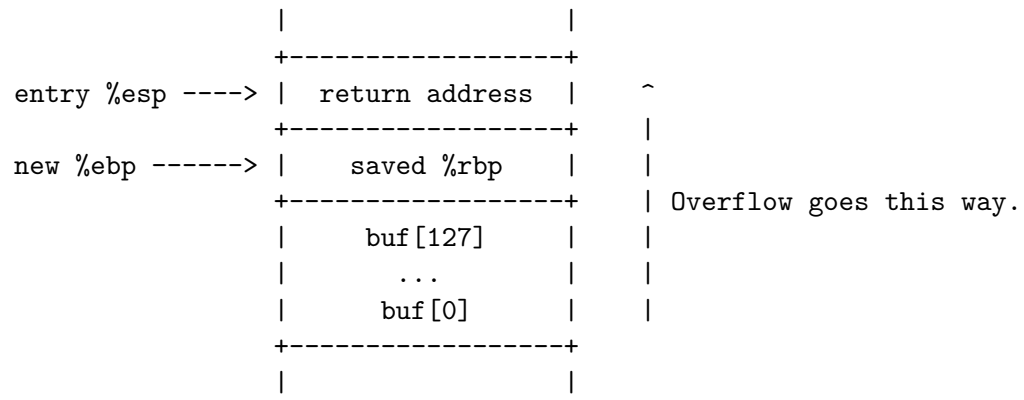
MIT 6.858 - Computer Systems Security "Lecture 4 - Buffer Overflows"

Lecture 4 - Buffer Overflows

Buffer overflows are an old but still relevant attack path. A buffer overflow is an attack which uses poorly sanitised/validated user input to overwrite data on the buffer. When a function is called, the registers and the return address are stored on the stack. Sometimes, a buffer is allocated to the function if the function needs to store some data. The buffer is allocated just under the pushed registers.

If the program accepts and stores user input in the buffer and does not check the length of the input, the user input can possibly overwrite the data in the stored registers if the programming language does not check for buffer bounds before writing. For example, whenever a function is called, the stack may contain the return address, the base pointer of the stack, and finally the buffer itself. If a buffer overflow attack exists, the attacker may write data that is so long that it fills the buffer and also overwrites the base pointer of the stack and the return address.



This way, the attacker can supply any return address, which means the attacker can run any function they want. In the classic buffer overflow attack, the attacker supplies the code they want to run in the buffer itself, and the specify the return address as the first address of the buffer.

Now, this attack could be completely mitigated if the programming language simply checked if the address being written to is part of the data structure being accessed. Most modern programming languages do that. However, C doesn't. If you are not using C, then you are probably safe from buffer overflows in your code. For example, if you are using a language like Java or Rust which performs bounds checking, then you most likely won't get buffer overflows. However, you can't always control the programming language you use, and you will, at some point, have to use libraries that are written in C. Even if you don't, you will probably want to run your code on Linux, which is written in C. Therefore, it is important to learn about buffer overflows even if you don't work with C directly.

Defenses Against Buffer Overflows

The lecture takes an iterative approach to defending against buffer overflows. In computer security, we rarely have solutions that perfectly solve a problem. It is more common that we have solutions that make it harder to pull of a particular attack successfully. As we develop such solutions, attackers modify their attacks to get around these solutions, forcing us to come up with better solutions, and so on.

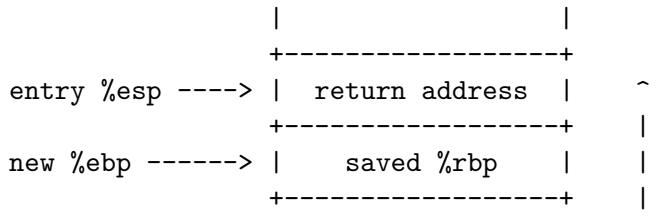
NX bit

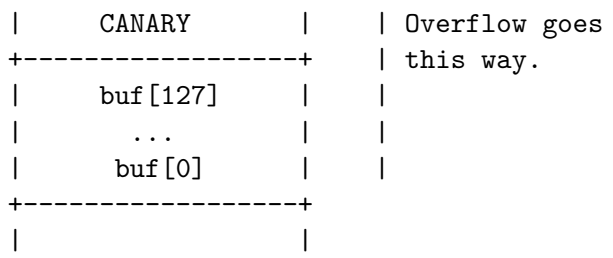
One approach we can take is to tell the hardware not to execute any instructions in pages where the stack is stored. This is done by the operating system, since it is responsible for paging and memory management. This is a simple solution, and it definitely works, but there is a way to bypass it.

Instead of the attacker providing the code they want to run in the buffer itself, the attacker can overwrite the return address to be a `libc` function, like `exec`, and achieve the same effect of running arbitrary code. This is called a "return-to-libc" type of buffer overflow.

Stack Canaries

The next approach to protect against "return-to-libc" attacks is called a stack canary. A stack canary is a number that is stored between the stack and the return addresses and saved registers. This is a randomly generated number and is different for every process. It is generated by the operating system and stored off the stack.





When the attacker overwrites addresses outside the buffer, the canary is also overwritten. Before returning, the compiler checks the canary to make sure it is correct. If the canary is not correct, then the compiler knows that a buffer overflow has occurred and crashes the program.

This is a pretty good solution, but it isn't completely foolproof since the canary could be guessed, either by reverse engineering how the canary is generated or just by brute force.

Address Space Layout Randomization

This is a common strategy and pretty effective in combating return-to-libc attacks. Instead of storing libc binaries in the same address space everytime, we randomize the addresses where we store system libraries on every startup, as well as the location of the application stack, code, and heap. This means that the attacker cannot guess where libc functions are stored (easily), and makes it much harder to pull off return-to-libc attacks.