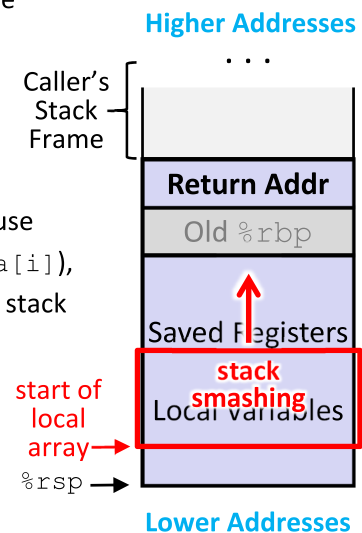
CSE 351 Lecture 15 – Buffer Overflow

Buffer Overflow

A ***buffer*** is a region of memory (usually an array) used to temporarily store data. In the most general sense, a ***buffer overflow*** is writing data past the end of a buffer and overwriting adjacent memory locations, which we know is achievable in C because there is no automatic bounds checking. Sometimes this is innocuous and goes unnoticed, sometimes this causes program execution to break, and sometimes this achieves more interesting (and malicious) outcomes.

> **Note:** This topic falls under the category of security and we would be remiss if we didn't at least mention the ethical ramifications of such techniques. Exploitation of security vulnerabilities can have massive real-world destructive and financial consequences – using such knowledge for personal interests is an abuse of your ethical responsibilities as a computer scientist.

To understand what can go wrong, we will rely on our understanding of the following previous material: memory addressing, layout of arrays, the memory layout, the behavior and internals of the stack and stack frames, and procedure calling conventions in assembly.



The form of buffer overflow we will focus on is known as ***stack smashing***, where we write past the end of a local array in the stack. Because arrays elements are laid out in increasing address order (*i.e.* `&a[i+1] > &a[i]`), buffer overflow naturally moves towards higher address. But because the stack grows *downward*, stack smashing will eventually lead to overwriting the return address in the current stack frame and data in previous stack frames! Because the `ret` instruction just pops off the word of data currently found at `%rsp` into `%rip`, modifying the stored return address can cause our program to jump to unexpected places – sometimes causing a seg fault, sometimes not!

Vulnerable Code

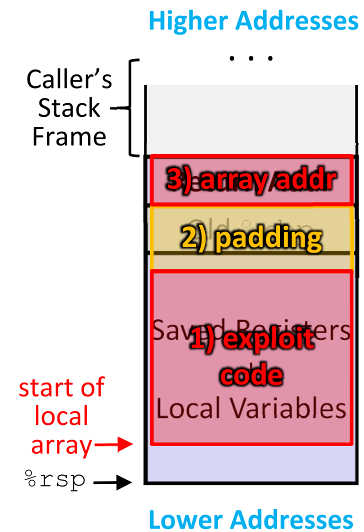
What would allow us to write arbitrary bytes past the end of an array? Lots of older library code was written without security in mind. This is especially problematic when getting user input – we often don't know ahead of time how many characters the user wants to input. A prime example of this is the function `gets()`, which reads a user-input string from the terminal and writes it into a programmer-defined array `dest` of fixed size: `char* gets(char* dest);`

`gets` will continue to write characters into the buffer until it encounters a newline or end-of-file character. If the user enters a longer string than we can hold in our buffer, then we get buffer overflow!

Similar issues can be found in library functions `strcpy` and `scanf` variants with the `%s` format specifier.

Code Injection Attacks

We can take advantage of the fact that everything in a computer is stored as data in memory (including instructions!) to perform a ***code injection***, where we use buffer overflow to write instructions into the buffer and then modify the return address to execute the injected code.



The typical procedure is shown in the diagram on the right. The attacker designs the input string in 3 parts:

1) Designed exploit code is compiled to its binary/machine code representation and put at the beginning of the buffer.
2) Based on the space available, any necessary padding is added to reach the current frame's stored return address.
3) The address of the beginning of the buffer is used to replace the return address.

When done successfully, the `ret` instruction will pop the address of the buffer into `%rip` instead of the return address to the caller and the program will begin executing the exploit code! Recall that even though the exploit code is no longer part of the stack, that data remains there until it is overwritten by future stack frames.

Notice that we don't want to mess with the data in the caller's stack frame in case we wish to resume normal execution later on. This attack relies on knowledge of the address of the buffer on the stack and the amount of space allocated between the array and the return address.

Preventing Buffer Overflow

Most modern systems employ two basic system-level protections against code injection attacks. The first is the *executable permission bit* for different memory segments. The stack should be marked non-executable, meaning that any attempt to execute code from a local buffer will fail. The second is *randomized stack offsets*, which shifts the stack addresses for an entire program and makes it harder for an attacker to know the exact address of the buffer.

We can also try to avoid overflow vulnerabilities in code by using library functions that limit input string lengths (*e.g.* `fgets()`) or by using a different language that does array bounds checking (*e.g.* Java).

A more advanced technique is known as using *stack canaries*, where the compiler places a random value just past the end of the buffer when it is allocated. If this value has changed when the array is about to be deallocated, then stack smashing is detected.