

## CSE 361S Intro to Systems Software Lab Assignment #5

Due: Thursday, November 10, 2011.

In this lab, you will mount a buffer overflow attack on your own program. We do not condone using this or any other form of attack to gain unauthorized access to a system. Rather, by doing this exercise, I hope you will learn a lot about how to defend against such attacks.

You may work in a group of up to two people in solving the problems in this lab.

Download the file `bufbomb.c` from the class web site and compile it to create an executable program. In `bufbomb.c` you will find the following functions:

```
int getbuf() {
    char buf[16];

    getxs(buf);
    return 1;
}

void test() {
    int val;

    printf("Type Hex String: ");
    val = getbuf();
    printf("getbuf returned 0x%x\n", val);
}
```

The function `getxs` (also in `bufbomb.c`) is similar to the library `gets`, except that it reads characters encoded as pairs of hex digits. For example, to give it a string "0123," the user would type in the string "30 31 32 33." The function ignores blank characters. Recall that decimal digit  $x$  has ASCII representation `0x3x`.

A typical execution of the program is as follows:

```
prompt> ./bufbomb
Type Hex String: 30 31 32 33
getbuf returned 0x1
```

Looking at the code for the `getbuf` function, it seems quite apparent that it will return value 1 whenever it is called. It appears as if the call to `getxs` has no effect. Your task is to make `getbuf` return `0xdeadbeef` to `test`, simply by typing an appropriate hexadecimal string to the prompt.

You will get there in several stages.

## Stage 1

If the string typed by the user to `getbuf` is no more than 15 characters long, it is clear that `getbuf` will return 1. Typically an error occurs if we type a longer string:

```
prompt> ./bufbomb
Type Hex String: 30 31 32 33 30 31 32 33 30 31 32 33 30 31
Ouch!: You caused a segmentation fault!
Better luck next time
```

As the error 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.

When `getbuf` executes its return statement, the program ordinarily resumes execution within function `test`. In the file `bufbomb.c`, there is a function `smoke` having the following C code:

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

Your task is to get `bufbomb` to execute the code for `smoke` when `getbuf` executes its `ret` instruction, rather than returning to `test`. You can do this by supplying an exploit string that overwrites the stored return pointer in the stack frame for `getbuf` with the address of the first instruction in `smoke`. Note that your exploit string may also corrupt other parts of the stack state, but this will not cause a problem, since `smoke` causes the program to exit directly.

Helpful hints:

- Use `objdump` to create a disassembled version of `bufbomb`. Study this closely to determine how the stack frame for `getbuf` is organized and how overflowing the buffer will alter the saved program state.
- Run the program under `gdb`. Set a breakpoint within `getbuf` and run to this breakpoint. Determine such parameters as the value of `%ebp` and the saved value of any state that will be overwritten when you overflow the buffer.
- 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.
- Keep in mind that your attack is very machine and compiler specific. You may need to alter your string when running on a different machine or with a different version of `gcc`.
- The placement of `buf` within the stack frame for `getbuf` depends upon which version of `gcc` was used to compile `bufbomb`. You will need to pad the beginning of your exploit string with the proper number of bytes to overwrite the return point. The values of these bytes can be arbitrary.

## Stage 2

In the file `bufbomb.c` there is also a function `fizz` having the following C code:

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

Similar to stage 1, 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 it the argument `0xdeadbeef`. You can do this by encoding `0xdeadbeef` in the appropriate place within your exploit string.

Helpful hints:

- 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 the argument.
- Finish the above by Nov. 3, as there is plenty of work still to be completed.

## Stage 3

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. When the calling function (in this case `getbuf`) executes its return 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.c` there is a function `bang` having the following C code:

```
int global_value = 0;

void bang() {
    if (global_value == 0xdeadbeef) {
        printf("Bang!: You set global_value to 0x%x\n",
            global_value);
    }
    else {
        printf("Misfire: global_value = 0x%x\n",
            global_value);
    }
    exit(0);
}
```

Similar to stages 1 and 2, your task is to get `bufbomb` to execute the code for `bang` rather than returning to `test`. Before this, however, you must set global variable `global_value` to `0xdeadbeef`. 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`.

Helpful hints:

- 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 address of `global_value` and the location of the buffer. Note that the location of the stack (and therefore the buffer) will be different each execution due to defense mechanisms built into the OS. It will be necessary to alter your exploit string to adjust for this, so each execution must be within `gdb` with a breakpoint that enables you to examine the stack location.
- 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. `Objdump` will produce some pretty strange looking assembly instructions when it tries to disassemble the data in your file, but the hexadecimal byte sequence should be correct.
- Keep in mind that your exploit string depends upon your machine and your compiler. Use the command `execstack -s bufbomb` (where `bufbomb`

is the name of your executable) to enable execution of code on the stack. Since `execstack` will not operate across a network file system, copy `bufbomb` to the local machine (e.g., to `/tmp`) and execute `execstack` on that copy. You can then copy the altered version of `bufbomb` back to your working directory.

- Our solution requires 16 bytes of exploit code. Fortunately, there are 20 bytes of available space on the stack, because we can overwrite the stored value of `%ebp`. This stack corruption will not cause any problems, since `bang` causes the program to exit directly.
- Watch your use of address modes when writing assembly code. Note that the instruction `movl $0x4, %eax` moves the value `0x00000004` into register `%eax`; whereas `movl 0x4, %eax` moves the value at memory location `0x00000004` into `%eax`. These are *not* the same thing!
- Do not attempt to use a `jmp` or `call` instruction to jump to the code for `bang`. These instructions use PC relative addressing, which is very tricky to set up correctly. Instead, push an address on the stack and use the `ret` instruction.

#### Stage 4

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 the saved value of register `%ebp` and the return pointer.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that patches up the stack and 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 the corruptions made to the stack state.

Your job for this stage is to supply an exploit string that will cause `getbuf` to return `0xdeadbeef` back to `test`, rather than the value `1`. Your exploit code should set `0xdeadbeef` 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`.

Helpful hints:

- In order to overwrite the return pointer, you must also overwrite the saved value of `%ebp`. However, it is important that this value is correctly restored before you return to `test`. You can do this by either: 1) making sure that your exploit string contains the correct value of the saved `%ebp` in the correct position, so that it never gets corrupted, or 2) restore the correct value as part of your exploit code.

- 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 and the saved value of `%ebp`.
- Again, let tools such as `gcc` and `objdump` do all the work of generating a byte encoding of the instructions.
- Keep in mind that your exploit string depends upon your machine and your compiler.

Once you complete this stage, 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.

## 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 */
    pushl $0x89abcdef    /* push value onto stack    */
    addl $17,%eax        /* add 17 to %eax          */
    .align 4             /* align to 4-byte boundary */
    .long 0xfedcba98     /* a 4-byte constant      */
    .long 0x00000000     /* padding                 */
```

The code can contain a mixture of instructions and data. We have added an extra word of all 0s to work around a shortcoming in `objdump` to be described shortly.

We can now assemble and disassemble this file:

```
prompt> gcc -c example.s
prompt> objdump -d example.o > example.d
```

The generated file `example.d` contains the following lines:

```
0:    68 ef cd ab 89          push   $0x89abcdef
5:    83 c0 11                add    $0x11,%eax
8:    98                      cwtl
9:    ba dc fe 00 00        mov   $0xfedc,%edx
```

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. This we can see that the instruction `pushl $0x89ABCDEF` has hex-formatted byte code `68 ef cd ab 89`.

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 sequence, since a little-endian machine stores the least significant byte first. Note also that it only generated two of the four bytes at the end with value `00`. Had we not added the padding, `objdump` gets even more confused and does not emit all of the bytes we want.

Finally, we can read off the byte sequence for our code (omitting the final 0s) as:

```
68 ef cd ab 89 83 c0 11 98 ba dc fe
```

## Lab Report

This lab report should be a single file, named `<names>_lab5.pdf` (follow the naming conventions we've been using all semester), which includes the following:

1. Names and email addresses for each team member.
2. Your exploit string for each of the four stages of the lab.
3. Sufficient explanation so that we can clearly follow how you developed each of the exploit strings. Take hints here from the hints that we provide. Include the following types of information:
  - a. output from the `objdump` program,
  - b. descriptions of the stack prior to and after your exploit string insertion,
  - c. what it is you learned when you put a `gdb` breakpoint in `getbuf` and looked around,
  - d. step-by-step, what was your reasoning,
  - e. your source and `objdump` output for your exploit codes, and
  - f. anything else that will help us believe you didn't just copy the exploit strings from your neighbor (which is just as unethical as gaining unauthorized access to systems via buffer overflow attacks).

This lab report should be uploaded to telexis with a filename that includes the entire team's names as well as the lab number (e.g., `SmithA_JonesB_lab5.pdf`).