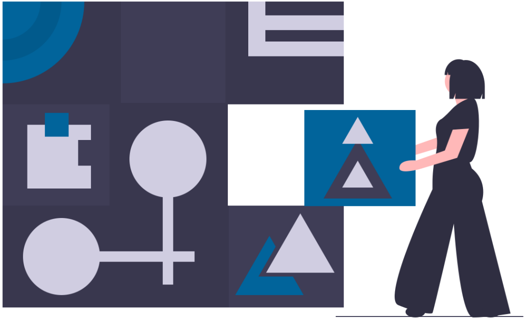
Stack-Based Buffer Overflows

and Execution of Shell Code

Benjamin Medicke

Master IT Security
IT-Sicherheit

FH University of Applied Sciences TECHNIKUM WIEN



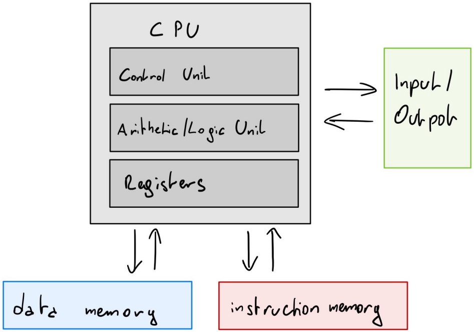
What to Expect

0 A Bit of Historical Context
1 Neccessary Theory
2 Practical Demostration
3 Sources and Further Reading

How Did We Get Here? | Data and Instructions

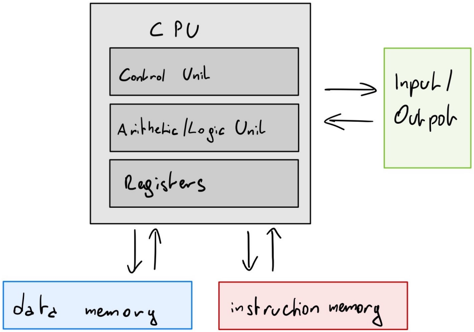
- programs are data and instructions
- the CPU needs access to both
- registers are the fastest memory
  - general purpose registers
  - instruction pointer
  - base and stack pointer

How Did We Get Here? | Architectures

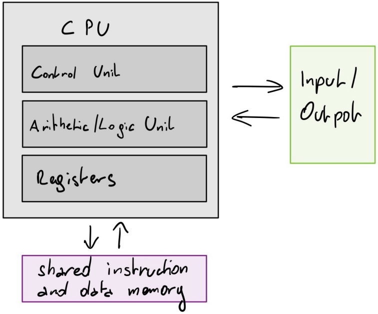


Harvard Architecture

How Did We Get Here? | Architectures



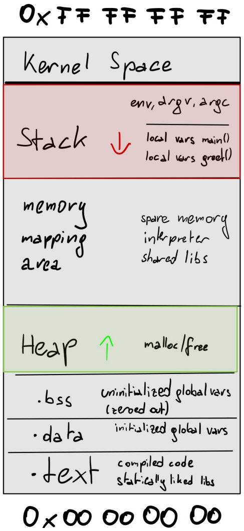
Harvard Architecture



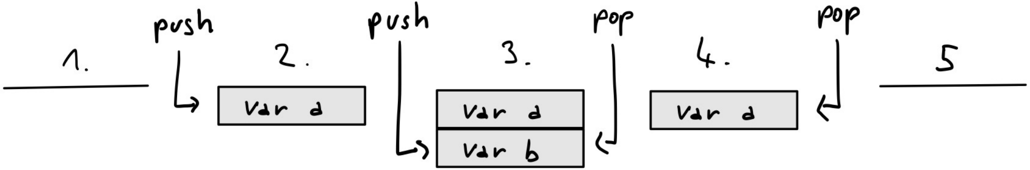
Von Neumann Architecture

Memory Layout

- the Heap grows up
- the Stack grows down
- instructions and data reside in the same memory



How a Stack Works



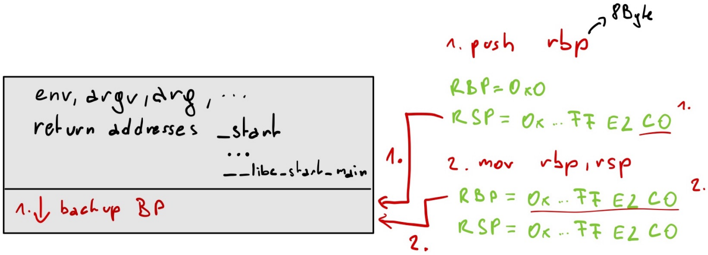
Stack Frames

env, argv, arg, ...
return addresses _start
...
__libc_start_main

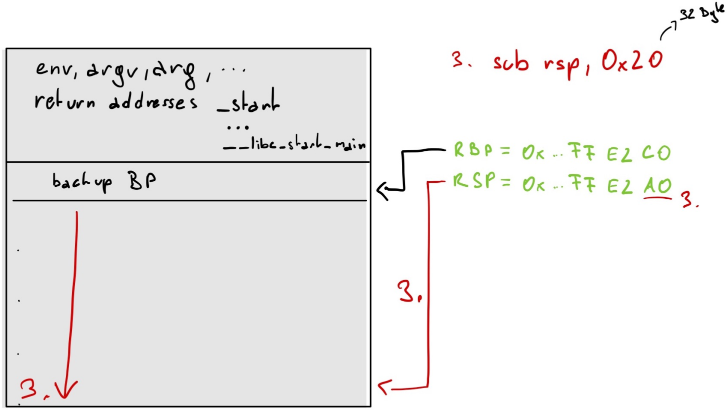
main+0

RBP = 0x0
RSP = 0x...FF E2 C8

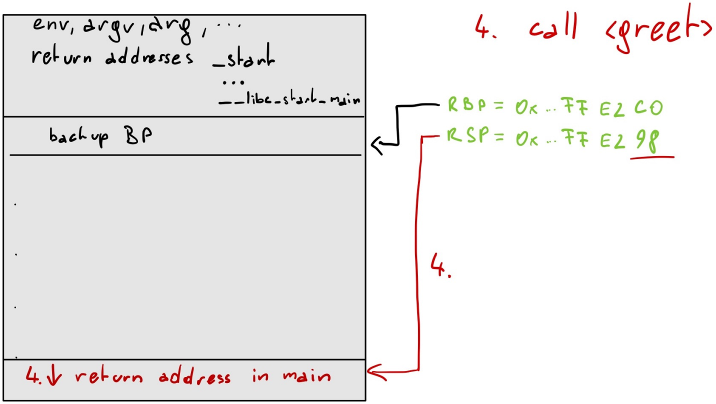
Stack Frames



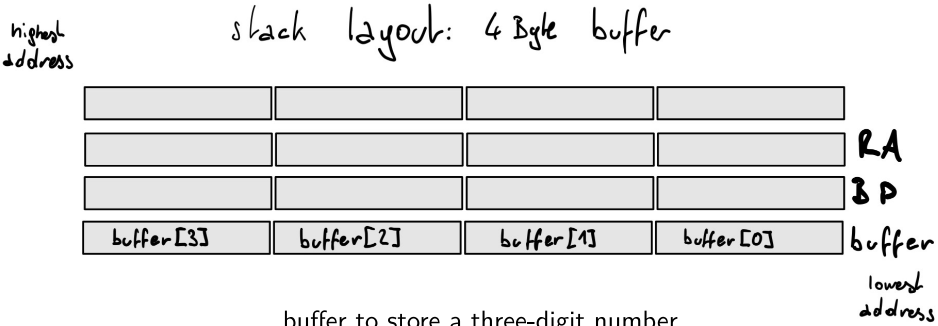
Stack Frames



Stack Frames

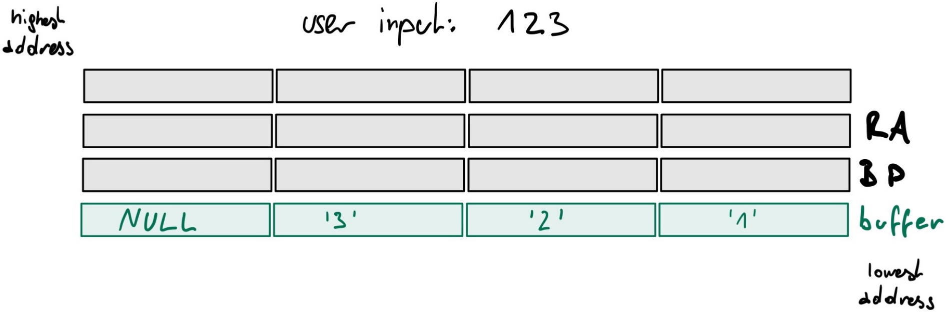


Buffer on the Stack | Layout

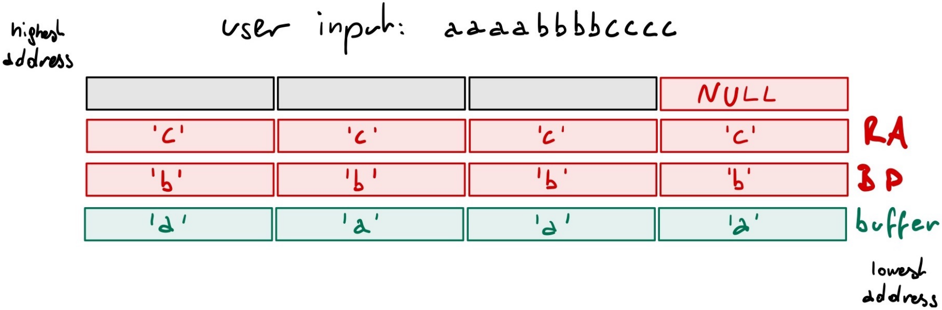


buffer to store a three-digit number

Buffer on the Stack | Valid Input



Buffer on the Stack | Overflow



Buffer on the Stack | Overflow | GDB GEF

```
$eax   : 0x1
$ebx   : 0x0
$ecx   : 0x30
$edx   : 0xffffd42c  →  "aaaabbbbcccc"
$esp   : 0xffffd438  →  0x00000000
$ebp   : 0x62626262 ("bbbb"?) a.
$esi   : 0xf7fbd000  →  0x001e6d6c
$edi   : 0xf7fbd000  →  0x001e6d6c
$eip   : 0x63636363 ("cccc"?) b.
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffffd454│+0x001c: 0x00000000
0xffffd450│+0x0018: 0xf7fbd000  →  0x001e6d6c
0xffffd44c│+0x0014: 0xffffd464  →  0x00000000
0xffffd448│+0x0010: 0xffffd4dc  →  0xffffd66c  →  "SHELL=/bin/bash"
0xffffd444│+0x000c: 0xffffd4d4  →  0xffffd636  →  "/home/ben/projects/reed/exploit-development/check_[...]"
0xffffd440│+0x0008: 0x00000001
0xffffd43c│+0x0004: 0xf7df4ee5  →  <__libc_start_main+245> add esp, 0x10
0xffffd438│+0x0000: 0x00000000   ← $esp
─────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x63636363
─────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "check_pin", stopped 0x63636363 in ?? (), reason: SIGSEGV c.
```

Demo | Buffer Overflow in rot13.c

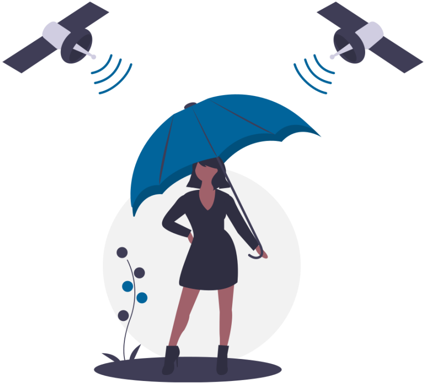
github.com/bmedicke/REED/blob/master/exploit-development/rot13.c

Demo | Takeaway

- consider a language with bounds checking
- make use `-fsanitize=address` (GCC, Clang)
- fuzz your own programs
- above all avoid unsafe functions such as
  - `scanf()`
  - `strcpy()`
  - `sprintf()`
  - `strcat()`
  - ...

Kur | Visualizing a Payload with Radare2 in and out of Memory

Kur | Countermeasures (and how to break them)



Sources and Further Reading | Books

[1] Dennis Andriesse and San Francisco. *PRACTICAL BINARY ANALYSIS Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. 2nd ed. No Starch Press, 2019, p. 431. ISBN: 1593279124. URL: https://lccn.loc.gov/2018040696.

[2] Jo Van Hoey. *Beginning x64 Assembly Programming: From Novice to AVX Professional Paperback*. 1st ed. Apress, 2019, p. 436. ISBN: 978-1484250754.

[3] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*. Vol. 3. 253665. 2011. DOI: 10.1109/MAHC.2010.22.

Sources and Further Reading | Other

[4] Benjamin Medicke. *RE 🌾 ED - notes about Reverse Engineering and Exploit Development*. URL: `https://github.com/bmedicke/reed`.

[5] John Von Neumann and Michael D. Godfrey. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. ISSN: 10586180. DOI: 10.1109/85.23838