FAUST Binary Exploitation Workshop

Stackbased Buffer Overflows

April 29, 2023

Daniel Tenbrinck

Disclaimer

Ethical Hacking

In the following slides you will gain knowledge of methods that can be used to exploit computer systems. The aim of this lecture is to let you understand underlying concepts of computers and common vulnerabilities in program code from a scientific perspective in terms of **ethical hacking**. This should be used for personal research only and as preparation for CTF challenges. We strongly urge you to only practice ethical hacking on computer systems you own or have explicit permission to use for ethical hacking.
If you do find any vulnerabilities in software, always report it through the proper channels!

Outline

Introduction to ELF binaries

The call stack

Stackbased buffer overflows

Useful toolz

Introduction to ELF binaries

What is a binary (file)?

To exploit a binary we first need to understand what it is.

In our context, a **binary**

- is a file
- is executable
- encodes computer instructions
- is not readable as text
- has different formats, e.g.,

  → **COM** files, DOS binary; very simple

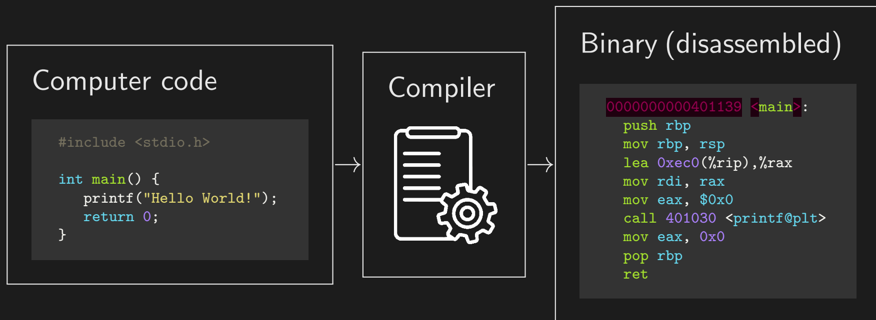
  → Portable Executable (**PE**), typical Windows *.exe binary

  → Executable and Linkable Format (**ELF**), Unix-like binary

  → many more can be found here

In this workshop a binary is a **compiled computer program** that contains:

- **machine code** (interpretable by the CPU)
- **data**, e.g., initialized variables, strings
- **meta information**, e.g., headers



What is a binary (file)?

In the following we restrict ourselves to **ELF binaries** for the **x86-64 instruction set**.

Use `file` command to see properties of binary:

```
$ file /usr/bin/cat

/usr/bin/cat: ELF 64-bit LSB pie executable, x86-64, version 1
(SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=512fa3723604a2176052d5a57317a29a58adb13a, for
GNU/Linux 4.4.0, stripped
```

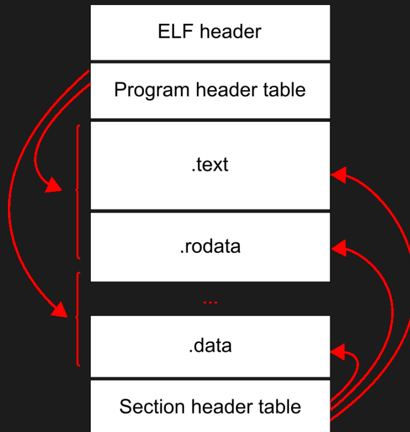
What is a binary (file)?

The operating system identifies ELF binaries by the first four bytes of the file, called **magic number**:

```
$ hexdump -C /usr/bin/cat | head

00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  e0 32 00 00 00 00 00 00  |..>......2......|
00000020  40 00 00 00 00 00 00 00  80 81 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1a 00 19 00  |....@.8...@.....|
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |@.......@.......|
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00  |................|
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00  |................|
```

```
readelf -h /bin/ls | grep Magic
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

Structure of an ELF binary



Extensive details on the ELF format can be found here or via the command: `man elf.5`

Structure of an ELF binary

We can see how sections are mapped to (memory) segments via:

```
$ readelf -l /bin/ls

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x5eb0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000003520 0x0000000000003520  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000013991 0x0000000000013991  R E    0x1000
  LOAD           0x0000000000018000 0x0000000000018000 0x0000000000018000
                 0x00000000000075e8 0x00000000000075e8  R      0x1000
  LOAD           0x000000000001ffb0 0x0000000000020fb0 0x0000000000020fb0
                 0x00000000000012c8 0x00000000000025b0  RW     0x1000
   [...]
```

Structure of an ELF binary

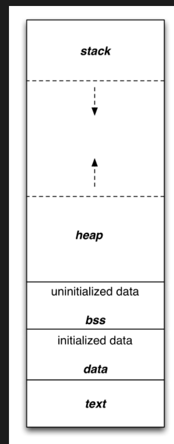
We can see how sections are mapped to (memory) segments via:

```
$ readelf -l /bin/ls

   [...]

Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym
.dynstr .gnu.version .gnu.version_r .rela.dyn
   03     .init .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init_array .fini_array .data.rel.ro .dynamic .got
```

Virtual memory layout

ELF segments are loaded into virtual memory before running a binary. The virtual memory of a loaded program has the following layout and contains:

- **text** segment: executable code of binary (read-execute)
- **data** segment: initialized variables of binary (read-write)
- **bss** segment: uninitialized static data of binary (read-write)
- **heap** segment: dynamically allocated memory (read-write)
- **stack** segment: call stack (read-write)



The call stack

Function calls via the stack

C code:

```c
int add(int a, int b){
  int sum = a + b;
  return sum;
}

int main(void){
  int s;
  s = add(8,2);
  return 0;
}
```

gcc -O0 ⟶

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

7fffffffe3b8:	Some data	← rsp

Registers:

`rbp: 1, rsp: 7fffffffe3b8`

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rsp

Registers:

`rbp: 1, rsp: 7fffffffe3b0`

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rsp, rbp

Registers:

rbp: 7fffffffe3b0   rsp: 7fffffffe3b0

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	← rsp

Registers:
rbp: 7fffffffe3b0 rsp: 7fffffffe3a0

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe3a0
rsi: 2
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe3a0
rsi: 2   rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	
7fffffffe398:	Return address: 401137	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe398
rsi: 2   rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Frame pointer of main: 7fffffffe3b0	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe390
rsi: 2   rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Frame pointer of main: 7fffffffe3b0	← rsp, rbp

Registers:

```
rbp: 7fffffffe390    rsp: 7fffffffe390
rsi: 2    rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	Some data		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Some data	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	Some data		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Value of rsi: 2	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	Some data		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Value of rsi: 2	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8   edx: 8
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	Some data		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Value of rsi: 2	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8   edx: 8   eax: 2
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	Some data		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Value of rsi: 2	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8   edx: 8   eax: 10
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content		
7fffffffe3b8:	Some data		
7fffffffe3b0:	0x1		
7fffffffe3a8:	Some data		
7fffffffe3a0:	Some data		
7fffffffe398:	Return address: 401137		
7fffffffe390:	Frame pointer of main: 7fffffffe3b0		← rsp, rbp
7fffffffe388:	10		
7fffffffe380:	Some data		
7fffffffe378:	Value of rdi: 8	Value of rsi: 2	

Registers:

```
rbp:  7fffffffe390   rsp:  7fffffffe390
rsi:  2   rdi:  8   edx:  8   eax:  10
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Frame pointer of main: 7fffffffe3b0	← rsp, rbp
7fffffffe388:	10	
7fffffffe380:	Some data	
7fffffffe378:	Value of rdi: 8 / Value of rsi: 2	

Registers:

```
rbp: 7fffffffe390   rsp: 7fffffffe390
rsi: 2   rdi: 8   edx: 8   eax: 10
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	
7fffffffe398:	Return address: 401137	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe398
rsi: 2   rdi: 8   edx: 8   eax: 10
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Value	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Some data	
7fffffffe3a0:	Some data	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe3a0
rsi: 2   rdi: 8   edx: 8   eax: 10
rip: 401137
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Value of eax: 10	
7fffffffe3a0:	Some data	← rsp

Registers:

```
rbp: 7fffffffe3b0   rsp: 7fffffffe3a0
rsi: 2   rdi: 8   edx: 8   eax: 10
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Address	Content	
7fffffffe3b8:	Some data	
7fffffffe3b0:	0x1	← rbp
7fffffffe3a8:	Value of eax: 10	
7fffffffe3a0:	Some data	← rsp

Registers:

```
rbp:  7fffffffe3b0   rsp:  7fffffffe3a0
rsi:  2   rdi:  8   edx:  8   eax:  0
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

7fffffffe3b8:	Some data	← rsp

Registers:

```
rbp: 1   rsp: 7fffffffe3b8
rsi: 2   rdi: 8   edx: 8   eax: 0
```

Function calls via the stack

Assembler code:

```
00401106 <add>:
 401106: push rbp
 401107: mov rbp,rsp
 40110a: mov DWORD PTR [rbp-0x14],edi
 40110d: mov DWORD PTR [rbp-0x18],esi
 401110: mov edx,DWORD PTR [rbp-0x14]
 401113: mov eax,DWORD PTR [rbp-0x18]
 401116: add eax,edx
 401118: mov DWORD PTR [rbp-0x4],eax
 40111b: mov eax,DWORD PTR [rbp-0x4]
 40111e: pop rbp
 40111f: ret

00401120 <main>:
 401120: push rbp
 401121: mov rbp,rsp
 401124: sub rsp,0x10
 401128: mov esi,0x2
 40112d: mov edi,0x8
 401132: call 1119 <add>
 401137: mov DWORD PTR [rbp-0x4],eax
 40113a: mov eax,0x0
 40113f: leave
 401140: ret
```

Call stack:

Registers:

`rsi: 2   rdi: 8   edx: 8   eax: 0`

Stackbased buffer overflows

Historical background

Software vulnerabilities via **stack buffer overflows** have already been openly addressed in a seminal work from 1996:

Smashing the Stack for Fun and Profit

Buffer overflows visualized

Buffer overflows occur when a program or process tries to write or read more data to/from a buffer than the buffer can hold.

Example:

```
int keyVariable;
char nameBuffer[12];
```

24 bytes somewhere on stack

Buffer overflows visualized

Buffer overflows occur when a program or process tries to write or read more data to/from a buffer than the buffer can hold.

Example:

```c
int keyVariable;
char nameBuffer[12];
```

Buffer overflows visualized

Buffer overflows occur when a program or process tries to write or read more data to/from a buffer than the buffer can hold.

Example:

```
int keyVariable;
char nameBuffer[12];
strcpy(nameBuffer, "TooBigForYourBuffer!")
```

Buffer overflows visualized

Buffer overflows occur when a program or process tries to write or read more data to/from a buffer than the buffer can hold.

Example:

```
int keyVariable;
char nameBuffer[12];
strcpy(nameBuffer, "TooBigForYourBuffer!")
```

nameBuffer												Unused				keyVariable							
T	o	o	B	i	g	F	o	r	Y	o	u	r	B	u	f	f	e	r	!	\0	0	0	1

(we assume **big endianess** for the moment!)

Buffer overflows visualized

Buffer overflows occur when a program or process tries to write or read more data to/from a buffer than the buffer can hold.

Example:

```
int keyVariable;
char nameBuffer[12];
strcpy(nameBuffer, "TooBigForYourBuffer!")
```

$\Rightarrow$ keyVariable $= 7378429050575912961 \neq 1$

(we assume **big endianess** for the moment!)

Insecure functions

Insecure functions:

Some examplary C functions that assume that buffer size checks are handled by the programmer:

- `gets`
- `strcpy`
- `strcat`
- `memcpy`
- `sprintf`
- ...

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
 40122f: mov DWORD PTR [rbp-0x4],0x0
 401236: lea rax,[rbp-0x18]
 40123a: mov rdi,rax
 40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	Some data	
7fffffffe380:	Some data	
7fffffffe378:	Some data	
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	0x0 ≙ Boolean false	
7fffffffe380:	Some data	
7fffffffe378:	Some data	
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	0x0 ≙ Boolean false	
7fffffffe380:	Some data	
7fffffffe378:	Some data	← rax
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	0x0 ≙ Boolean false	
7fffffffe380:	Some data	
7fffffffe378:	Some data	← rax, rdi
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
 40122f: mov DWORD PTR [rbp-0x4],0x0
 401236: lea rax,[rbp-0x18]
 40123a: mov rdi,rax
 40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	0x0 ≙ Boolean false	
7fffffffe380:	Some data	
7fffffffe378:	Some data	← rax, rdi
	⋮	

User input: AAAAAAAAAAAAAAAAAA (18 bytes)

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address									
7fffffffe398:	Return address: 401137								
7fffffffe390:	Saved frame pointer								← rbp
7fffffffe388:	0	0	0	0	0	0	A	A	
7fffffffe380:	A	A	A	A	A	A	A	A	
7fffffffe378:	A	A	A	A	A	A	A	A	← rax, rdi

⋮

User input: AAAAAAAAAAAAAAAAAA (18 bytes)

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values** ←
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow

Assembler code:

```
...
 40122f: mov DWORD PTR [rbp-0x4],0x0
 401236: lea rax,[rbp-0x18]
 40123a: mov rdi,rax
 40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	0x4141 ≙ Boolean true	
7fffffffe380:	AAAAAAAA	
7fffffffe378:	AAAAAAAA	← rax, rdi
	⋮	

User input: AAAAAAAAAAAAAAAAAA (18 bytes)

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values ←
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	int keyVariable = 0x0	
7fffffffe380:	Some data	
7fffffffe378:	Some data	← rax, rdi
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values ←
- overwrite **return address** to modify program flow

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	int keyVariable = 0xdeadbeef	
7fffffffe380:	AAAAAAAA	
7fffffffe378:	AAAAAAAA	← rax, rdi
	⋮	

User input: ???

Hint: Mind the endianness of the computer!

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow ←

Assembler code:

```
...
 40122f: mov DWORD PTR [rbp-0x4],0x0
 401236: lea rax,[rbp-0x18]
 40123a: mov rdi,rax
 40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	Return address: 401137	
7fffffffe390:	Saved frame pointer	← rbp
7fffffffe388:	Some data	
7fffffffe380:	Some data	
7fffffffe378:	Some data	← rax, rdi
	⋮	

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow ←

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Value	
7fffffffe398:	New return address: 40128a	
7fffffffe390:	AAAAAAAA	← rbp
7fffffffe388:	AAAAAAAA	
7fffffffe380:	AAAAAAAA	
7fffffffe378:	AAAAAAAA	← rax, rdi
	⋮	

User input: ???

Hint: Mind the endianness of the computer!

Stack buffer overflows

Buffer overflows on the stack can be used to:

- change **Boolean values**
- overwrite **local variables** with exact values
- overwrite **return address** to modify program flow ←

Even better idea: Write **shellcode** onto stack!

Assembler code:

```
...
40122f: mov DWORD PTR [rbp-0x4],0x0
401236: lea rax,[rbp-0x18]
40123a: mov rdi,rax
40123d: call 4011d8 <gets>
...
```

Call stack:

Address	Content	
7fffffffe398:	New return address: 7fffffffe378	
7fffffffe390:	<Shellcode>	← rbp
7fffffffe388:	<Shellcode>	
7fffffffe380:	<Shellcode>	
7fffffffe378:	<Shellcode>	← rax, rdi
	⋮	

User input: ???

Hint: Mind the endianness of the computer!

Useful toolz

How to analyze and exploit binaries?

The following list shows some **useful tools** for binary exploitation:

- GNU compiler collection (`gcc`) → (re-)compile source code
- GNU debugger (`gdb`) → inspect binary at runtime
- `objdump` → display static information about binary
- `strings` → show all strings in binary
- `netcat` → communicate with binary via network
- `checksec` → show security properties of binary
- Python (`pwntools`) → write complex exploits
- `ghidra` → disassemble, decompile, and analyze binaries

objdump in a nutshell

Commands for (statically) disassembling binary:

Option	Description
`--disassemble=<function>`	disassemble specific function
`-D`	disassemble all sections
`-M intel | att`	use specific assembler syntax

gdb in a nutshell

Commands for controlling program flow:

Command (Abbreviation)	Description
`run (r)`	run binary
`break (b) <function>`	set breakpoint at entry of function
`break (b) *<address>`	set breakpoint at memory address
`continue (c)`	continue execution (after break)
`step (si)`	step one instruction (follow call instructions)
`next (ni)`	step one instruction (step over call instructions)

gdb in a nutshell

Commands for disassembling binary:

Command (Abbreviation)	**Description**
`disassemble <function>` (`disas`)	disassemble function
`set disassembly-flavor intel | att`	set disassembler syntax to Intel / AT&T style

gdb in a nutshell

Commands for examining registers and memory:

Command (Abbreviation)	**Description**
`x/nfu <address>`	examine number (`n`) of memory units (`u`) with specified format (`f`)
`x/s 0x8048770`	print string at memory address
`x/4xg 0x4032016`	print four 8-byte words (giants) at memory address as hexadecimal
`x/2dw 0x4018158`	print two 4-byte words at memory address as decimal
`info registers` (`i r`)	print (almost) all registers
`print` (`p`) `$<register>`	print value of register

How to provide byte values input for binary?

We need to distinguish **two cases:**

1. input is provided as argument to the main function

```
./program $(echo -e '\xde\xad\xbe\xef')
```

2. input is provided via stdin

```
echo -e '\xde\xad\xbe\xef' | ./program # closes stdin!!!

# better idea
echo -e '\xde\xad\xbe\xef' > ./byte_sequence
cat ./byte_sequence - | ./program # keeps stdin open

# for remote challenges
cat ./byte_sequence - | nc challenge.domain.com 1337
```

The final slide

Now it is time to get our hands dirty...
...and hack into these binaries!

Questions? Feedback? Suggestions?

daniel.tenbrinck@fau.de