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.