Gustavo Duarte (/gustavo/blog/)

brain food for hackers

Epilogues, Canaries, and Buffer Overflows

Mar 19th, 2014

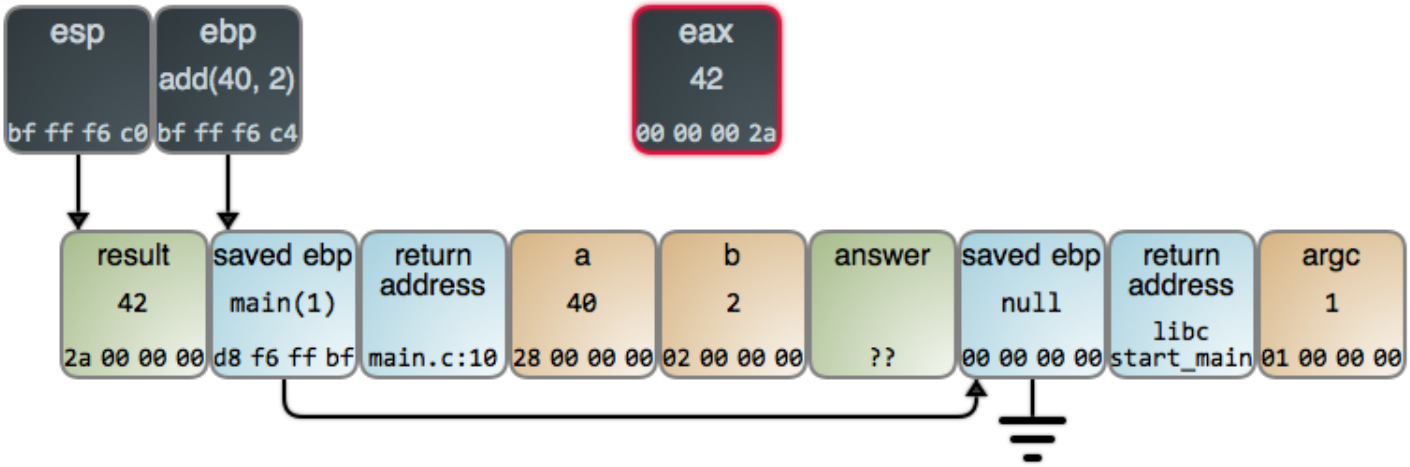
Last week we looked at how the stack works (/gustavo/blog/post/journey-to-the-stack) and how stack frames are built during function *prologues*. Now it's time to look at the inverse process as stack frames are destroyed in function *epilogues*. Let's bring back our friend `add.c`:

Simple Add Program - add.c

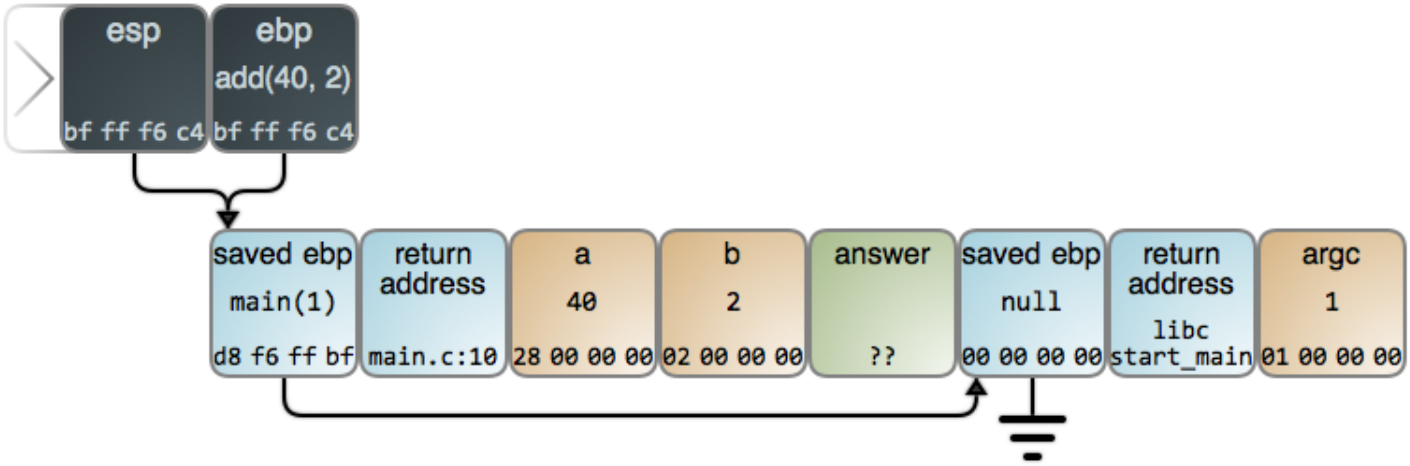
```
1  int add(int a, int b)
2  {
3    int result = a + b;
4    return result;
5  }
6
7  int main(int argc)
8  {
9    int answer;
10   answer = add(40, 2);
11 }
```

We're executing line 4, right after the assignment of `a + b` into `result`. This is what happens:

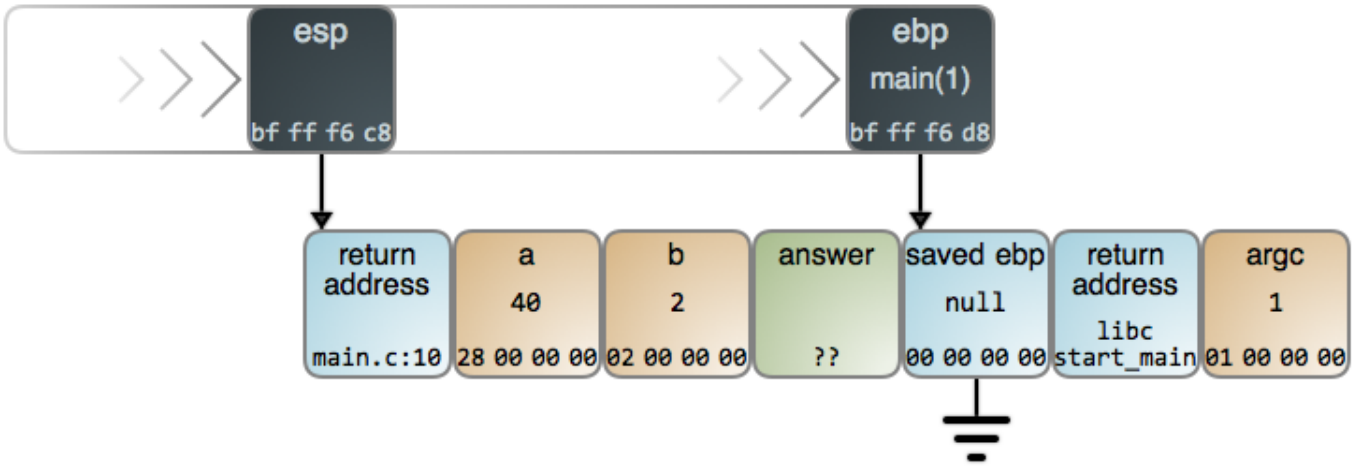
```
1.  movl -4(%ebp), %eax # send result as the return value through eax
```



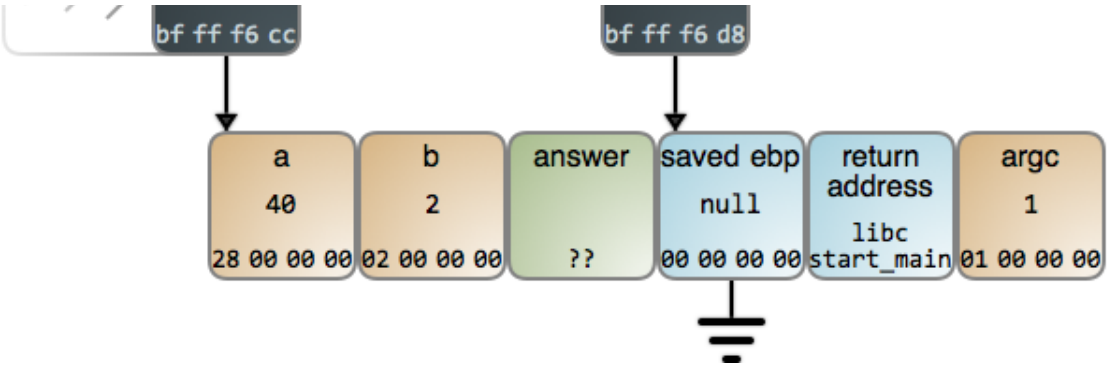

```
2.  leave # part 1: copy ebp to esp
```




```
3.  leave # part 2: pop into ebp
```



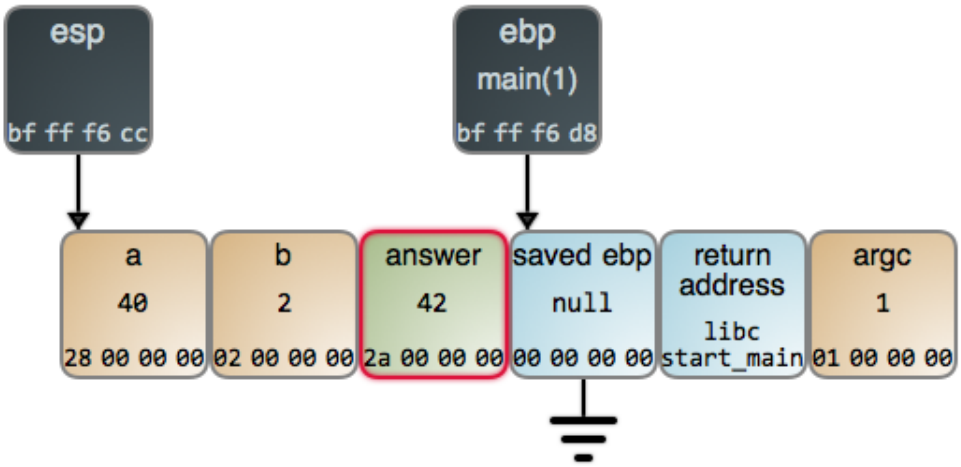

```
4.  ret # pop into eip (instruction pointer)
```



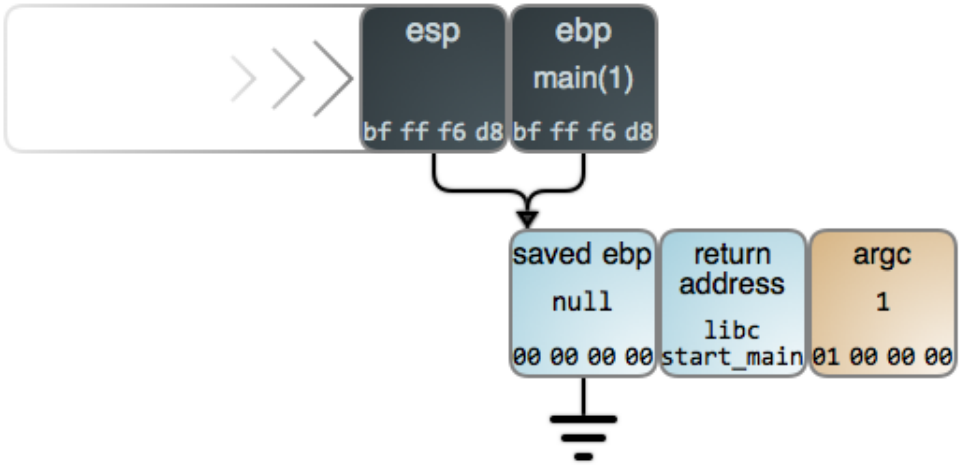
The first instruction is redundant and a little silly because we know `eax` is already equal to `result`, but this is what you get with optimization turned off. The `leave` instruction then runs, doing two tasks for the price of one: it resets `esp` to point to the start of the current stack frame, and then restores the saved ebp value. These two operations are logically distinct and thus are broken up in the diagram, but they happen atomically if you're tracing with a debugger.

After `leave` runs the previous stack frame is restored. The only vestige of the call to `add` is the return address on top of the stack. It contains the address of the instruction in `main` that must run after `add` is done. The `ret` instruction takes care of it: it pops the return address into the `eip` register, which points to the next instruction to be executed. The program has now returned to main, which resumes:

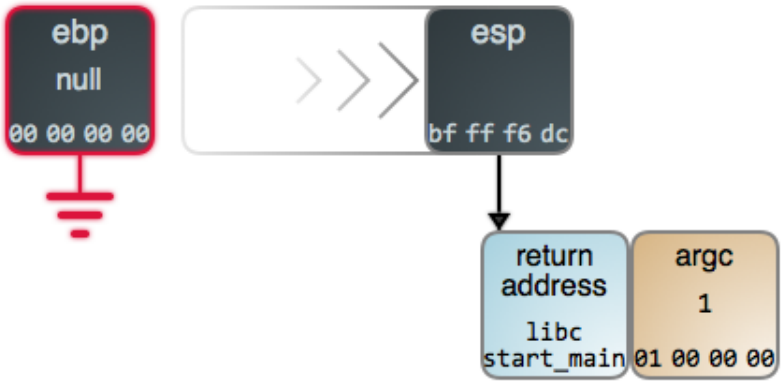
```
5.  movl %eax, -4(%ebp) # copy eax to answer
```



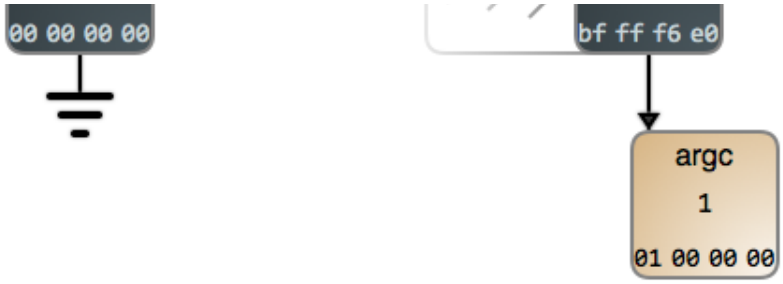

```
6.  leave # part 1: copy ebp to esp
```




```
7.  leave # part 2: pop into ebp
```




```
8.  ret # pop into eip (instruction pointer)
```



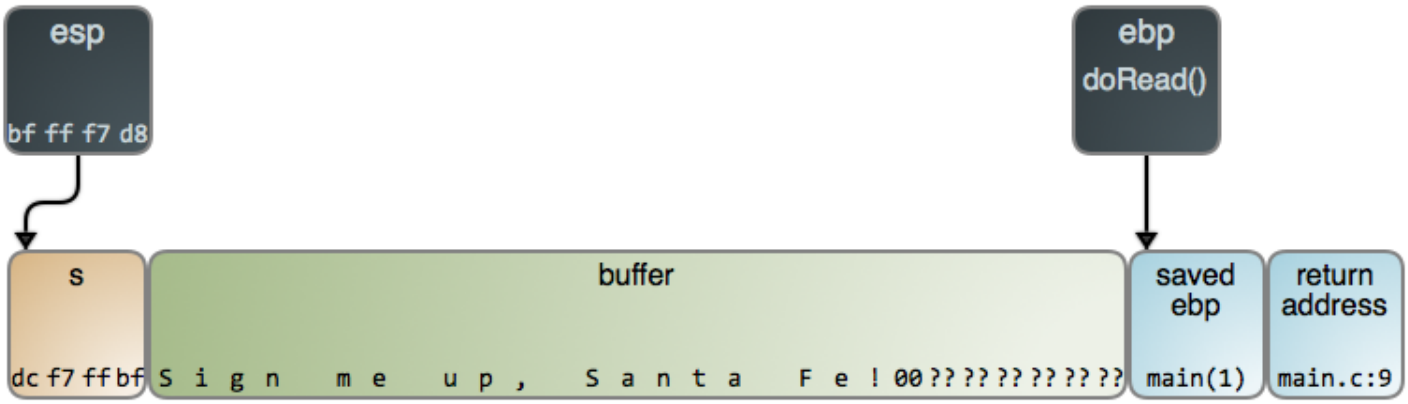
`main` copies the return value from `add` into local variable `answer` and then runs its own epilogue, which is identical to any other. Again the only peculiarity in `main` is that the saved ebp is null, since it is the first stack frame in our code. In the last step, execution has been returned to the C runtime (`libc`), which will exit to the operating system. Here's a diagram with the full return sequence (/gustavo/blog/img/stack/returnSequence.png) for those who need it.

You now have an excellent grasp of how the stack operates, so let's have some fun and look at one of the most infamous hacks of all time: exploiting the stack buffer overflow. Here is a vulnerable program:

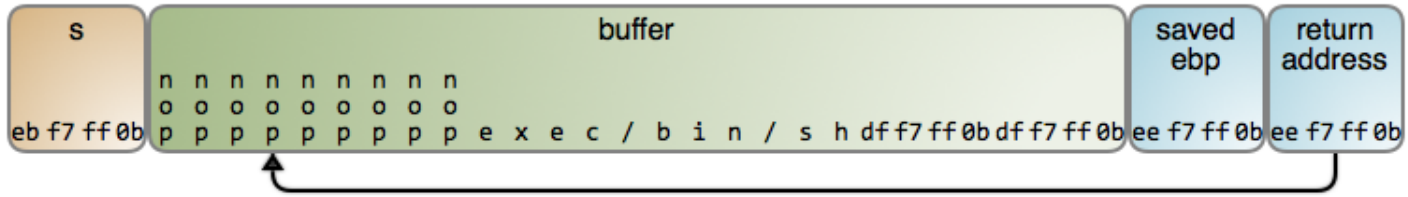
Vulnerable Program - buffer.c

```
1  void doRead()
2  {
3          char buffer[28];
4          gets(buffer);
5  }
6
7  int main(int argc)
8  {
9          doRead();
10 }
```

The code above uses gets (http://linux.die.net/man/3/gets) to read from standard input. `gets` keeps reading until it encounters a newline or end of file. Here's what the stack looks like after a string has been read:



The problem here is that `gets` is unaware of `buffer`'s size: it will blithely keep reading input and stuffing data into the stack beyond `buffer`, obliterating the saved ebp value, return address, and whatever else is below. To exploit this behavior, attackers craft a precise payload and feed it into the program. This is what the stack looks like during an attack, after the call to `gets`:



The basic idea is to provide malicious assembly code to be executed *and* overwrite the return address on the stack to point to that code. It is a bit like a virus invading a cell, subverting it, and introducing some RNA to further its goals.

And like a virus, the exploit's payload has many notable features. It starts with several `nop` instructions to increase the odds of successful exploitation. This is because the return address is absolute and must be guessed, since attackers don't know exactly where in the stack their code will be stored. But as long as they land on a `nop`, the exploit works: the processor will execute the nops until it hits the instructions that do work.

The `exec /bin/sh` symbolizes raw assembly instructions that execute a shell (imagine for example that the vulnerability is in a networked program, so the exploit might provide shell access to the system). The idea of feeding raw assembly to a program expecting a command or user input is shocking at first, but that's part of what makes

security research so fun and mind-expanding. To give you an idea of how weird things get, sometimes the vulnerable program calls `tolower` or `toupper` on its inputs, forcing attackers to write assembly instructions whose bytes do not fall into the range of upper- or lower-case ascii letters.

Finally, attackers repeat the guessed return address several times, again to tip the odds ever in their favor. By starting on a 4-byte boundary and providing multiple repeats, they are more likely to overwrite the original return address on the stack.

Thankfully, modern operating systems have a host of protections against buffer overflows (http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/), including non-executable stacks and *stack canaries*. The "canary" name comes from the canary in a coal mine (http://en.wiktionary.org/wiki/canary_in_a_coal_mine) expression, an addition to computer science's rich vocabulary. In the words of Steve McConnell:

> Computer science has some of the most colorful language of any field. In what other field can you walk into a sterile room, carefully controlled at 68°F, and find viruses, Trojan horses, worms, bugs, bombs, crashes, flames, twisted sex changers, and fatal errors?
>
> — **Steve McConnell** *Code Complete 2*

At any rate, here's what a stack canary looks like:

Canaries are implemented by the compiler. For example, GCC's stack-protector (http://gcc.gnu.org/onlinedocs/gcc-4.2.3/gcc/Optimize-Options.html) option causes canaries to be used in any function that is potentially vulnerable. The function prologue loads a magic value into the canary location, and the epilogue makes sure the value is intact. If it's not, a buffer overflow (or bug) likely happened and the program is aborted

via __stack_chk_fail (http://refspecs.linux-foundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/libc---stack-chk-fail-1.html). Due to their strategic location on the stack, canaries make the exploitation of stack buffer overflows much harder.

This finishes our journey within the depths of the stack. We don't want to delve too greedily and too deep. Next week we'll go up a notch in abstraction to take a good look at recursion, tail calls and other tidbits, probably using Google's V8. To end this epilogue and prologue talk, I'll close with a cherished quote inscribed on a monument in the American National Archives:

Posted by Gustavo Duarte Mar 19th, 2014 Internals (/gustavo/blog/category/internals/), Software Illustrated (/gustavo/blog/category/software-illustrated/)

« Journey to the Stack, Part I (/gustavo/blog/post/journey-to-the-stack/)
Recursion: dream within a dream » (/gustavo/blog/post/recursion/)

Load Comments

Recent Posts

(/gustavo/blog/about/)

(//twi(ttetp.co/m/feilto:deedufsatrvok@nde.rcuoamr/Gtuessta.voorDg)uarte)