CHAPTER

21

Binary Auditing: Hacking Closed Source Software

Many security-critical and widely deployed code bases are closed source, including some of the dominant operating system families for both servers and desktops. In order to assess the security of closed source software beyond the capabilities of fuzz-testing, binary auditing is a necessity.

In general, binary auditing is considered more difficult than auditing with source code. While this might seem like bad news for beginners, it could also be considered a benefit. There are far fewer people auditing binaries at this point in time, and fewer eyes make for easier work. Many bug classes that are virtually extinct in open source software still linger in closed source commercial code bases.

Binary auditing is still an imperfect science, and many things that can be quite easily verified while auditing source code are conversely quite difficult to determine while examining a binary. With practice and the help of some useful tools, much of the frustration associated with binary auditing can be removed.

Many security researchers do not stretch beyond the limitations of fuzz-testing when auditing commercial software. Although fuzz-testing has proven that it can reveal bugs in software, you really cannot fuzz all possible input patterns to any large piece of software in a reasonable amount of time. Binary auditing can offer a more complete view of the inner workings of an application and security flaws it might contain.

Binary versus Source-Code Auditing: The Obvious Differences

Binary auditing can be likened to source-code auditing in that you're looking for the same bug classes and flaws in software; however, the method for looking for them has changed. If you are already familiar with source-code auditing, you probably won't have to change your thought process much. However, your methodology will change quite a bit.

First and foremost, you'll need an excellent understanding of the assembly language relevant to the platform on which your binary will run. If you're unclear on any important instructions, you will likely misinterpret much of the code you read and end up confused and frustrated. If you're not able to read and understand a disassembly, thoroughly learning the relevant assembly language is a good place to start.

Some binaries, especially p-code binaries such as Java classes or Visual Basic applications, can be fully decompiled to something that closely resembles their original source code. However, most binaries cannot be reliably decompiled with today's tools. This chapter focuses on auditing Intel x86 binaries, especially those compiled with the Microsoft Visual C++ compiler.

When auditing a binary, as when auditing source code, it's paramount to understand the code you're reading. However, what might be a very obvious security check in source can often translate into one or two instructions. Therefore, it's definitely necessary to remain aware of program execution at any point in a function. For example, it's often necessary to know what values are stored in what registers at a certain point in execution, and many values may be swapped in and out of a particular register in any given block of code.

Some vulnerabilities are just as easy or easier to spot in a binary than in source code; however, most bugs will be more subtle and harder to detect for someone attempting their first binary audit. As you become more familiar with code constructed by certain compilers, auditing binaries will become nearly as easy as auditing source code.

IDA Pro—The Tool of the Trade

Interactive Disassembler Pro, more commonly known as IDA Pro, is well recognized as the best tool for analyzing or auditing binaries. It is developed and sold by the Belgian company, Datarescue (`www.datarescue.com`), and available for a reasonable price. If you will be doing a large amount of binary auditing, you should seriously consider purchasing a license. Although IDA Pro does have its shortfalls, it is still a very good disassembler and far ahead of its competition.

IDA Pro supports many different binary formats across a multitude of platforms and will most likely support even the most obscure formats that you want to disassemble. It stores disassembled program output in a database format and allows for the naming and renaming of virtually every aspect of the program being analyzed. Line-by-line comments are a feature that is often helpful when you are trying to analyze complex code constructs. Like many disassemblers, IDA Pro can list strings and cross references to most pieces of code or data.

Features: A Quick Crash Course

A basic understanding of the features of IDA Pro will help enormously with any binary analysis you do. It's obviously not necessary to understand all the advanced features to begin to audit binaries.

The main view of IDA Pro (View-A) is where most of the information you need will be found. This is the disassembly view and contains the disassembled representation of the code you're analyzing.

The display is color-coded to make viewing easier. Constant values are green, named values are blue, imported functions are pink, and most of the code is dark blue. You can also highlight a particular string in yellow by placing the cursor over it (this is very useful when trying to locate references to a particular address or register in a large block of code). The main view will show code on a function-by-function basis. Code regions that belong to valid functions have their addresses colored-coded black, and code regions that do not belong to any function are brown. Imported data addresses (IAT or idata) are pink, read-only data is grey, and writable data is yellow.

IDA Pro has a *hex-view*, where the hex and string representation of the code can be viewed. A names window lists all named locations in the application, a function window lists all functions found, and a strings window lists all known strings in the program. Other windows exist such as those for listing structures and enumerations. It's possible to find most of the information needed in these windows.

IDA Pro will store cross references for code that is pointed to by any jumps, calls, or data references. This is useful when tracing execution flow backwards from any location. It will also attempt to interpret the layout of the local stack for any function. IDA Pro will do this correctly for functions with a standard stack frame, but it occasionally has problems with functions that have optimized out the frame pointer.

IDA Pro has the ability to name any location in a program and to enter comments at any location. This makes code analysis much simpler and can make it a lot easier for you to come back to a piece of code the next day and still remember what's going on. IDA Pro also has had some code built in since version 4.2 that can represent code graphically. In many cases, this has turned

out to be very useful. There are several third-party plug-ins for IDA Pro that can also be useful, but most of them aren't specifically designed for binary auditing.

It is possible to specify the type of data located at any particular location in memory. Although IDA Pro will attempt to guess to the best of its ability whether a particular address contains code, binary data, string data, or other formats, it may not always get it right. The user has the ability to change anything that might not look quite right.

Debugging Symbols

Microsoft offers symbol information for download for every major revision of its operating systems. Windows symbol packages can be downloaded from the Windows Hardware and Driver Central page on Microsoft.com (`www.microsoft.com/whdc/hwdev/`) and are extremely useful when analyzing binaries. Symbols are generally distributed in the form of a PDB file, which is a program database format generated by MSVC++. At minimum, these files contain function names for nearly every function and static data location in a binary. For certain binaries, PDB files will contain undocumented internal structures and names for local variables. A binary is surprisingly easier to understand when everything has names.

Symbol packages are distributed by Microsoft on a service-pack basis, and are not generally available for hot fixes. Nearly every application, library, and driver within the core operating system will have publicly available symbols. IDA Pro can import PDB files and rename all the functions in a binary. In addition, third-party tools such as `PdbDump` can interpret PDB files and extract useful information.

Binary Auditing Introduction

In order to audit binaries successfully, you must understand compiler-generated code correctly and accurately. There are many compiler code constructs whose purposes aren't intuitive or immediately obvious. This chapter attempts to introduce binary auditors to most of the standard code constructs as well as non-standard code constructs that are often seen in code, with the hope of making compiled code almost as easy to understand as source code.

Stack Frames

Understanding the *stack frame* layout of any given function will make understanding the code much easier, and in some cases determining whether a

stack-based overflow exists will be much easier as well. Although there are some common stack frame layouts on x86, nothing is standardized, and the layout is compiler-determined. The more common examples are covered here.

Traditional BP-Based Stack Frames

The most common stack frame layout for functions is the *traditional BP-based frame* where the frame pointer register, `EBP`, is a constant pointer to the previous stack frame. The frame pointer is also a constant location relative to which function arguments and local stack variables are accessed.

The prologue for a function using this traditional stack frame looks like the following in Intel notation:

```
push ebp              // save the old frame pointer to the stack
mov ebp, esp          // set the new frame pointer to esp
sub esp, 5ch          // reserve space for local variables
```

At this point, local stack variables are located at a negative offset to `EBP`, and function arguments are located at a positive offset. The first function argument is found at `EBP + 8`. IDA Pro will rename the location `EBP + 8` to `EBP+arg_0`.

Nearly all references to arguments and local stack variables will be made relative to the frame pointer in functions with this frame type. This stack layout has been very well documented and is the easiest to follow when auditing. Most code generated by MSVC++ and by gcc will make use of this stack frame.

Functions without a Frame Pointer

For the sake of optimization, many compilers will generate code that optimizes out the use of the frame pointer. Some compilers may even in some cases use the frame pointer register as a general-purpose register. In this case, a function will access its arguments and local variables relative to the stack pointer `ESP` instead of the frame pointer. Although the frame pointer in a traditional stack frame is constant, the stack pointer floats location throughout the function, changing every time an operation pushes or pops something from the stack. The following example attempts to illustrate this:

```
this_function:
push esi
push edi
push ebx

push dword ptr [esp+10h]      // first argument to this_function
push dword ptr [esp+18h]      // second argument to this_function
call some_function
```

When the function is first entered, the first argument is at `ESP+4`. After saving three registers, that first argument is now at `ESP+10h`. After pushing the first function argument as a parameter to `some_function`, the second function argument is now located at `ESP+18h`.

IDA Pro makes an attempt to determine the location of the stack pointer at any given place in a function. By doing this, it tries to identify what stack pointer relative data accesses really refer to. However, when it does not know the calling conventions used by external functions, IDA Pro may get this wrong and create a very confusing disassembly. Sometimes, it may be necessary to manually calculate the location of the stack pointer at a certain point in a function in order to determine the size of stack buffers. Thankfully, this confusion does not happen too often.

Non-Traditional BP-Based Stack Frames

Microsoft Visual Studio .NET 2003 occasionally creates code with a stack frame that makes use of a constant frame pointer, although not in the traditional sense. When the frame pointer is constant, and all access to arguments and local stack variables are relative to it, it does not point to the calling function's frame pointer but rather to a location at a negative offset from where the traditional frame pointer would be. A sample function prologue might look like the following:

```
push ebp
lea ebp, [esp-5ch]
sub esp, 98h
```

The first function argument would be located at `EBP+64h`, instead of the traditional location of `EBP+8`. The memory range from `EBP-3ch` to `EBP+5ch` would be occupied by local stack variables.

The Windows Server 2003 operating system was compiled with code that contains this *non-traditional BP-based frame*, and it can be found throughout system libraries and services. At the time of writing, IDA Pro does not recognize this code construct and will completely misinterpret the local stack frame for functions with this type. Hopefully support for this compiler quirk will be added in the near future.

Calling Conventions

Different functions in an application may use different calling conventions, especially if parts of the application were written in different languages. It is useful to understand the different calling conventions seen in C-based languages.

In general, only two calling conventions will be commonly seen in C or C++ code generated by MSVC++ or gcc.

The C Calling Convention

The *C calling convention* does not only refer to C code, but is a way of passing arguments and restoring the program stack. With this calling convention, function arguments are pushed onto the stack from right to left as they appear in the source code. In other words, the last argument is pushed first and the first argument is the last one pushed prior to the function call. It is up to the calling function to restore its stack pointer after the call returns. An example of the C calling convention is:

```
some_function(some_pointer,some_integer);
```

This function call would look something like the following when using the C calling convention:

```
push some_integer
push some_pointer
call some_function
add esp, 8
```

Note that the second function argument is pushed before the first and that the stack pointer is restored by the calling function. Because this function had two arguments, the stack pointer had to be incremented by 8 bytes. It is also common to see the stack restored by using the x86 instruction `POP` with the destination of a scratch register. In this example, it would have been possible to restore the stack by doing `POP ECX` twice, restoring 4 bytes each time.

The Stdcall Calling Convention

The other calling convention commonly seen in C and C++ code is `Stdcall`. Arguments are passed in the same order as in the C calling convention, with the first function argument being pushed to the stack last before the function call. However, it is generally up to the called function to restore the stack. This is usually done on x86 by using the return instruction that releases stack space. For example, a function that has three arguments and uses the `Stdcall` calling convention would return with `RET 0Ch`, releasing 12 bytes from the stack upon return.

`Stdcall` is generally more efficient because the calling function does not have to release stack space. Functions that accept a variable number of arguments, such as `printf`-like functions, cannot release the stack space taken up by their arguments. This must be done by the calling function, which has knowledge of how many arguments existed.

Compiler-Generated Code

Compilers can generate much code that can be confusing at first glance. Let's look at some of the common areas in which a compiler will add instructions and at ways that we can recognize these compiler-generated structures.

Function Layouts

The layout of compiler-generated code in a function is somewhat variable. A function will generally begin with a function prologue and end with a function epilogue and a return. However, a function does not necessarily have to end in a return, and it is fairly common to see a function with code after its return instruction. This code will eventually jump back to the return instruction. Although a function may return in many places, the compiler will optimize the function's ability to jump to one common return location.

Since Visual Studio 6, the MSVC++ compiler has generated code with very unconventional function layouts. The compiler uses some logic to make determinations as to what branches are likely to be taken and which ones are less likely. Those deemed less likely are taken out of line of the main function and are placed as code fragments at far-off memory locations. These code snippets are often code that deals with uncommon error conditions or unlikely scenarios. However, vulnerabilities are often likely to exist in these code fragments and they should be reviewed when auditing binaries. These code fragments are often indicated by red jump arrows in IDA Pro and have been a common part of the MSVC++ compiled code for many years. IDA Pro may not deal properly with these code fragments and may not note accesses to local stack variables within them or graph them correctly.

In highly optimized code, several functions may share code fragments. For example, if several functions return in the same manner and restore the same registers and stack space, it is technically possible for them to share the same function epilogue and return code. However, this is quite uncommon and has only really been seen within `NTDLL` on Windows NT operating systems.

If Statements

`if` statements are one of the most common C code constructs and sometimes are very easy to see and interpret in compiled code. They are most often represented by the `CMP` or `TEST` instructions, followed by a conditional jump. The following example shows a simple C `if` statement and its corresponding assembly representation.

C Code:

```
int some_int;

if(some_int != 32)
     some_int = 32;
```

Compiled Representation (`ebp-4 = some_int`):

```
mov eax, [ebp-4]
cmp eax, 32
jnz past_next_instruction
mov eax, 32
```

`if` statements are generally characterized by forward jumps or branches; however, this is not necessarily true, and reorganization of code by the compiler can create havoc with this problem. In some contexts, it will be very obvious that a conditional branch was an `if` statement, but in other contexts `if` statements are difficult to differentiate from other code constructs such as loops. A better understanding of the overall structure of a function should make it clear where `if` statements are found.

For and While Loops

Loop constructs within an application are a very common place to find vulnerabilities. Recognizing them within binaries is often a key part of auditing. While it's not really possible to absolutely distinguish different types of loops from one another in compiled code, recognizing them functionally within binaries is usually pretty simple. They are generally characterized by a backwards branch or jump that leads to a repeated section of code. The following example illustrates a simple `while` loop and its compiled representation.

C Code:

```
char *ptr,*output,*outputend;

while(*ptr) {

          *output++ = *ptr++;

          if(output >= outputend)
                    break;

}
```

Compiled Representation (`ecx = ptr, edx = output, ebp+8 = outputend`):

```
mov al, [ecx]
test al, al
jz loop_end

mov [edx], al
inc ecx
inc edx

cmp edx, [ebp+8]
jae loop_end
jmp loop_begin
```

The code could have been functionally the same as a simple `for` loop, which makes it difficult to determine what kind of statement was in the original source code. However, the code's functionality is more important than its original state as source code, and loops like the one shown here are the source of many errors in closed source applications.

Switch Statements

`switch` statements are generally rather complex constructs in assembly code and can sometimes lead to compiled code that looks a little bit strange. Depending on the compiler and on the actual `switch` statement, the constructed code might vary quite a lot in structure.

A `switch` statement can be inefficiently broken down into several `if` statements, and some compilers will do this in certain situations. The statements themselves may be simpler to understand, and an auditor reading the code may never suspect that the code in question was ever anything but a group of sequential `if` statements.

If the `switch` cases are sequential, the compiler will often generate a jump table and index it with the `switch` case. This is a very efficient way to deal with switches with sequential cases, but is not always possible. An example might look like the following.

C Code:

```
int some_int,other_int;

switch(some_int) {

          case 0:
          other_int = 0;
          break;
     case 1:
          other_int = 10;
          break;
```

```
            case 2:
                 other_int = 30;
                 break;
            default:
                 other_int = 50;
                 break;
      }
```

Compiled Representation (`some_int = eax, other_int = ebx`):

```
            cmp eax, 2
            ja default_case

            jmp switch_jmp_table[eax*4];

case_0:
            xor ebx, ebx
            jmp end_switch
case_1:
            mov ebx, 10
            jmp end_switch
case 2:
            mov ebx, 30
            jmp end_switch
default_case:
            mov ebx, 50
end_switch:
```

At a read-only location in memory, the data table `switch_jmp_table` would be found containing the offsets of `case_0`, `case_1`, and `case_2` sequentially.

IDA Pro does a very good job of detecting `switch` statements constructed as just shown, and would indicate very accurately to the user which cases would be triggered by which values.

In the case where `switch` case values are not sequentially ordered, they cannot be easily or efficiently used as an index into a jump table. At this point, compilers often use a construct where the `switch` value is decremented or subtracted from until it reaches a zero value matching the `switch` case value. This allows the `switch` statement to efficiently deal with case values that are distant numerically. For example, if a `switch` statement was meant to deal with the case values 3, 4, 7, and 24 it might do so in the following manner (`EAX` = case value):

```
sub eax, 3
jz case_three
dec eax
jz case_four
sub eax, 3
jz case_seven
```

```
sub eax, 17
jz case_twenty_four
jmp default
```

This code would deal correctly with all the possible `switch` cases, as well as with default values, and is commonly seen in code generated by modern MSVC++ compilers.

memcpy-Like Code Constructs

Many compilers will optimize the `memcpy` library function into some simple assembly instructions that are much more efficient than a function call. This type of memory copy operation can potentially be the source of buffer overflow vulnerabilities and can easily be recognized within a disassembly. The set of instructions used is the following:

```
mov esi, source_address
mov ebx, ecx
shr ecx, 2              // length divided by four
mov edi, eax              // destination address
repe movsd              // copy four byte blocks
mov ecx, ebx
and ecx, 3              // remainder size
repe movsb              // copy it
```

In this case, the data is copied from the source register `ESI` to the destination register `EDI`. The data is copied in 4-byte blocks initially for the sake of speed by the instruction `repe movsd`. This copies `ECX` number of 4-byte blocks from `ESI` to `EDI`, which is why the length in `ECX` is divided by 4. The `repe movsb` instruction copies the remainder of the data.

`memset` is often optimized out in exactly the same manner using the `repe stosd` instruction with the `AL` register holding the character to `memset`. `memmove` is not optimized in this manner due to the possibility of overlapping data regions.

strlen-Like Code Constructs

Like `memcpy`, the `strlen` library function is often optimized into some simple x86 assembly instructions by certain compilers. Once again, this saves the overhead introduced by a function call. For those not familiar with compiler-generated code, the `strlen` code construction may seem strange at first. It generally looks much like the following:

```
mov edi, string
or ecx, 0xffffffff
xor eax, eax
```

```
repne scasb
not ecx
dec ecx
```

The result of these instructions is that the length of the string is stored in the ECX register. The `repne scasb` instruction scans from EDI for the character stored in the low byte of EAX, which is zero in this case. For each character this operation examines, it decrements ECX and increments EDI.

At the end of the `repne scasb` operation, when a null byte is found, EDI is pointing one character past the null byte and ECX is the negative string length minus two. A logical NOT of ECX, followed by a decrement results in the correct string length in ECX. It is often common to see a `sub edi, ecx` immediately following the `not ecx` instruction, which resets EDI back to its original position.

This code construct will be widely used in any code that handles string data; therefore, you should recognize it and understand how it works.

C++ Code Constructs

Most modern closed source code in operating systems and servers is written in C++. In many respects, this code is constructed in ways very similar to plain C code. The calling conventions are very close, and for compilers that support both C and C++, the same assembly code generation engines are used. However, in certain ways, auditing C++ code is different, and some special cases must be mentioned. In general, auditing binaries composed of C++ code is a little more difficult than that written in plain C; however, with some familiarization it's not that much of a leap.

The this Pointer

The `this` pointer refers to a specific instance of the class to which the current function (method) belongs. `this` must be passed to a function by its caller; however it is not passed as a normal function argument might be. Instead, the `this` pointer is passed in the ECX register. This calling convention used in C++ code is called `thiscall`. The following code shows an example of a function passing a class pointer to another function:

```
push    edi
push    esi
push    [ebp+arg_0]
lea     ecx, [ebx+5Ch]
call    ?ParseInput@HTTP_HEADERS@@QAEHPBDKPAK@Z
```

As you can see, a pointer is stored in the ECX register immediately before calling a function. In this case, the value stored in ECX is a pointer to a

HTTP_HEADERS object. Because the ECX register is quite volatile, the this pointer is often kept in another register after a function call, but it is almost always passed in the ECX register.

Reconstructing Class Definitions

When analyzing C++ code, having an understanding of the object structure is very helpful. This information can be difficult to gather if an auditor is not looking in the right places; many object structures are quite complex and contain nested objects.

The general methodology for reconstructing objects is to find all accesses relative to the object pointer, and therefore enumerate members of that class. In most cases, types for these members must be inferred or guessed, but in some cases you can determine whether they're used as arguments to known functions or in some other familiar context.

If you are trying to reconstruct an object manually, generally the best places to begin looking are the constructor and deconstructor for that class. These are functions that initialize and free up objects, and therefore are functions that generally access the most members of an object in any one place. These two functions will generally give a lot of information about a class, but not necessarily all the necessary information. It may be necessary to go to other methods in the class to get a more complete picture of the object.

If a program has symbol information, the constructor and destructor can be found quickly for any given application. They will have the notation Classname::Classname and Classname::~Classname. However, if they cannot be found by their name, they can often be recognized by their structure and where they are referenced. Constructors are often linear blocks of code that initialize a large number of structure members. They are generally void of comparisons or conditional jumps, and will often zero out structure members or large portions of a structure. Destructors will often free many structure members.

Halvar Flake has written an excellent tool (OBJRec) as a plug-in for IDA Pro that will take some of the tedium out of manually enumerating structure members for an object. This tool is available for download at www.openrce.org/downloads/details/39/OBJRec_for_x86.

vtables

The manner in which compilers deal with *virtual function tables* (vtables) can add much annoyance when you are auditing binaries. When a function from the vtable is called, it can be challenging to track down exactly where that call

is going without runtime analysis. For example, the following type of code will commonly be seen in compiled C++ code:

```
mov     eax, [esi]
lea     ecx, [ebp+var_8]
push    ebx
push    ecx
mov     ecx, esi
push    [ebp+arg_0]
push    [ebp+var_4]
call    dword ptr [eax+24h]
```

In this case, `ESI` contains the object pointer, and some function pointer is called from its `vtable`. To discover where this function call goes, it is necessary to know the structure of the `vtable`. The structure can usually be found in the constructor for the class. In this case, the following code is found in the constructor:

```
mov     dword ptr [esi], offset vtable
```

For this particular example, it was easy to locate the function call in the `vtable`, but quite often a function pointer from the `vtable` of a nested object is called. This type of situation can require quite a bit of work to resolve.

Quick but Useful Tidbits

Some of the points presented here are fairly obvious but nonetheless useful; if you don't already know them, you will miss several key points when auditing binaries.

- The return value for a function call is in the `EAX` register.
- The jump instructions `JL/JG` are used in signed comparisons.
- The jump instructions `JB/JA` are used in unsigned comparisons.
- `MOVSX` sign extends the destination register, whereas `MOVZX` zero-extends it.

Manual Binary Analysis

The time-proven manual method of reading a disassembly is still a very effective way of locating vulnerabilities in binaries. Depending on the quality of the code, an auditor might take different approaches when beginning a manual binary audit of an application.

Quick Examination of Library Calls

If the code quality is quite bad, it's usually productive to begin by looking for simple coding mistakes that can only really be found these days in closed source software. Simple calls to the traditional problem library functions should be examined with the hope of quickly finding a bug.

The traditional problem functions such as `strcpy`, `strcat`, `sprintf`, and all their derivatives should be examined. The Windows operating system has many variants of these functions, including versions for wide and ASCII character sets. For example, `strcpy`-like functions might include `strcpy`, `lstrcpyA`, `lstrcpyW`, `wcscpy`, and custom functions included in the application.

Another common source of problems on Windows is the function `MultiByteToWideChar`. The sixth argument to the function is the size of the destination buffer of wide characters. However, the size is specified as the number of wide characters, not the total size of the buffer. A common coding error has historically been to pass a `sizeof()` value as the sixth argument, and with each wide character being 2 bytes, this could potentially lead to a write twice the size of the destination buffer. This has led to security vulnerabilities in Microsoft's IIS Web server in the past.

Suspicious Loops and Write Instructions

When looking at simple API calls fails to turn up obvious security vulnerabilities, it's time to do some actual binary auditing. Like auditing in any other form, this involves gaining a certain understanding of the application being examined and reading relevant code sections. If an application has an obvious starting point for auditing, such as a routine that processes untrusted attacker-defined data, begin there and read onwards. If no such point is obvious, it's often possible to locate a good starting point by looking for protocol-specific information within the code. For example, a Web server parsing incoming requests will most likely begin by parsing the request method; searching the binary for common request methods can be a good way to locate a starting point for your audit.

Some common code constructs are indicative of dangerous code that may contain buffer overflows. Some examples follow.

A variable indexed write into a character array:

```
mov [ecx+edx], al
```

A variable indexed write to a local stack buffer:

```
mov [ebp+ecx-100h], al
```

A write to a pointer, followed by an increment of that pointer:

```
mov [edx], ax
inc edx
inc edx
```

A sign extended copy from an attacker-controlled buffer:

```
mov cl, [edx]
movsx eax, cl
```

An addition to or subtraction from a register containing attacker-controlled data (leading to an integer overflow):

```
mov eax, [edi]
add eax, 2
cp eax, 256
jae error
```

Value truncation as a result of being stored as a 16- or 8-bit integer:

```
push edi
call strlen
add esp, 4
mov word ptr [ebp-4], ax
```

By recognizing these code constructs and many like them, it should be possible to locate a wide range of memory corruption vulnerabilities within binaries.

Higher-Level Understanding and Logic Bugs

Although most vulnerabilities discovered today are memory corruption issues, some bugs are completely unrelated to memory corruption and are simply logic flaws in an application. A good example of this is the IIS double-decode flaw discovered several years ago. These types of vulnerabilities are admittedly hard to discover by binary analysis and require either luck or a very good understanding of an application to locate. There's obviously no particular way to find these types of bugs, but in general the best way to find these types of issues is to examine in depth the code that accesses any critical resources based on user-specified data. It helps to have creativity and an open mind when looking for these types of bugs, but a lot of spare time is an obvious requirement.

Graphical Analysis of Binaries

Some functions, especially those that are very large or complex, make more sense when displayed graphically. On a graph, certain complex loops become easily recognizable; it is much more difficult to confuse code sections when you view them as a part of a large graph rather than as a linear disassembly. IDA Pro can generate graphs of any given function, with each node being a continuous section of code. Nodes are linked by branches or execution flow, and each node is pretty much guaranteed to be executed as a contiguous block of code. Most graphs are too large to view comfortably as a whole on most monitors. Consequently, it is quite useful to print out hard copies of function graphs, which often span multiple pages, and then analyze them on paper.

The graphing engine of IDA Pro will misinterpret some compiler-generated code, however. For example, it will not include code fragments generated by MSVC++ in a function graph, which leads to incomplete and often useless graphs. A graphing plug-in for IDA Pro created by Halvar Flake will properly include these code fragments, creating complete and usable graphs for MSVC++ compiled code.

Manual Decompilation

Some functions are too large to be analyzed properly in a disassembly. Others contain very complex loop constructs whose security cannot easily be determined by traditional binary analysis. An alternative that may work in these cases is manual decompilation.

An accurate decompilation will obviously be easier to audit than a disassembly, but much care must be taken to ensure that any work done is accurate. There's little point in auditing a decompilation with errors. It is helpful to completely set aside the security auditing mindset (if that is possible), and just create a source code representation of the function. In that way, the decompilation is less likely to be tainted by wishful thinking.

Binary Vulnerability Examples

Let's look at a concrete example of applying binary analysis to search for a security hole.

Microsoft SQL Server Bugs

Two of the coauthors of this book, David Litchfield and Dave Aitel, discovered some very serious vulnerabilities in Microsoft SQL Server. SQL Server bugs have been used in such worms as the Slammer worm and have had far-reaching

consequences for network security. A quick examination of the core network services of the unpatched SQL Server network library quickly reveals the source of these bugs.

The vulnerability discovered by Litchfield is the result of an unchecked `sprintf` call in the packet processing routines of the SQL resolution service.

```
mov     edx, [ebp+var_24C8]
push    edx
push    offset aSoftwareMic_17 ; "SOFTWARE\\Microsoft\\Microsoft SQL
Server"...
push    offset aSSMssqlserverC ; "%s%s\\MSSQLServer\\CurrentVersion"
lea     eax, [ebp+var_84]
push    eax
call    ds:sprintf
add     esp, 10h
```

In this case, `var_24C8` contains packet data read off the network and can be close to 1024 bytes, and `var_84` is a 128-byte local stack buffer. The consequences of this operation are obvious, and it is extremely obvious when examining a binary.

The SQL Server Hello vulnerability discovered by Dave Aitel is also a result of an unchecked string operation, in this case simply `strcpy`.

```
mov     eax, [ebp+arg_4]
add     eax, [ebp+var_218]
push    eax
lea     ecx, [ebp+var_214]
push    ecx
call    strcpy
add     esp, 8
```

The destination buffer, `var_214`, is a 512-byte local stack buffer, and the source string is simply packet data. Once again, rather simplistic bugs tend to persist longer in closed source software that is widely available only as a binary.

LSD's RPC-DCOM Vulnerability

The infamous and widely exploited vulnerability discovered by The Last Stage of Delirium (LSD) in RPC-DCOM interfaces was the result of an unchecked string copy loop when parsing server names out of `UNC` path names. Once again, when located in `rpcss.dll`, the memory copy loop is quite obviously a security risk.

```
mov     ax, [eax+4]
cmp     ax, '\'
jz      short loc_761AE698
```

```
sub     edx, ecx
loc_761AE689:
mov     [ecx], ax
inc     ecx
inc     ecx
mov     ax, [ecx+edx]
cmp     ax, '\'
jnz     short loc_761AE689
```

The UNC path name takes the format of \\server\share\path, and is transmitted as a wide character string. The copy loop in the code above skips the first 4 bytes (two backslash characters) and copies into the destination buffer until a terminating backslash is seen, without any bounds-checking. Loop constructs like this are quite commonly the source of memory corruption vulnerabilities.

IIS WebDAV Vulnerability

The IIS WebDAV vulnerability disclosed in the Microsoft Security Bulletin MS03-007 was a somewhat uncommon case, in which an 0day exploit was uncovered in the wild and resulted in a security patch. This vulnerability was not discovered by security researchers, but rather by a third party with malicious intentions.

The actual vulnerability was the result of a 16-bit integer wrap that can commonly occur in the core Windows runtime library string functions. The data storage types used by functions such as RtlInitUnicodeString and RtlInitAnsiString have a length field that is a 16-bit unsigned value. If strings are passed to these functions that exceed 65,535 characters in length, the length field will wrap, and result in a string that appears to be very small. The IIS WebDAV vulnerability was the result of passing a string longer than 64K to RtlDosPathNameToNtPathName_U, resulting in a wrap in the length field of the Unicode string and a very large string having a small length field. This particular bug is rather subtle and was most likely not discovered by binary auditing; however, with practice and time these types of issues can be found.

The basic data structure for a Unicode or ANSI string looks like the following:

```
typedef struct UNICODE_STRING {
          unsigned short length;
          unsigned short maximum_length;
          wchar *data;
};
```

The code within RtlInitUnicodeString looks like the following:

```
mov     edi, [esp+arg_4]
mov     edx, [esp+arg_0]
mov     dword ptr [edx], 0
mov     [edx+4], edi
```

```
or      edi, edi
jz      short loc_77F83C98
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasw
not     ecx
shl     ecx, 1
mov     [edx+2], cx                    // possible truncation here
dec     ecx
dec     ecx
mov     [edx], cx                    // possible truncation here
```

In this case, the wide string length is determined by `repne scasw` and multiplied by two, with the result stored in a 16-bit structure field.

Within a function called by `RtlDosPathNameToNtPathName_U`, the following code is seen:

```
mov     dx, [ebp+var_30]
movzx   esi, dx
mov     eax, [ebp+var_28]
lea     ecx, [eax+esi]
mov     [ebp+var_5C], ecx
cmp     ecx, [ebp+arg_4]
jnb     loc_77F8E771
```

In this case, `var_28` is another string length, and `var_30` is the attacker's long `UNICODE_STRING` structure with a truncated 16-bit length value. If the sum of the two string lengths is less than `arg_4`, which is the length of the destination stack buffer, then the two strings are copied into the destination buffer. Because one of these strings is significantly larger than the stack space reserved, an overflow occurs. The character copying loop is fairly standard and easily recognizable. It looks like the following:

```
mov     [ecx], dx
add     ecx, ebx
mov     [ebp+var_34], ecx
add     [ebp+var_60], ebx
loc_77F8AE6E:
mov     edx, [ebp+var_60]
mov     dx, [edx]
test    dx, dx
jz      short loc_77F8AE42
cmp     dx, ax
jz      short loc_77F8AE42
cmp     dx, '/'
jz      short loc_77F8AE42
cmp     dx, '.'
jnz     short loc_77F8AE63
jmp     loc_77F8B27F
```

In this case, the string is copied into the destination buffer until a dot (.), forward slash (/), or a null byte is encountered. Although this particular vulnerability resulted in writing up to the top of the stack and crashing the thread, an SEH exception handler pointer was overwritten, which resulted in arbitrary code execution.

Conclusion

Many of the vulnerabilities discovered in closed source products are those that were weeded out of open source software years ago. Because of some of the challenges inherent to binary auditing, most of this software is under-audited or only fuzz-tested, and many vulnerabilities still lurk unnoticed. Although there is a bit of overhead work involved in binary auditing, it is not much more difficult than source-code auditing and simply requires a little more time. As time passes, many of the more obvious vulnerabilities will be fuzz-tested out of commercial software, and to find more subtle bugs, an auditor will have to do more in-depth binary analysis. Binary auditing may eventually become as commonplace as source code review—it is definitely a field in which much work still needs to be done.