

HOMWORK 2 KEY

ENTS 689i Network Immunity
Fall 2008

QUESTION 1: (5 pts) Briefly describe (no more than one paragraph) the low-level flaw that has been inserted into httpd-broken.c.

The `get_line()` function has been modified to no longer take buffer size into account when filling the buffer from the network socket. Since `get_line()` will fill the buffer until it sees a newline (`\n`) character, a buffer overflow is possible when there is more data than the size of the passed in buffer.

QUESTION 2: (5 pts) Briefly (no more than one paragraph) describe the buffer overflow test that the fuzzer performed. What patterns of requests did it make and why?

The fuzzer is sending requests that are increasing in size with each new request. It is doing so in order to find the first buffer increase that results in the program crashing or not responding. As some students noticed, the size of the request is varying close to powers of 2, with some edge cases checked for common lengths such as 1023, 1024, etc. You didn't need to say anything about the actual contents of the request, although the question may have been a little ambiguous in that regard.

QUESTION 3: (10 pts) What happened? Why did the program crash? What is the value of `eip` after this test and why? What do you see in the Taof debugging file this time?

The fuzzer overflowed `buf` with a long request. Since `buf` is located on the stack in the frame for `accept_request()` (`buf` is declared as a local variable in that function), the overflow causes the saved `ebp` and saved return address to be overwritten in that stack frame. `eip` has value `0x41414141` because the buffer is being filled with the character 'A' as padding, which is `0x41` in ASCII. Since `eip` is 32 bits, there are four of them. `eip` was set when the function `accept_request()` returned to main, since its return address had been overwritten and the `retn` instruction took that value from the stack and put it in `eip`. The program crashed because address `0x41414141` is not mapped into the processes' address space, therefore causing a memory fault. The debugging file will show the exact request that was sent to crash the server and will also report that it believes the server has crashed and it therefore cannot continue.

QUESTION 4: (5 pts) At what address does httpd-broken say `buf` is located?

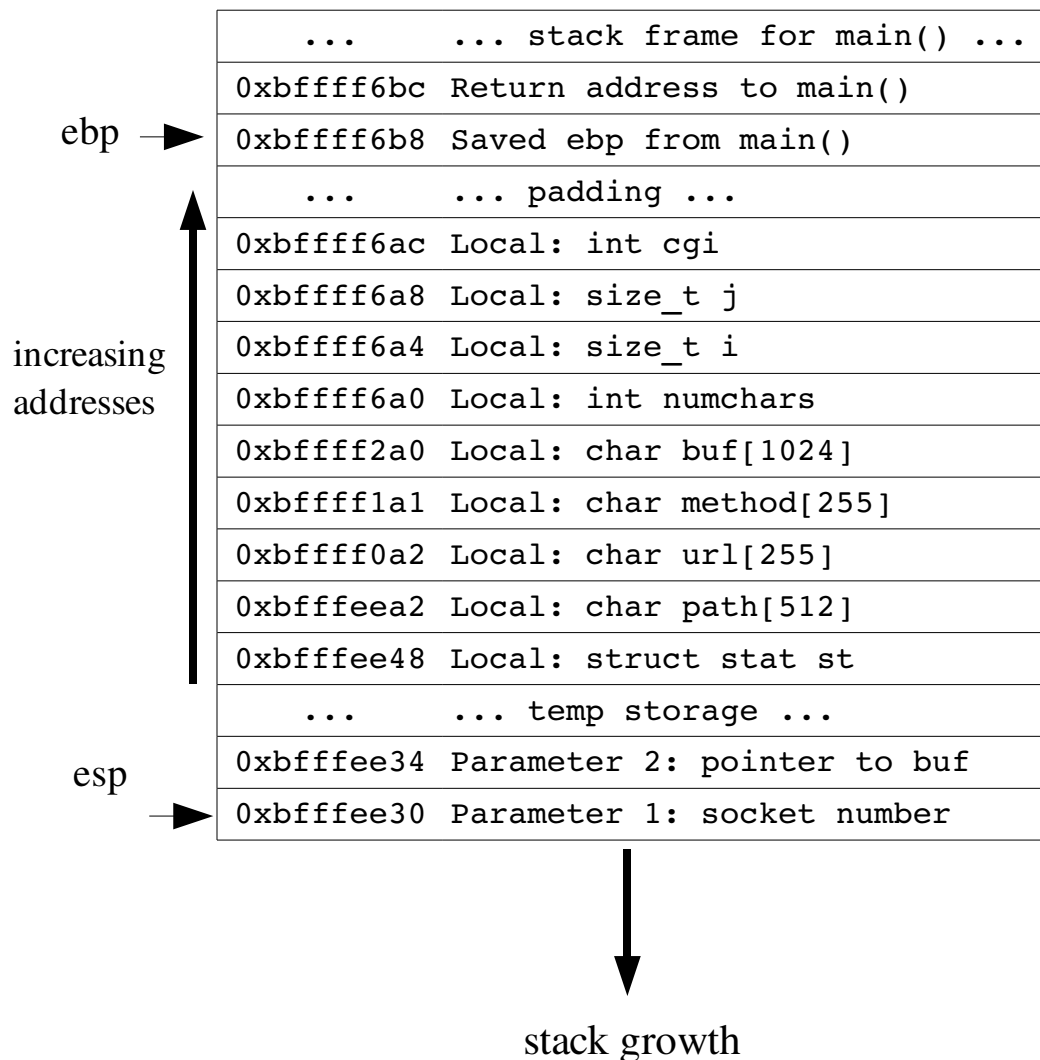
This address varied for different students, but most were in the range `0xbfffXXXX`, which is a stack address for this program. These were free points.

QUESTION 5: (5 pts) What happened when you ran the exploit? What output did httpd give?

Running the exploit caused the http server to be replaced with two programs, first a shell (`/bin/bash`) and then `/bin/echo`. The debugger gave some output showing that these program changes had occurred, but the real output was the string "Now I pwn your computer." You should also notice that the http server is no longer running (the program exited).

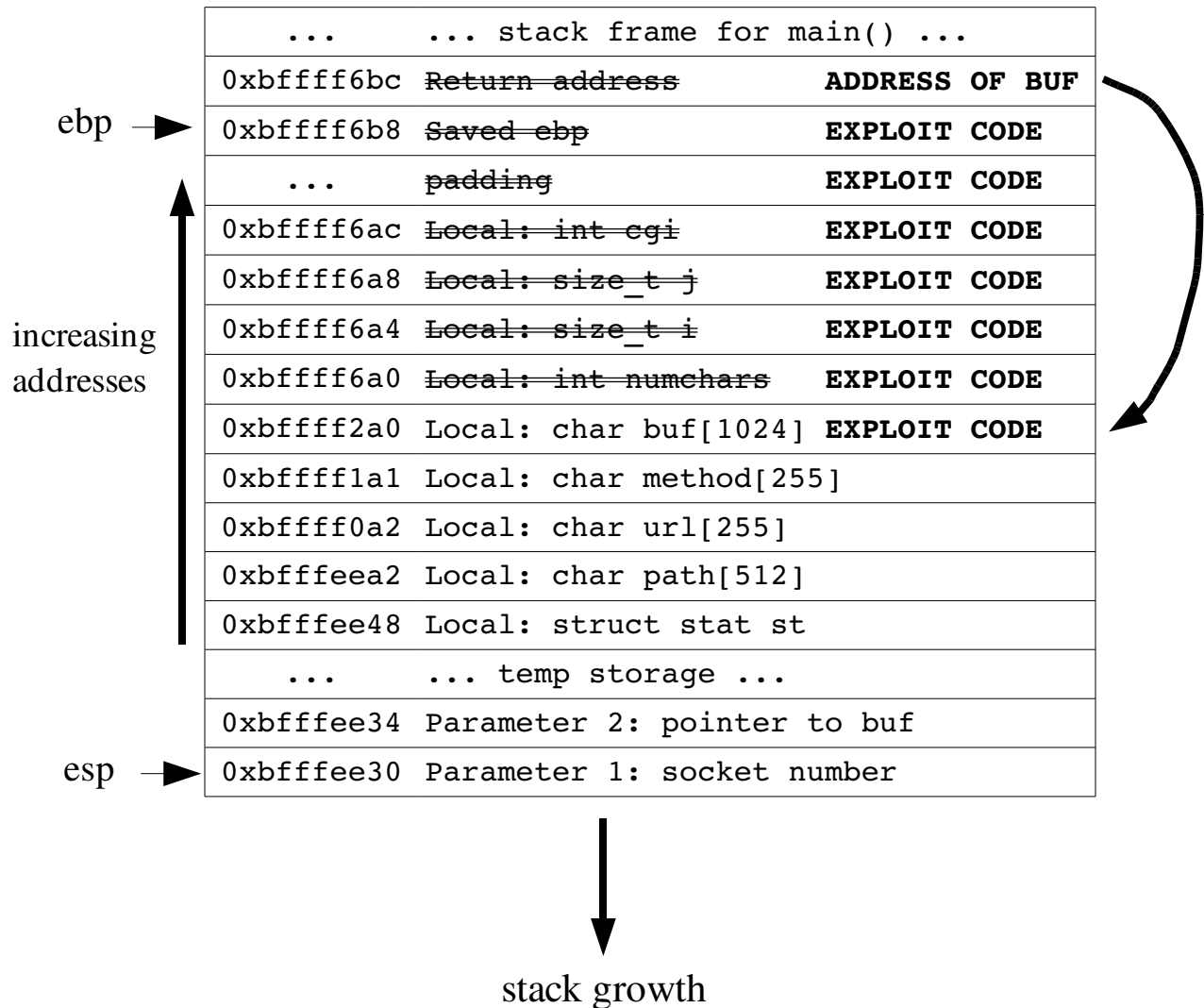
QUESTION 6: (20 pts) Draw a picture of the contents of the stack frame for the call to `accept_request` just BEFORE the attacker inserts his data (i.e., before the call to `get_line`). ASCII art is fine. Include, at a minimum, the locations of all local variables, the return address, the frame pointer, and the stack pointer. Your drawing should use abstract terms like “return address of `accept_request`” to describe the stack, not the actual addresses that are stored on it.

You did not have to give literal addresses for this answer. If you did, they would likely be different numbers than mine. The way I did this was to use the debugger and to set a breakpoint on the call to `get_line()` in `accept_request`. This was “`break *0x08048b02`” in my running process. I got that address by disassembling `accept_request` with the command “`disas accept_request`”. The key line of output was “`0x08048b02 <accept_request+62>: call 0x804959d <get_line>`” and you can see the address that I got. Then I ran the program (“`run`”) and launched the exploit, causing a break to occur. After the break, I used the command “`x/10 $esp`” to see the beginning of the stack. I also looked for the addresses of the local variables using “`print &varname`” where “`varname`” is the actual variable name. Finally, I looked at the values of the stack beyond the locals to see where the return address is located (found using the “`bt`” command or disassembling `main`). Here is what my stack looks like:



QUESTION 7: (20 pts) Draw a picture of the contents of the stack frame for `accept_request` just AFTER the attacker has inserted his data (i.e., after the call to `get_line`). The same guidelines apply for this stack drawing.

This time I simply set a breakpoint at the return address and allowed the program to continue until after the call to `get_line()`. Since the exploit has now been injected, the stack will look as follows:



QUESTION 8: (5 pts) What output did `httpd-protected` give? You don't have to give the exact output, just describe it.

You should see something like **“*** stack smashing detected ***: ./httpd-protected terminated”** and then a backtrace (set of function calls that led to the detection) and a description of the various memory sections of the program, including their locations, protections, and names. Finally, the program will abort and stop running.

QUESTION 9: (10 pts) Why did this happen? Name two protections that the compiler could have added and explain them.

This happened because the compiler added a check for a buffer overflow that was performed at runtime. The most likely mechanism used for this check was a stack canary. A stack canary is random value that the compiler inserts just before the saved frame pointer or the return address. Before executing a function return, the compiler checks to make sure that the canary is still there. A second protection that the compiler could have used is local variable reordering. The compiler may put array/buffer variables above non-array variables on the stack. A variant of this approach is to keep a shadow stack or make copies of critical variables.

Other answers that received credit were pointer encryption and address space layout randomization. However, note that ASLR is typically implemented by the OS, not the compiler.

QUESTION 10: (5 pts) What do you notice about the address of buf for each new run of httpd-broken? Why do you think this happened?

The value of buf is different with every execution of the program. This happens because address space layout randomization has been enabled in the kernel. Because the stack is placed at a random location and buf is located on the stack, the location of buf will be different in each process instance.

QUESTION 11: (10 pts) Did the exploit succeed? What is the last kernel log message when you run 'dmesg' from the command line? Why do you think that happened?

The exploit did not succeed. The last kernel log was “httpd-broken[3071] general protection ip:8048e6d sp:bffff6fc error:0 in httpd-broken[8048000+3000].” This message indicates that the kernel aborted the process because a memory access violation occurred. Because we enabled ExecShield, the stack portion of the program was not executable. When the exploit tried to execute its buffer on the stack, the hardware raised an exception and the kernel aborted the process. Note that this is different from the stack canary case where the program itself decided to abort because of the canary check. (You did not need to note that to get full credit).