Buffer Overflow

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Buffer Overflow

- You may have heard of the “buffer overflow” method for exploiting a vulnerability of a program
  - e.g., to cause a Web server to do something potentially harmful, such as running code it wasn’t supposed to run
- The way in which this technique works is based on damaging the runtime stack
- Now that we know what the stack looks like, let’s see if we can understand how buffer overflow works
- We use the standard, simplest, example

The Basic Idea

- The goal is to have a program run code it wouldn't run in a normal/valid/allowed execution
- This is done by overwriting a return address on the stack
- When RET is executed, it pops off a 4-byte value from the stack, interprets it as an address in the text segment, and jumps to it
- If, somehow, these 4 bytes were modified illegally, then the program jumps to any address and starts running code
  - Some function in the program
  - Some library function or system call
  - Some arbitrary code (if one is a bit clever)
- This can be easily done if
  - The original program does unsafe memory operations
  - The attacker has knowledge of the program and of the architecture
    - e.g., installed it on a similar architecture and looked as the assembly
  - The attacker is reasonably clever

Corrupting the Stack

- Consider the following C program sketch, which takes one command-line argument:

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

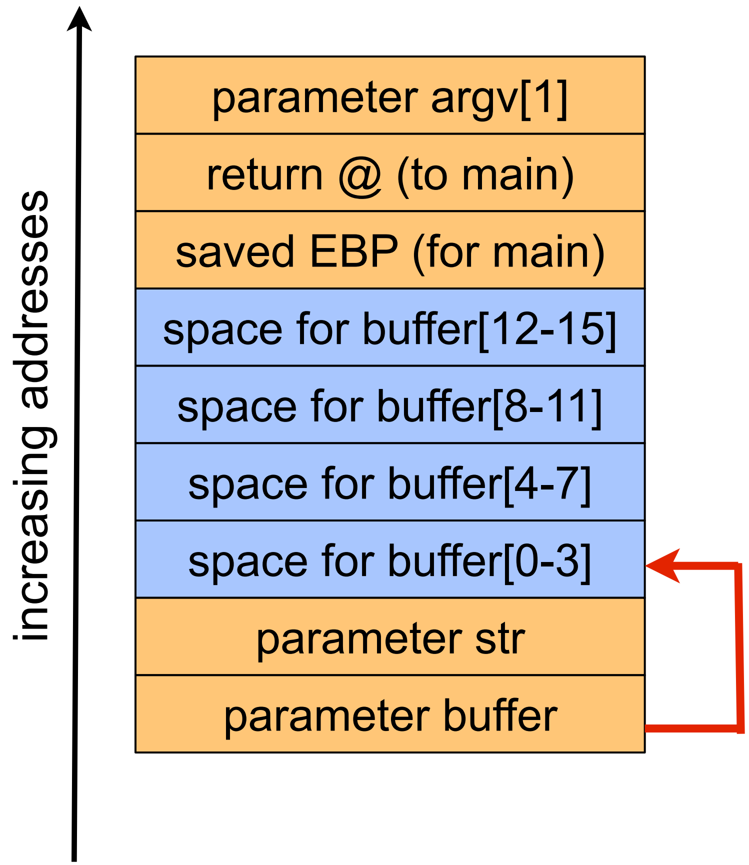
strcpy simply goes through the bytes in str and copies each byte into buffer, until it hits a \0 character

The Stack

- The Stack before the call to strcpy()

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

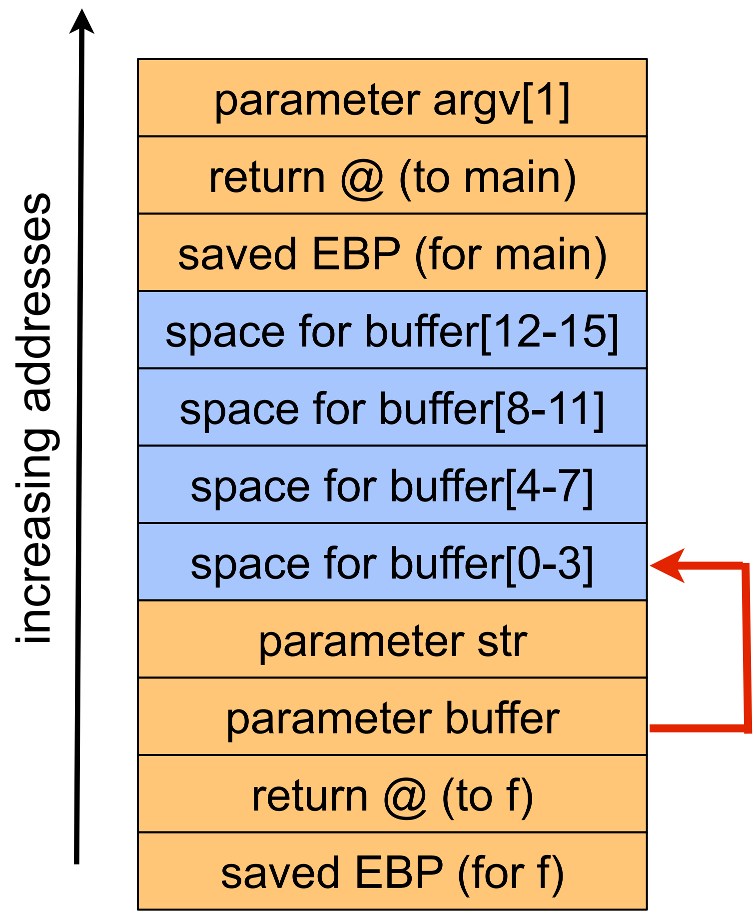


The Stack

- The Stack in the call to strcpy()

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

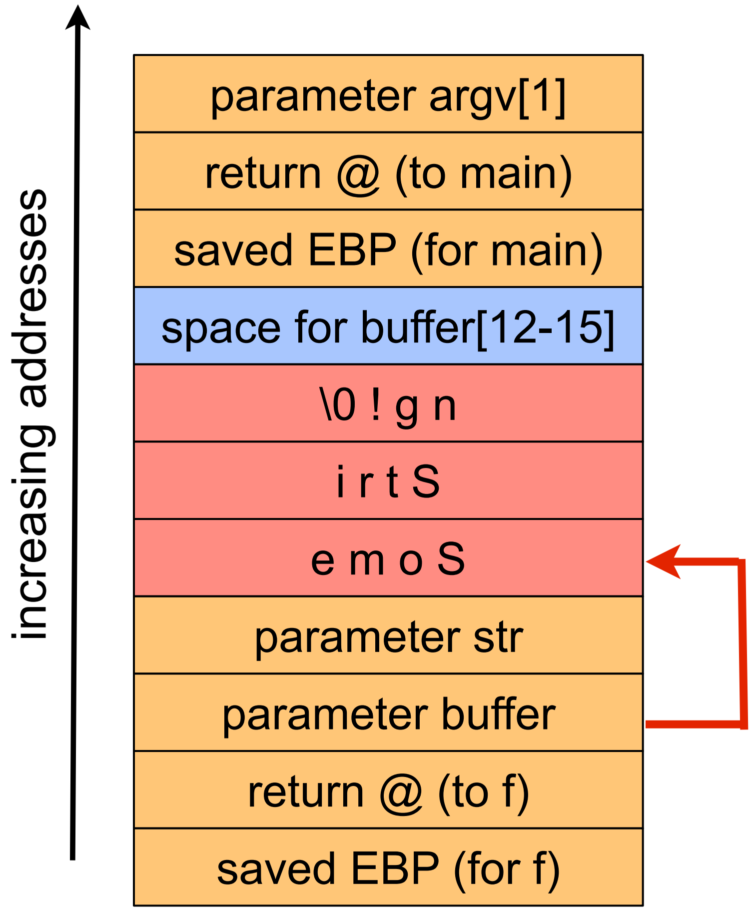


Writing into the buffer

- Say that argv[1]=”SomeString!\0”
- strcpy() writes it on the stack, in buffer[]

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```



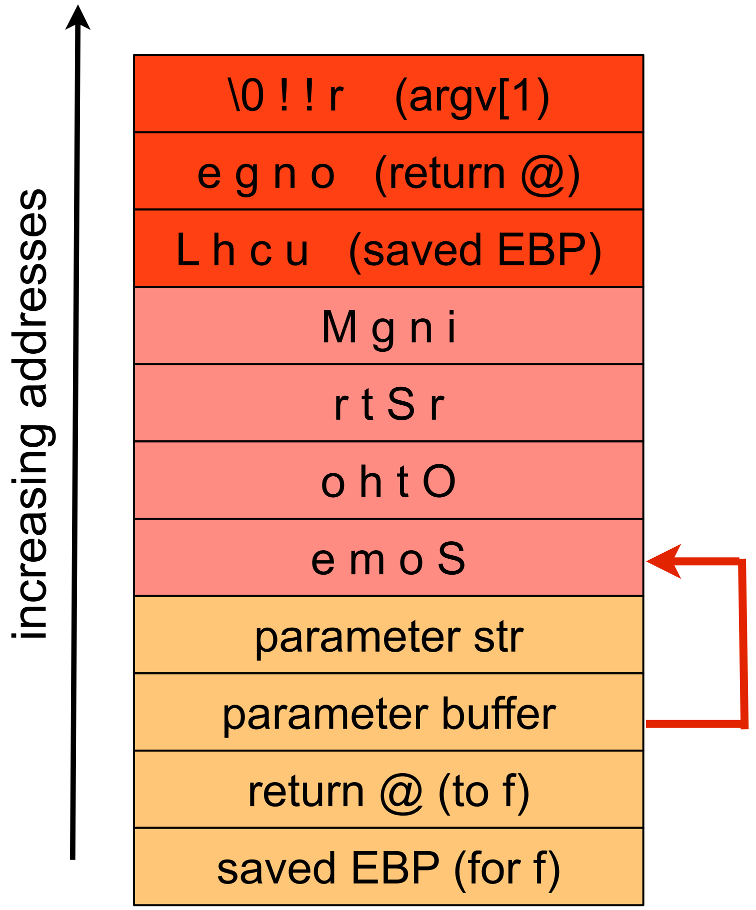
Bad code

- The problem is that the code is buggy
- C being C, you can write past the end of the array
- Say the code is part of a Web server, which is compiled and running on some host
- Say that the string passed to exploitable() comes from some Web request via the network
- If a string that is too long is passed, then the stack will be corrupted..

Writing into the buffer

- Say that argv[1]=”SomeOtherStringMuchLonger!!\0”
- strcpy() writes it on the stack, in buffer[]

- When strcpy returns, it restores ebp for exploitable and returns to exploitable
- exploitable then pops the two parameters for strcpy
- When exploitable returns it
  - removes space for buffer
  - restores the saved EBP to “uchL” (which is bogus)
  - **jumps to address “onge”!**



So What?

- If an attacker knows the address of some subprogram, he/she can create a string so that bytes 20-23 (“onge” in our example) form the bytes of this address!
- This requires that the attacker know the address of some subprogram to call
  - Can be discovered by “looking” at the program in debug mode (see later in the semester)
  - Only doable for known/standard programs
    - e.g., knowing that a Web server runs Apache, knowing which version it is, knowing the address of some function in that version, then one can perhaps exploit a buffer overflow
- More involved exploit: the overflowing string contains code, and the “fake” return address points to this code
  - One can then run arbitrary code that “looks like a string”

What can we do about it?

- A simple idea: make sure the subprogram doesn't overwrite activation records willy nilly
  - The activation record should be the subprogram's "play pen"
- But this would be tricky and costly
  - Some writes to the stack outside the activation record should be allowed (i.e., g passes to f a pointer to one of its local variables)
  - Would have to do an "is this ok?" check for every memory store operation
- Another idea, is to use a **stack canary**

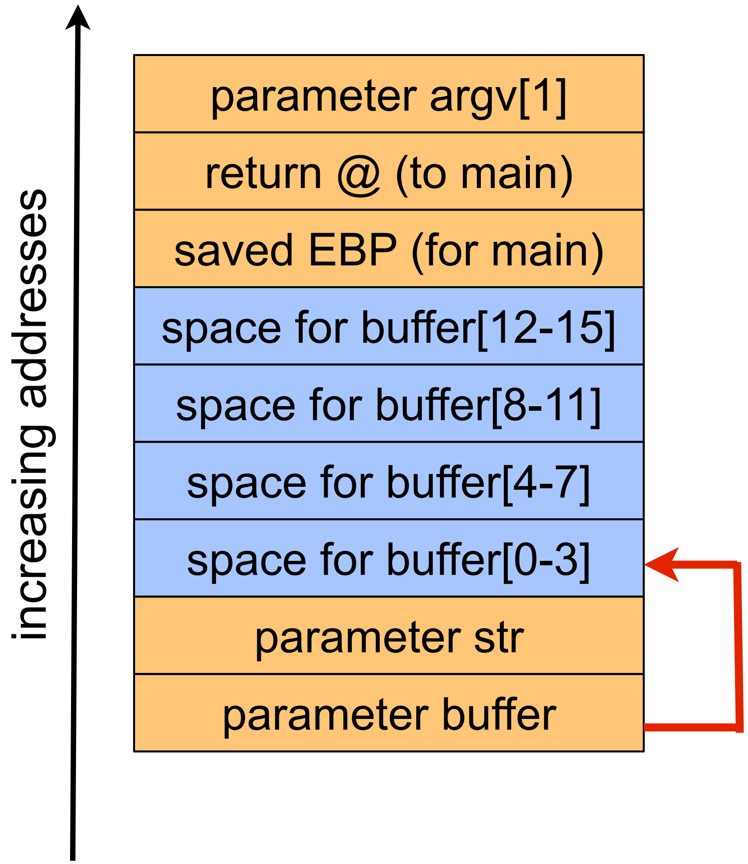
  - Have the compiler insert hidden local variables with secret values known to the compiler
  - Before doing the ret instruction, check that the canary hasn't changed!

Stack Canary

- Stack without canary

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

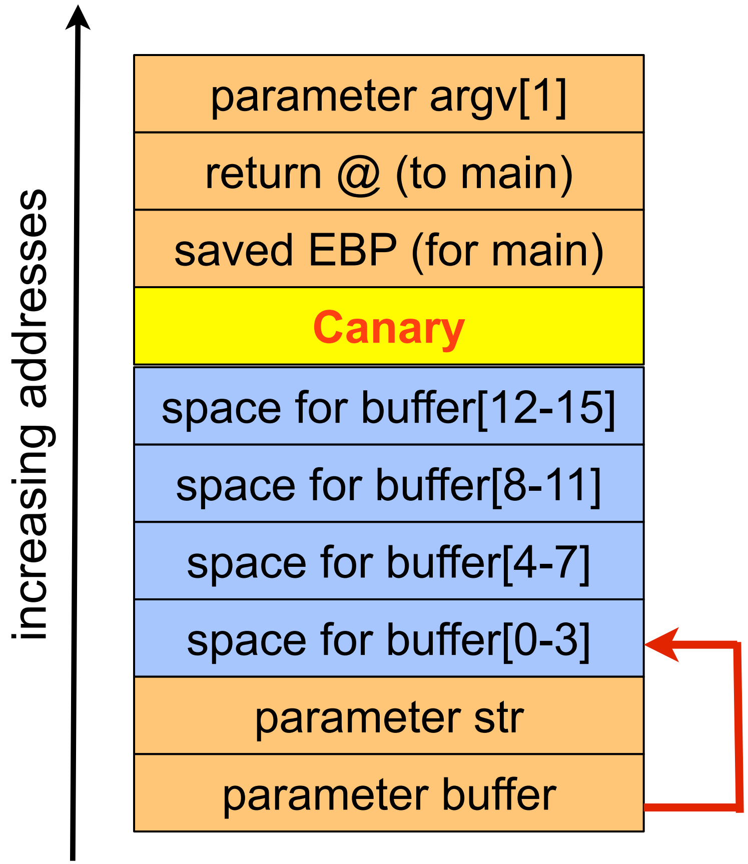


Stack Canary

- Stack with canary

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

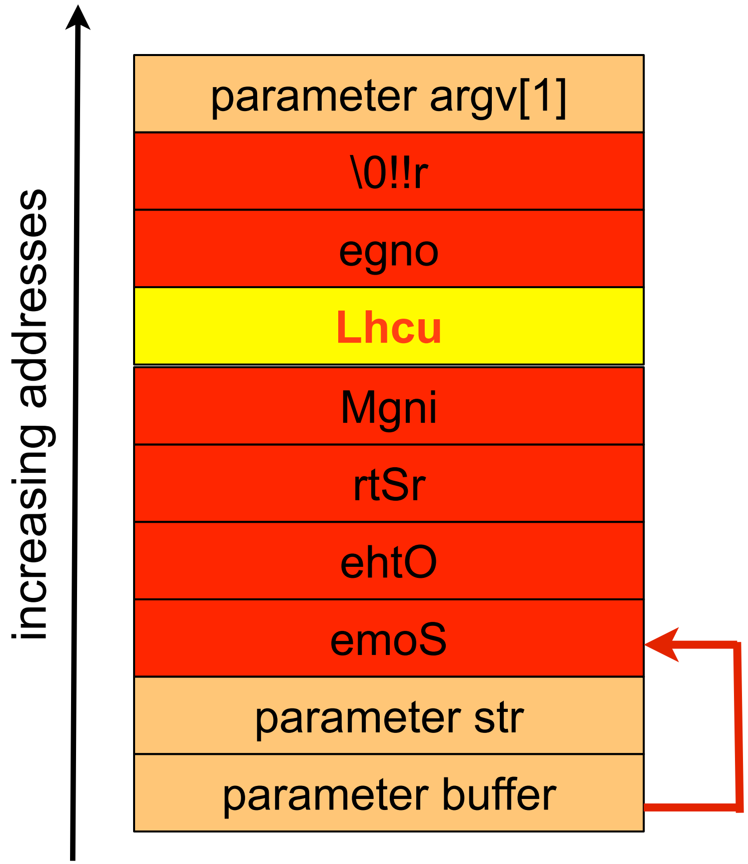


Stack Canary

- Buffer overflow modifies the canary!

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

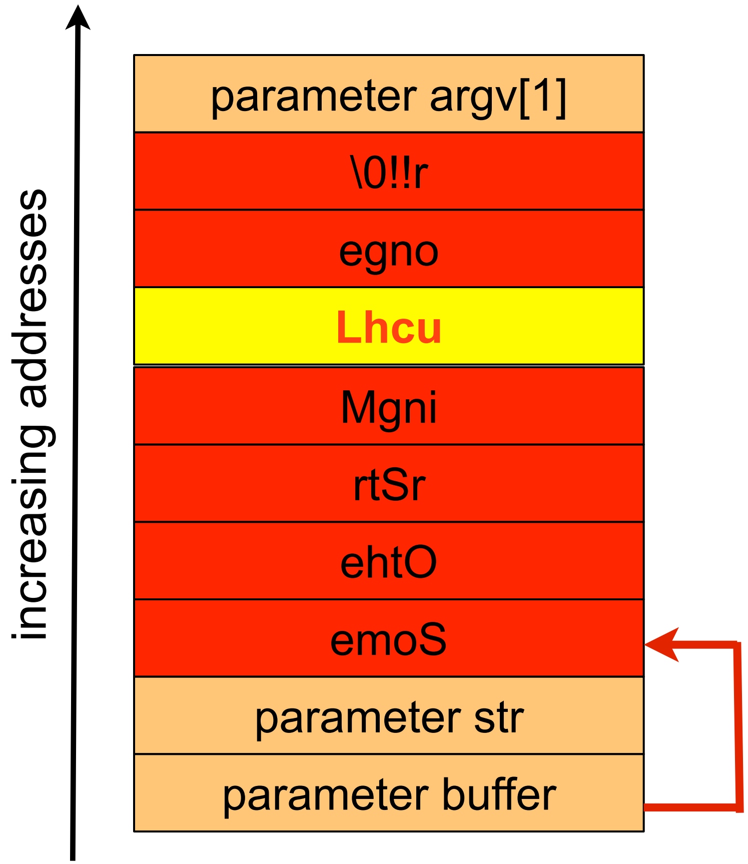


Stack Canary

- The ret instruction BEFORE doing a pop checks the canary

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```

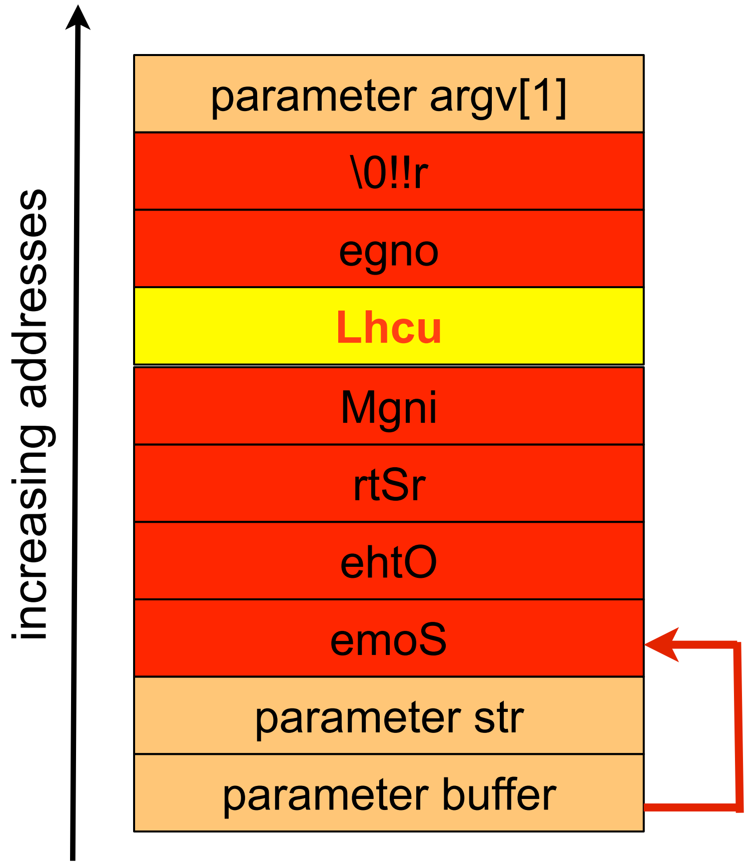


Stack Canary

- The canary has changed, and we branch to code that terminates the program

```
void exploitable(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main(int argc, char **argv) {
    exploitable(argv[1]);
}
```



What to do in practice

- Most compilers allow you to generate code that does runtime checks
- Check your compiler's documentation

- In gcc, flag -fstack-protector-all will make a canary for all functions
  - Safe, but a bit slow..

Conclusion

- Understanding what the stack looks like is necessary to understand how the system can be attacked
- This was the simplest example, and there is more to this
  - See any security course