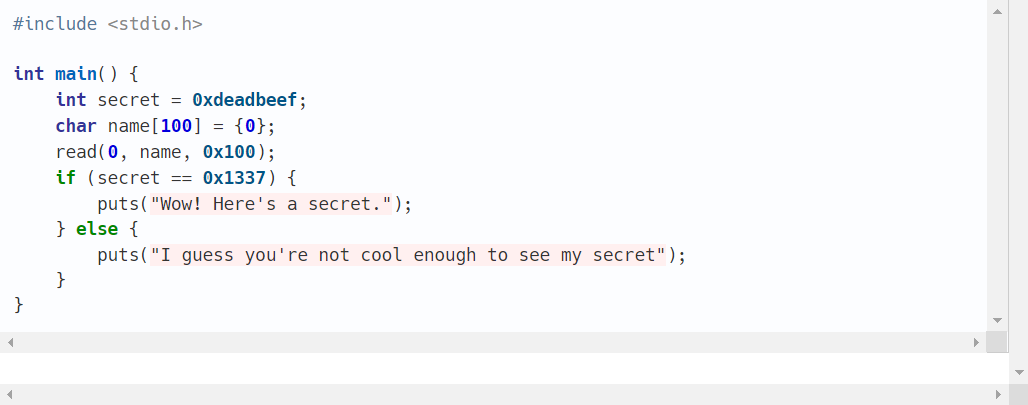
**Buffer Overflow**

A Buffer Overflow is a vulnerability in which data can be written which exceeds the allocated space, allowing an attack to overwrite other data.

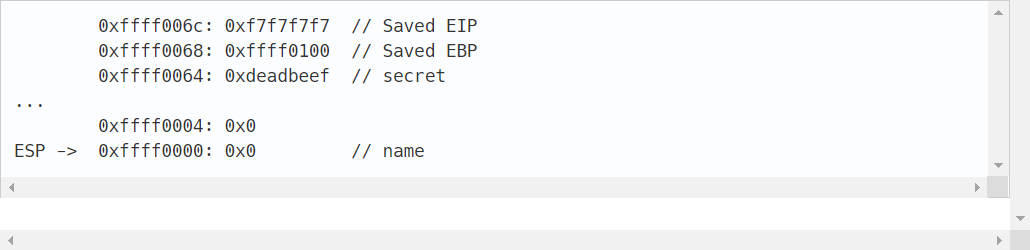
**Stack buffer overflow**

The simplest and most common buffer overflow is one where the buffer is on the stack.



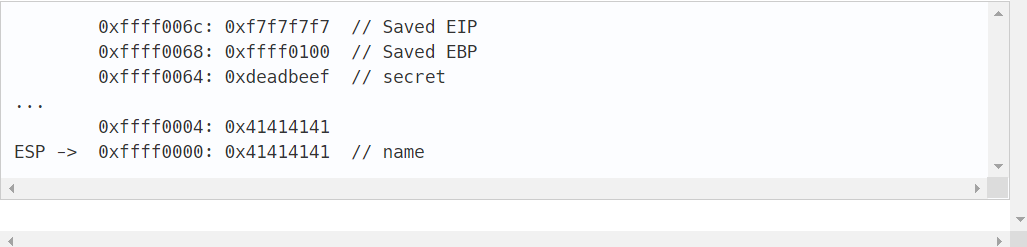
There’s a small mistake in this program which will allow us to see the secret. **name** is decimal 100 bytes, however we are reading in hex 100 bytes (=256 decimal bytes). Let’s check whether we can use this to our advantage.

If the compiler chose to layout the stack like this:

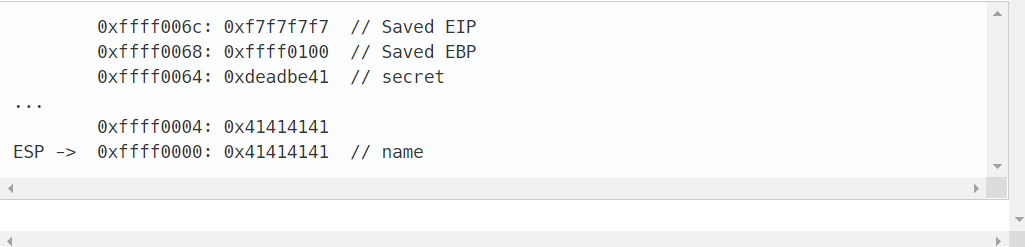


Let’s see at what happens when we read in 0x100bytes if ‘A’s.

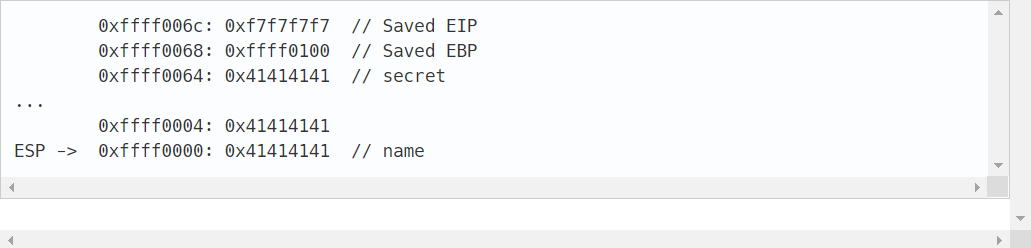
The first decimal 100 bytes are saved properly.



However, when the 101 byte is read in, we can see an issue.



The least significant byte of **secret** has been overwritten. If we follow the next 3 bytes to be read in, we will see the entirety of **secret** is “clobbered” with our “A’s



The remaining 152 bytes would continue clobbering values up the stack.

**Passing an impossible check**

How can we use this to pass the seemingly impossible check in the original program? If we carefully line up our input so that the bytes that overwrite **secret** happen to be the bytes that represent 0x1337 in little-endian we will see the secret message.

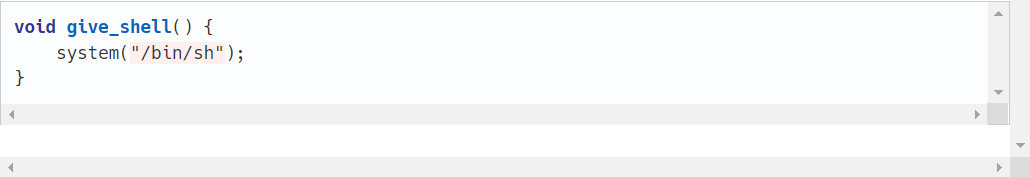
This will fill the **name** buffer with 100 ‘A’s, then overwrite **secret** with the 32-bit little-endian encoding of 0x1337.

**Moving one step further**

As discussed on [the stack](https://ctf101.org/binary-exploitation/what-is-the-stack) page, the instruction that the current function should jump to when it is done is also saved on the stack (denoted as "Saved EIP" in the above stack diagrams). If we can overwrite this, we can control where the program jumps after **main** finishes running, giving us the ability to control what the program does entirely.

Usually, the end objective in binary exploitation is to get a shell (often called "popping a shell") on the remote computer. The shell provides us with an easy way to run *anything* we want on the target computer.

Say there happens to be a nice function that does this defined somewhere else in the program that we normally can't get to:



Based on the Buffer overflow knowledge we have all we have to do is to overwrite the save EIP on the stack to the address where **give shell** is. Then, when main returns, it will pop that address off the stack and jump to it, running **give shell**, and giving us our shell.

Assuming **give shell** is at 0x08048fd0, we could use something like **python -c “print ‘A’\*108+ ‘\xd0\x8f\x04\x08’”**

We send 108 'A's to overwrite the 100 bytes that is allocated for **name**, the 4 bytes for **secret**, and the 4 bytes for the saved EBP. Then we simply send the little-endian form of **give shell**'s address, and we would get a shell.