CYBR 437: Secure Coding

Intro to Buffer Overflow

Instructor: Abinash Borah

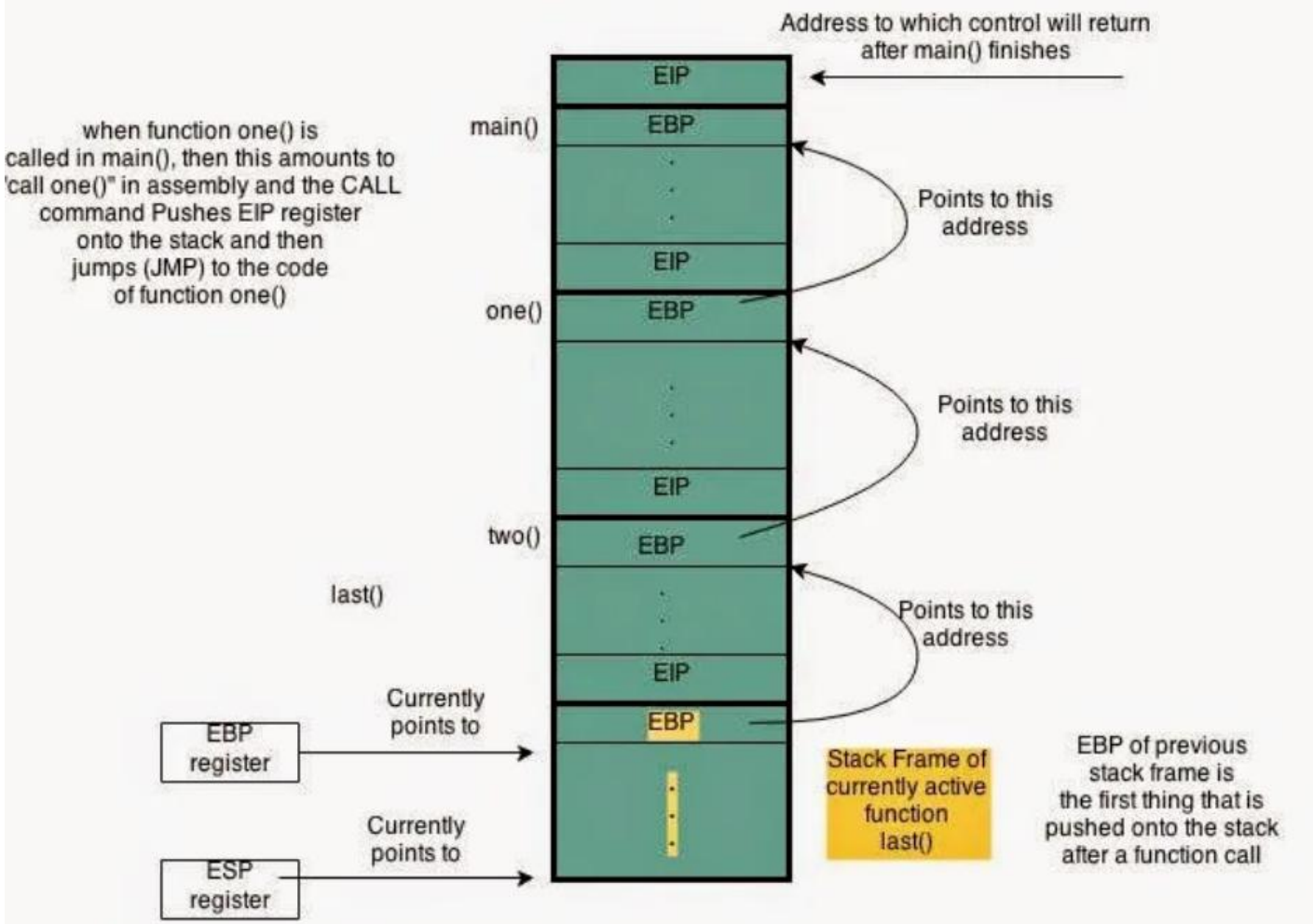
Outline

- Stack Registers
- Buffer Overflow

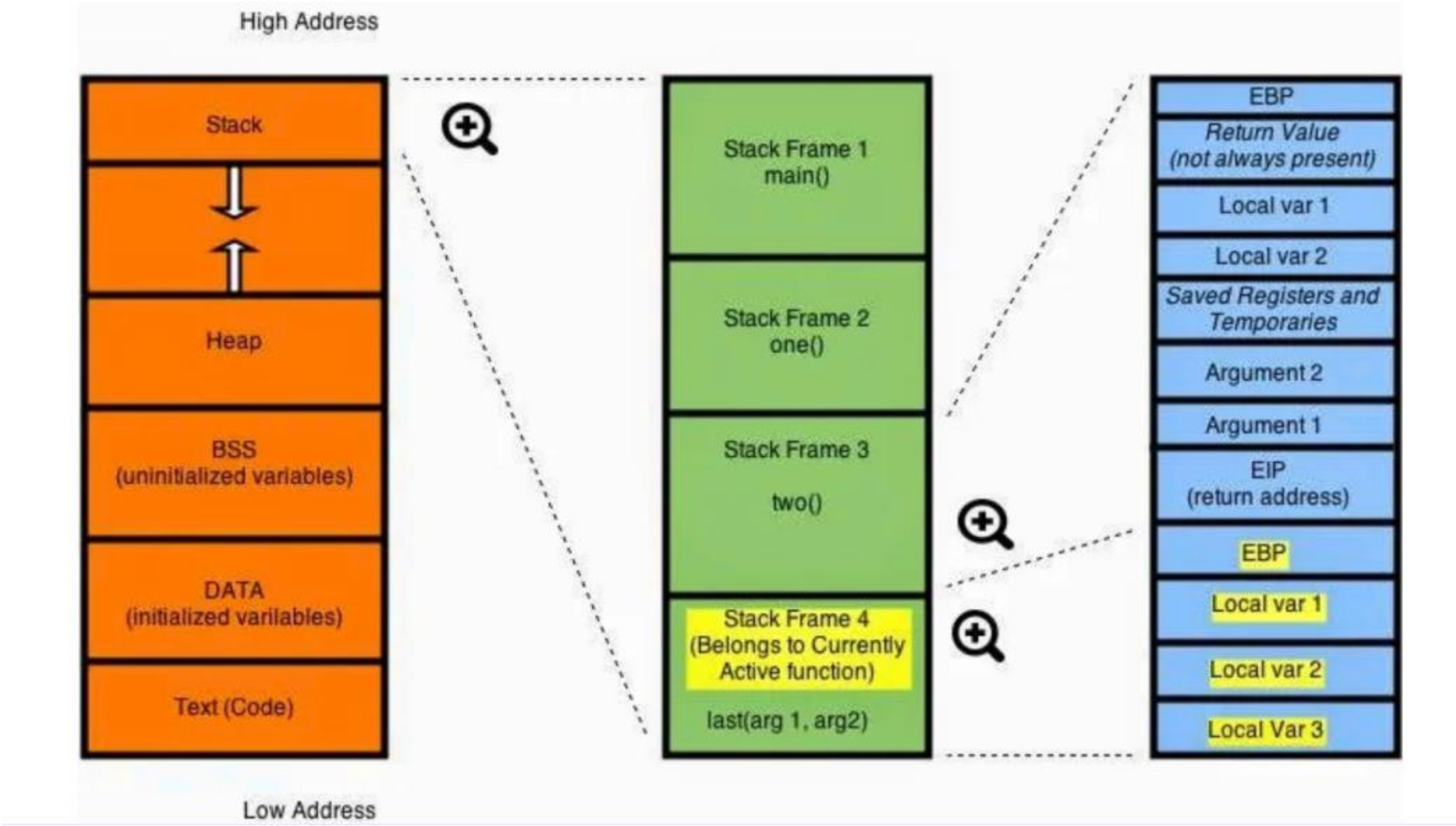
Stack Registers

- EBP (Extended Base Pointer):
  - The base pointer stores the address of the base of the stack frame
  - Serves as a fixed reference point for accessing local variables and function parameters within a function's stack frame
- ESP (Extended Stack Pointer):
  - The stack pointer always points to the lowest address on the stack
  - Essential for managing function calls, local variables, and preserving register values during program execution
- EIP (Extended Instruction Pointer):
  - The instruction pointer always points to the next instruction to be executed
  - EIP is the last thing to be saved on the stack frame of a given function - saved before transfer to another function
  - EIP is loaded with the old value (the calling function's next line of code)
  - ESP and EBP are set to the old values of the calling function

Stack Register Example



Stack Register Example (contd.)

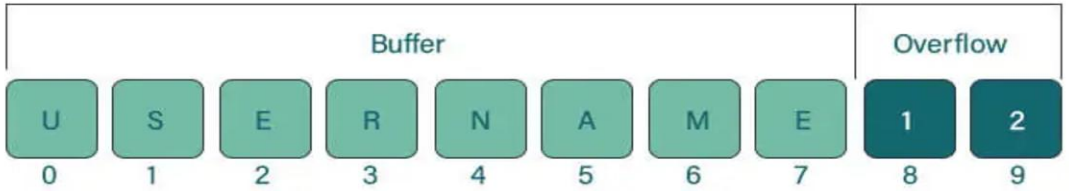


Stack Canary

- A hidden value called the stack canary is placed on the stack
- The value changes every time the program is run
- Before a function returns from the function call, the stack canary is checked
- If the stack canary appears to be modified, the program immediately terminates
- Stack canary affects the layout of the variables within the body of the function
- If the stack canary is enabled, the compiler will rearrange the variables, grouping them by type
- With the stack canary disabled, the variables will be in the order they were declared

What is a Buffer Overflow?

- A buffer overflow/overrun occurs when a program writes data to a buffer beyond the buffer's allocated memory, overwriting adjacent memory locations
- Most buffer overflows are caused by a combination of manipulating memory and mistaken assumptions about the composition or size of the data
- Attackers can exploit buffer overflow to crash a system or to insert specially crafted code to gain control of a system
- Can occur due to a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer



Consequences of Buffer Overflow

- Adjacent memory locations could hold other variables or parameters or program control flow data
- The buffer could be located on the stack, in the heap, or in the data section of the process
- Consequences:
  - Corruption of data used by the program
  - Possible memory access violations
  - Unexpected transfer of control in the program
  - Execution of attacker-chosen code
- Transfer of control could be to an attacker's code with the privileges of the attacked process

Example[1]

- Three variables: `valid, str1, and str2`
- Assume that they are stored in adjacent memory locations, from highest to lowest

```
int main(int argc, char *argv[]) {
  int valid = FALSE;
  char str1[8];
  char str2[8];
  next_tag(str1);
  gets(str2);
  if (strncmp(str1, str2, 8) == 0)
     valid = TRUE;
  printf("buffer1:str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

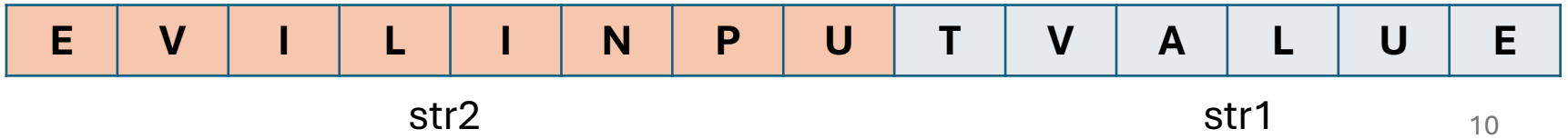
```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
```

Example

- Three variables: `valid, str1, and str2`
- Assume that they are stored in adjacent memory locations, from highest to lowest

```
int main(int argc, char *argv[]) {
  int valid = FALSE;
  char str1[8];
  char str2[8];
  next_tag(str1);
  gets(str2);
  if (strncmp(str1, str2, 8) == 0)
     valid = TRUE;
  printf("buffer1:str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
```



Example

- Three variables: `valid, str1, and str2`
- Assume that they are stored in adjacent memory locations, from highest to lowest

```
int main(int argc, char *argv[]) {
  int valid = FALSE;
  char str1[8];
  char str2[8];
  next_tag(str1);
  gets(str2);
  if (strncmp(str1, str2, 8) == 0)
     valid = TRUE;
  printf("buffer1:str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```