This project was developed by Randy Bryant; it was adapted for Virginia Tech by Godmar Back (gback@cs.vt.edu).

**Minimum requirement:** To obtain a passing grade in CS 3214, we expect that by the end of the semester your group was successful in igniting at least Level 2, which is the “Firecracker” phase of the bomb.

**Introduction**

This assignment will help you develop a detailed understanding of IA-32 calling conventions and stack organization. It involves applying a series of *buffer overflow attacks* on an executable file bufbomb provided to you.

**Note:** In this project, you will gain firsthand experience with one of the methods commonly used to exploit security weaknesses in operating systems and network servers. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of this form of security weakness so that you can avoid it when you write system or application code. We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources. There are criminal statutes governing such activities.

**Logistics**

You may work in a group of up to two people in solving the problems for this project. The only “hand-in” will be an automated logging of your successful attacks. Any clarifications and revisions to the project will be posted on the course web page.

We generated the buffer bomb executable using gcc’s `-m32` flag, so all code produced by the compiler follows IA-32 rules, even if the host is an x86-64 system.
Hand Out Instructions

You can obtain your buffer bomb by pointing your Web browser at:

\[ \text{http://courses.cs.vt.edu/~cs3214/fall2012/buflab/buflab-handout.tar} \]

The server will return a tar file called \textit{buflab-handout.tar} to your browser. Start by copying \texttt{buflab-handout.tar} to a (protected) directory in which you plan to do your work. It is okay for each group member to maintain a copy in their own respective directory. Then give the command \texttt{\textasciitilde{}tar xvf buflab-handout.tar"}. This will create a directory called \texttt{buflab-handout} containing the following three executable files:

- \texttt{bufbomb}: The buffer bomb program you will attack.
- \texttt{makecookie}: A utility that generates a “cookie” based on your teamid.
- \texttt{hex2raw}: A utility to help convert between string formats.

In the following instructions, we will assume that you have copied the three programs to a protected local directory, and that you are executing them in that local directory.

Team IDs and Cookies

Phases of this project will require a slightly different solution from each group. The correct solution will be based on your teamid. The teamid is the concatenation of your SLO IDs using a + in between. If you’re in a group by yourselves, simply use your SLO ID. While the order of the two SLO IDs in a teamid does not matter, you must always choose the same ordering for the IDs. Teams “rnikola+butta” and “butta+rnikola” are considered distinct. \textbf{You must follow this scheme for generating your teamid}. Our grading program will only give credit to those people whose SLO IDs can be extracted from the teamids.

A \textit{cookie} is a string of eight hexadecimal digits that is (with high probability) unique to your team id. You can generate your cookie with the \texttt{makecookie} program giving your teamid as the argument. For example:

\begin{verbatim}
unix> ./makecookie rnikola+butta
0x1005b2b7
\end{verbatim}

In four of your five buffer attacks, your objective will be to make your cookie show up in places where it ordinarily would not.

The \textbf{BUF BOMB} Program

The \textbf{BUF BOMB} program reads a string from standard input. It does so with the function \texttt{getbuf} defined below:
int getbuf()
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}

The function Get is similar to the (long-deprecated) standard library function gets—it reads a string from standard input (terminated by ‘\n’ or end-of-file) and stores it (along with a null terminator) at the specified destination. In this code, you can see that the destination is an array buf having sufficient space for 32 characters.

Get (and gets) grabs a string off the standard input stream and stores it into its destination address (in this case buf). However, Get() has no way of determining whether buf is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

If the string typed by the user to getbuf is no more than 31 characters long, it is clear that getbuf will return 1, as shown by the following execution example:

unix> ./bufbomb -u rnikola+butta
Type string: I love CS3214.
Dud: getbuf returned 0x1

Typically an error occurs if we type a longer string:

unix> ./bufbomb -u rnikola+butta
Type string: It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!

As the error message indicates, overrunning the buffer typically causes the program state to be corrupted, leading to a memory access error. Your task is to be more clever with the strings you feed BUFBOMB so that it does more interesting things. These are called exploit strings.

BUFBOMB takes several different command line arguments:

-u teamid: Operate the bomb for the indicated teamid. You should always provide this argument for several reasons:

- It is required to submit your successful attacks to the grading server.
- BUFBOMB determines the cookie you will be using based on your teamid, as does the program MAKECOOKIE.
- We have built features into BUFBOMB so that some of the key stack addresses you will need to use depend on your teamid’s cookie.

-h: Print list of possible command line arguments.
-n: Operate in “Nitro” mode, as is used in Level 4 below.

-s: Submit your solution exploit string to the grading server.

At this point, you should think about the x86 stack structure a bit and figure out what entries of
the stack you will be targeting. You may also want to think about exactly why the last example
created a segmentation fault, although this is less clear.

Your exploit strings will typically contain byte values that do not correspond to the ASCII values
for printing characters. The program HEX2RAW can help you generate these raw strings. It takes
as input a hex-formatted string. In this format, each byte value is represented by two hex digits.
For example, the string “012345” could be entered in hex format as “30 31 32 33 34 35.”
(Recall that the ASCII code for decimal digit $x$ is $0x3x$.)

The hex characters you pass HEX2RAW should be separated by whitespace (blanks or newlines). I
recommend separating different parts of your exploit string with newlines while you’re working
on it. HEX2RAW also supports C-style block comments, so you can mark off sections of your
exploit string. For example:

```bash
bf 66 7b 32 78 /* mov $0x78327b66,%edi */
```

Be sure to leave space around both the starting and ending comment strings (/*, */ so they will
be properly ignored.

If you generate a hex-formatted exploit string in the file exploit.txt, you can apply the raw
string to BUFBOMB in several different ways:

1. You can set up a series of pipes to pass the string through HEX2RAW.

```bash
unix> cat exploit.txt | ./hex2raw | ./bufbomb -u rnikola+butta
```

2. You can store the raw string in a file and use I/O redirection to supply it to BUFBOMB:

```bash
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./bufbomb -u rnikola+butta < exploit-raw.txt
```

This approach can also be used when running BUFBOMB from within GDB:

```bash
unix> gdb bufbomb
(gdb) run -u rnikola+butta < exploit-raw.txt
```

**Important points:**

- Your exploit string must not contain byte value $0x0A$ at any intermediate position, since this
  is the ASCII code for newline (‘\n’). When Gets encounters this byte, it will assume you
  intended to terminate the string.
• **HEX2RAW** expects two-digit hex values separated by a whitespace. So if you want to create a byte with a hex value of 0, you need to specify 00. To create the word 0xDEADBEEF you should pass DE AD BE EF to **HEX2RAW**.

When you have correctly solved one of the levels, say level 0:

```
../hex2raw < smoke-rnikola+butta.txt | ../bufbomb -u rnikola+butta
UserId: rnikola+butta
Cookie: 0x1005b2b7
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

then can submit your solution to the grading server using the `-s` option:

```
../hex2raw < smoke-rnikola+butta.txt | ./bufbomb -u rnikola+butta -s
UserId: rnikola+butta
Cookie: 0x1005b2b7
Type string:Smoke!: You called smoke()
VALID
Sent exploit string to server to be validated.
NICE JOB!
```

The server will test your exploit string to make sure it really works, and it will update the Buffer Lab scoreboard page indicating that your teamid (listed by your cookie for anonymity) has completed this level.

You can view the scoreboard by pointing your Web browser at [http://courses.cs.vt.edu/~cs3214/fall2012/buflab/buflab-scoreboard.html](http://courses.cs.vt.edu/~cs3214/fall2012/buflab/buflab-scoreboard.html)

Unlike the Bomb Lab, there is no penalty for making mistakes in this project. Feel free to fire away at **BUFBOMB** with any string you like. Of course, you shouldn’t brute force this project either, since it would take longer than you have to do the assignment.

**IMPORTANT NOTE:** You can work on your buffer bomb on any Linux machine, but in order to submit your solution, you will need to be running on one of the machines that are part of the rlogin.cs.vt.edu cluster.

**Level 0: Candle 10 pts**

The function `getbuf` is called within **BUFBOMB** by a function `test` having the following C code:

```c
1 void test()
2 {
3     int val;
4     volatile int local = 0xdeadbeef;
```
val = getbuf();

/* Check for corrupted stack */
if (local != 0xdeadbeef) {
    printf("Sabotaged!: the stack has been corrupted\n");
} else if (val == cookie) {
    printf("Boom!: getbuf returned 0x%x\n", val);
    validate(3);
} else {
    printf("Dud: getbuf returned 0x%x\n", val);
}
}

When getbuf executes its return statement (line 5 of getbuf), the program ordinarily resumes execution within function test (at line 7 of this function). We want to change this behavior. Within the file bufbomb, there is a function smoke having the following C code:

void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}

Your task is to get BUFBOMB to execute the code for smoke when getbuf executes its return statement, rather than returning to test. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since smoke causes the program to exit directly.

Some Advice:

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of BUFBOMB. Use objdump -d to get this disassembled version.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of getbuf to make sure it is doing the right thing.
- The placement of buf within the stack frame for getbuf depends on which version of GCC was used to compile bufbomb, so you will have to read some assembly to figure out its true location.
Level 1: Sparkler 15 pts

Within the file `bufbomb` there is also a function `fizz` having the following C code:

```c
void fizz(int val)
{
  if (val == cookie) {
    printf("Fizz!: You called fizz(0x%x)\n", val);
    validate(1);
  } else
    printf("Misfire: You called fizz(0x%x)\n", val);
  exit(0);
}
```

Similar to Level 0, your task is to get BUFBOMB to execute the code for `fizz` rather than returning to `test`. In this case, however, you must make it appear to `fizz` as if you have passed your cookie as its argument. How can you do this?

**Some Advice:**

- Note that the program won’t really call `fizz`—it will simply execute its code. This has important implications for where on the stack you want to place your cookie.

Level 2: Firecracker 15 pts

A much more sophisticated form of buffer attack involves supplying a string that encodes actual machine instructions. The exploit string then overwrites the return pointer with the starting address of these instructions on the stack. When the calling function (in this case `getbuf`) executes its `ret` instruction, the program will start executing the instructions on the stack rather than returning. With this form of attack, you can get the program to do almost anything. The code you place on the stack is called the *exploit* code. This style of attack is tricky, though, because you must get machine code onto the stack and set the return pointer to the start of this code.

Within the file `bufbomb` there is a function `bang` having the following C code:

```c
int global_value = 0;

void bang(int val)
{
  if (global_value == cookie) {
    printf("Bang!: You set global_value to 0x%x\n", global_value);
    validate(2);
  } else
    printf("Misfire: global_value = 0x%x\n", global_value);
  exit(0);
}
```
Similar to Levels 0 and 1, your task is to get BUF Bomb to execute the code for bang rather than returning to test. Before this, however, you must set global variable global_value to your teamid’s cookie. Your exploit code should set global_value, push the address of bang on the stack, and then execute a ret instruction to cause a jump to the code for bang.

**Some Advice:**

- You can use GDB to get the information you need to construct your exploit string. Set a breakpoint within getbuf or Gets and run to this breakpoint. Determine parameters such as the address of global_value and the location of the buffer.

- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with gcc -m32 -c and disassemble it with objdump -d. You should be able to get the exact byte sequence that you will type at the prompt. (A brief example of how to do this is included at the end of this writeup.)

- Keep in mind that your exploit string depends on your machine, your compiler, and even your teamid’s cookie. You must do this project on the machines of the rlogin cluster. Make sure you include the proper teamid on the command line to BUF Bomb.

- Watch your use of address modes when writing assembly code. Note that movl $0x4, %eax moves the value 0x00000004 into register %eax; whereas movl 0x4, %eax moves the value at memory location 0x00000004 into %eax. Since that memory location is usually undefined, the second instruction will cause a segfault!

- Do not attempt to use either a jmp or a call instruction to jump to the code for bang. These instructions uses PC-relative addressing, which is very tricky to set up correctly. Instead, push an address on the stack and use the ret instruction.

**Level 3: Dynamite 20 pts**

Our preceding attacks have all caused the program to jump to the code for some other function, which then causes the program to exit. As a result, it was acceptable to use exploit strings that corrupt the stack, overwriting saved values.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that changes the program’s register/memory state, but makes the program return to the original calling function (test in this case). The calling function is oblivious to the attack. This style of attack is tricky, though, since you must: 1) get machine code onto the stack, 2) set the return pointer to the start of this code, and 3) undo any noticeable corruptions made to the stack state.

Your job for this level is to supply an exploit string that will cause getbuf to return your cookie back to test, rather than the value 1. You can see in the code for test that this will cause the
program to go “Boom!.” Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a ret instruction to really return to test.

**Some Advice:**

- You can use GDB to get the information you need to construct your exploit string. Set a breakpoint within `getbuf` and run to this breakpoint. Determine parameters such as the saved return address.

- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with GCC and disassemble it with OBJDUMP. You should be able to get the exact byte sequence that you will type at the prompt. (A brief example of how to do this is included at the end of this writeup.)

- Keep in mind that your exploit string depends on your machine, your compiler, and even your teamid’s cookie. Do all of your work on the machines assigned by your instructor, and make sure you include the proper teamid on the command line to BUFBOMB.

Once you complete this level, pause to reflect on what you have accomplished. You caused a program to execute machine code of your own design. You have done so in a sufficiently stealthy way that the program did not realize that anything was amiss.

**Level 4: Nitroglycerin 15 pts**

Please note: You’ll need to use the “-n,” command-line flag in order to run this stage.

From one run to another, especially by different users, the exact stack positions used by a given procedure will vary. One reason for this variation is that the values of all environment variables are placed near the base of the stack when a program starts executing. Environment variables are stored as strings, requiring different amounts of storage depending on their values. Thus, the stack space allocated for a given user depends on the settings of his or her environment variables. Stack positions also differ when running a program under GDB, since GDB uses stack space for some of its own state. Lastly, modern versions of Linux deliberate randomize the location of the stack from run to run to make the job of attackers seeking to exploit buffer overflow vulnerabilities such as the one contained in this bomb even harder.

In the code that calls `getbuf`, we have incorporated features that stabilize the stack, so that the position of `getbuf`’s stack frame will be consistent between runs. This made it possible for you to write an exploit string knowing the exact starting address of `buf`. If you tried to use such an exploit on a normal program, you would find that it works some times, but it causes segmentation faults at other times. Hence the name “dynamite”—an explosive developed by Alfred Nobel that contains stabilizing elements to make it less prone to unexpected explosions.
For this level, we have gone the opposite direction, making the stack positions even less stable than they normally are. Hence the name “nitroglycerin”—an explosive that is notoriously unstable.

When you run BUF BOMB with the command line flag “-n,” it will run in “Nitro” mode. Rather than calling the function `getbuf`, the program calls a slightly different function `getbufn`:

```c
int getbufn()
{
    char buf[KABOOM_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

This function is similar to `getbuf`, except that it has a buffer of 512 characters. You will need this additional space to create a reliable exploit. The code that calls `getbufn` first allocates a random amount of storage on the stack (using library function `alloca`) that ranges between 0 and 255 bytes. Thus, if you were to sample the value of `%ebp` during two successive executions of `getbufn`, you would find they differ by as much as ±127.

In addition, when run in Nitro mode, BUF BOMB requires you to supply your string 5 times, and it will execute `getbufn` 5 times, each with a different stack offset. Your exploit string must make it return your cookie each of these times.

Your task is identical to the task for the Dynamite level. Once again, your job for this level is to supply an exploit string that will cause `getbufn` to return your cookie back to test, rather than the value 1. You can see in the code for test that this will cause the program to go “KABOOM!.” Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `testn`.

**Some Advice:**

- You can use the program HEX2RAW to send multiple copies of your exploit string. If you have a single copy in the file `exploit.txt`, then you can use the following command:

  ```
  unix> cat exploit.txt | ./hex2raw -n | ./bufbomb -n -u rnikola+butta
  ```

  You must use the same string for all 5 executions of `getbufn`. Otherwise it will fail the testing code used by our grading server.

- The trick is to make use of the `nop` instruction. It is encoded with a single byte (code 0x90). It may be useful to read about “nop sleds” on page 262 of the CS:APP2e textbook.

**Logistical Notes**

Handin occurs to the grading server whenever you correctly solve a level and use the `-s` option. Upon receiving your solution, the server will validate your string and update the Buffer Lab
scoreboard Web page, which you can view by pointing your Web browser at http://courses.
cs.vt.edu/~cs3214/fall2012/buflab/buflab-scoreboard.html

You should be sure to check this page after your submission to make sure your string has
been validated. (If you really solved the level, your string should be valid.)

Note that each level is graded individually. You do not need to do them in the specified order, but
you will get credit only for the levels for which the server receives a valid message. You can check
the Buffer Lab scoreboard to see how far you’ve gotten.

The grading server creates the scoreboard by using the latest results it has for each phase.

Good luck and have fun!

Generating Byte Codes

Using GCC as an assembler and OBJDUMP as a disassembler makes it convenient to generate the
byte codes for instruction sequences. For example, suppose we write a file example.S containing
the following assembly code:

```
# Example of hand-generated assembly code
push $0xabcdef # Push value onto stack
add $17,%eax # Add 17 to %eax
.align 4 # Following will be aligned on multiple of 4
.long 0xfedcba98 # A 4-byte constant
```

The code can contain a mixture of instructions and data. Anything to the right of a ‘#’ character is
a comment.

We can now assemble and disassemble this file:

```
unix> gcc -m32 -c example.S
unix> objdump -d example.o > example.d
```

The generated file example.d contains the following lines

```
0: 68 ef cd ab 00 push $0xabcdef
5: 83 c0 11 add $0x11,%eax
8: 98 cwtl
9: ba .byte 0xba
a: dc fe fdivr %st,%st(6)
```

Each line shows a single instruction. The number on the left indicates the starting address (starting
with 0), while the hex digits after the ‘:’ character indicate the byte codes for the instruction. Thus,
we can see that the instruction push $0xABCDEF has hex-formatted byte code 68 ef cd ab
00.
Starting at address 8, the disassembler gets confused. It tries to interpret the bytes in the file example.o as instructions, but these bytes actually correspond to data. Note, however, that if we read off the 4 bytes starting at address 8 we get: 98 ba dc fe. This is a byte-reversed version of the data word 0xFEDCBA98. This byte reversal represents the proper way to supply the bytes as a string, since a little endian machine lists the least significant byte first.

Finally, we can read off the byte sequence for our code as:

```
68 ef cd ab 00 83 c0 11 98 ba dc fe
```

This string can then be passed through HEX2RAW to generate a proper input string we can give to BUFBOMB. Alternatively, we can edit example.d to look like this:

```
68 ef cd ab 00 /* push $0xabcdef */
83 c0 11 /* add $0x11,%eax */
98
ba dc fe
```

which is also a valid input we can pass through HEX2RAW before sending to BUFBOMB.