h
004113E7 mov     eax,0CCCCCCCCh
004113EC rep stos dword ptr es:[edi]

    int (*func) (); //declare pointer to function that returns an int and takes a void or no args
    func = (int (*) ())&rawData; //assign func pointer to address of code, (int (*) ()) is the type cast.
004113EE mov     dword ptr [func],offset rawData (417000h)
    func(); //call shellcode
004113C5 mov     esi,esp
004113C7 call   dword ptr [func]
004113CA cmp     esi,esp
004113CC call   @ILT+310(__RTC_CheckEsp) (41113Bh)

}
004113D1 xor     eax,eax
004113D3 pop     edi
004113D4 pop     esi
004113D5 pop     ebx
004113D6 add     esp,0CCh
004113DC cmp     ebp,esp
004113DE call   @ILT+310(__RTC_CheckEsp) (41113Bh)
004113E3 mov     esp,ebp
004113E5 pop     ebp
004113E6 ret

--- No source file ---
```

You can press F10 until the pop-up box says hello and the program ends. But we want to see our character array filled with hex code execute.

When you reach the line that says:

```
Call dword, ptr [func]
```

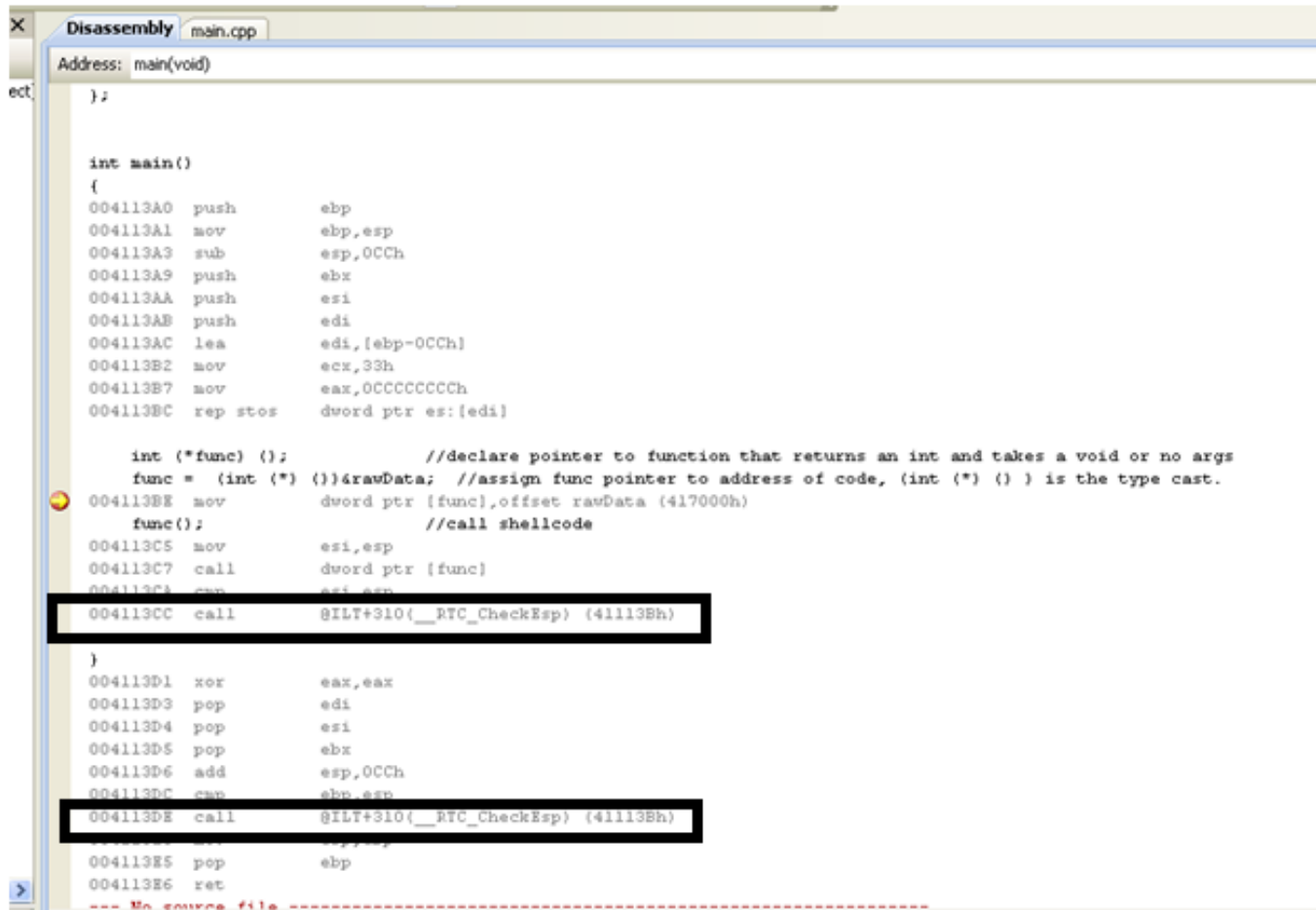
Press F11. This will step into, which means it will follow the assembly into that function.

If you press F11 when the yellow arrow is on the line that says `call dword, ptr [func]` you will then jump to that code. That should look like this:

```
00416FF8 add     byte ptr [eax],al
00416FFA add     byte ptr [eax],al
00416FFC add     byte ptr [eax],al
00416FFE add     byte ptr [eax],al
rawData:
00417000 xor     eax,eax
00417002 xor     ebx,ebx
00417004 xor     ecx,ecx
00417006 xor     edx,edx
00417008 jmp     rawData+41h (417041h)
0041700A pop     ecx
0041700B mov     byte ptr [ecx+0Ah],dl
0041700E mov     ebx,offset _LoadLibrary@4 (7C801D7Bh)
00417013 push   ecx
00417014 call   ebx
00417016 jmp     rawData+51h (417051h)
00417018 pop     ecx
00417019 xor     edx,edx
0041701B mov     byte ptr [ecx+0Bh],dl
0041701E push   ecx
0041701F push   eax
00417020 mov     ebx,offset _GetProcAddress@8 (7C80AE30h)
00417025 call   ebx
00417027 jmp     rawData+62h (417062h)
00417029 pop     ecx
0041702A xor     edx,edx
0041702C mov     byte ptr [ecx+3],dl
0041702F xor     edx,edx
00417031 push   edx
00417032 push   ecx
00417033 push   ecx
00417034 push   edx
00417035 call   eax
00417037 xor     edx,edx
00417039 push   eax
0041703A mov     eax,offset _ExitProcess@4 (7C81CAFah)
0041703E call   eax
```

You are now looking at the code then you original assembled using NASM and Cygwin. If has been translated into hex data and entered into our program through the source code.

Going back to when we first started debugging. There are a lot of extra assembly instructions in there that we weren't expecting. These are security features added to our program by Microsoft.



```
Disassembly main.cpp
Address: main(void)
};

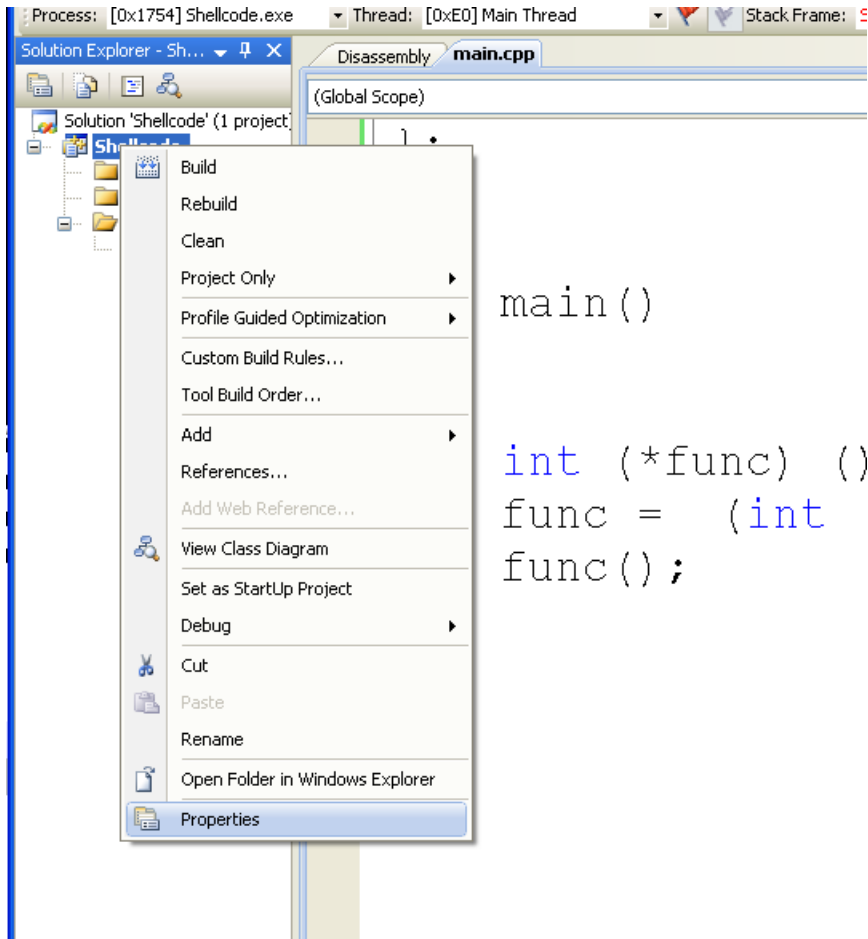
int main()
{
004113A0 push    ebp
004113A1 mov     ebp,esp
004113A3 sub     esp,0CCh
004113A9 push    ebx
004113AA push    esi
004113AB push    edi
004113AC lea    edi,[ebp-0CCh]
004113B2 mov     ecx,33h
004113B7 mov     eax,0CCCCCCCCh
004113BC rep stos dword ptr es:[edi]

    int (*func) (); //declare pointer to function that returns an int and takes a void or no args
    func = (int (*) ({}))&rawData; //assign func pointer to address of code, (int (*) ({})) is the type cast.
004113BE mov     dword ptr [func],offset rawData (417000h)
    func(); //call shellcode
004113C5 mov     esi,esp
004113C7 call   dword ptr [func]
004113CA can   esi,esp
004113CC call   @ILT+310( __RTC_CheckEsp ) (41113Bh)

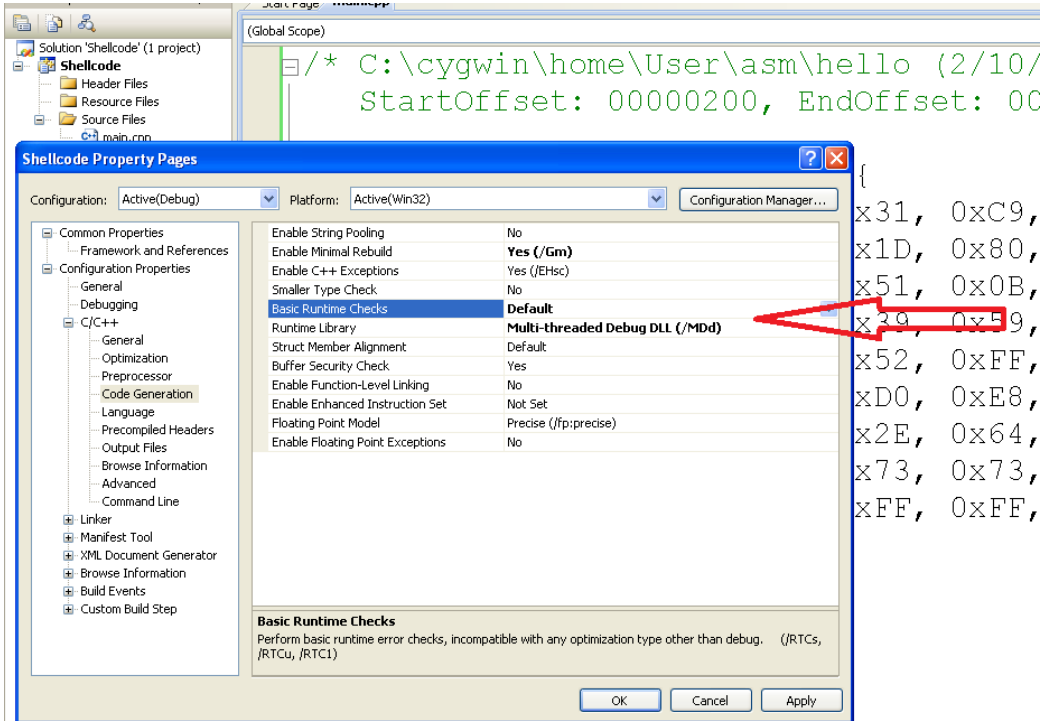
}
004113D1 xor     eax,eax
004113D3 pop     edi
004113D4 pop     esi
004113D5 pop     ebx
004113D6 add     esp,0CCh
004113DC can   ebp,esp
004113DE call   @ILT+310( __RTC_CheckEsp ) (41113Bh)
004113E5 pop     ebp
004113E6 ret

--- No source file ---
```

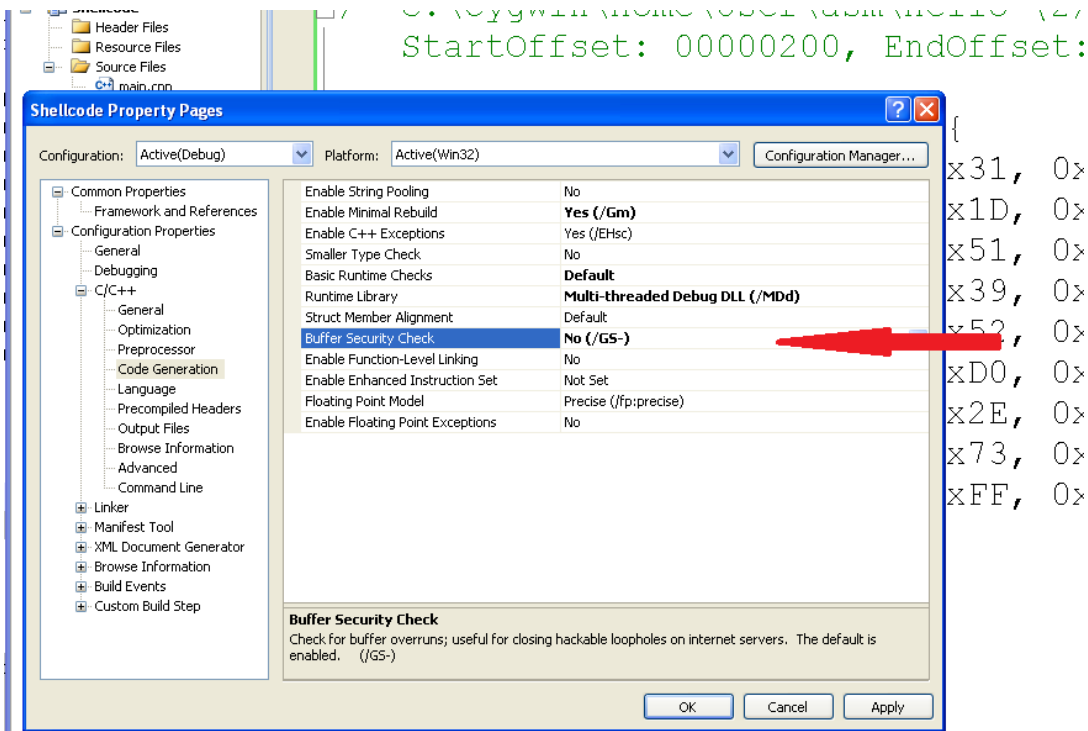

We can remove some of these for the sake of clarity by going to project→properties. Make sure you are no longer debugging.



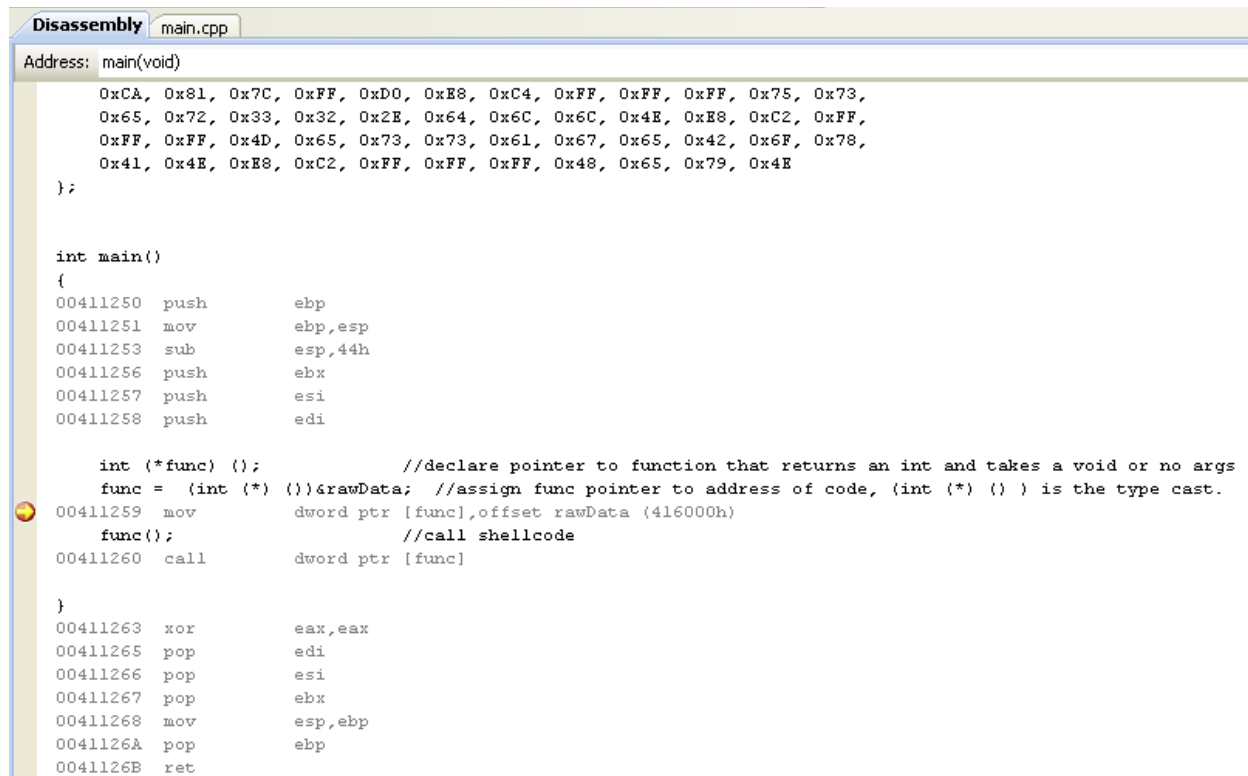
Then go to Configuration Properties→C/C++→Code Generation→Basic Runtime Checks and set it to default.



Also set Buffer Security Check to No.



Now rebuild and start debugging. The assembly should be significantly less cluttered.



```
Disassembly main.cpp
Address: main(void)
    0xCA, 0x81, 0x7C, 0xFF, 0xD0, 0xE8, 0xC4, 0xFF, 0xFF, 0xFF, 0x75, 0x73,
    0x65, 0x72, 0x33, 0x32, 0x2E, 0x64, 0x6C, 0x6C, 0x4E, 0xE8, 0xC2, 0xFF,
    0xFF, 0xFF, 0x4D, 0x65, 0x73, 0x73, 0x61, 0x67, 0x65, 0x42, 0x6F, 0x78,
    0x41, 0x4E, 0xE8, 0xC2, 0xFF, 0xFF, 0xFF, 0x48, 0x65, 0x79, 0x4E
};

int main()
{
00411250  push     ebp
00411251  mov     ebp,esp
00411253  sub     esp,44h
00411256  push     ebx
00411257  push     esi
00411258  push     edi

    int (*func) ();          //declare pointer to function that returns an int and takes a void or no args
    func = (int (*) ())&rawData; //assign func pointer to address of code, (int (*) ()) is the type cast.
00411259  mov     dword ptr [func],offset rawData (416000h)
    func();                //call shellcode
00411260  call    dword ptr [func]

}
00411263  xor     eax,eax
00411265  pop     edi
00411266  pop     esi
00411267  pop     ebx
00411268  mov     esp,ebp
0041126A  pop     ebp
0041126B  ret
```

There is a repository of shellcode at <http://www.shell-storm.org/shellcode/>

See what you can get to run on your computer. I will have the third tutorial up shortly that will show how to take control of a vulnerable program and inject shellcode into it.