CMSC414 Computer and Network Security

Mitigations and Tutorial

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 8, 2024

Agenda

- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations
- Demo related to Project 1

Pointer Authentication

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)
- **Pointer Authentication:** place some secret value in the pointers

Pointer Authentication

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)
- **Pointer Authentication:** place some secret value in the pointers
  - In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits

Pointer Authentication

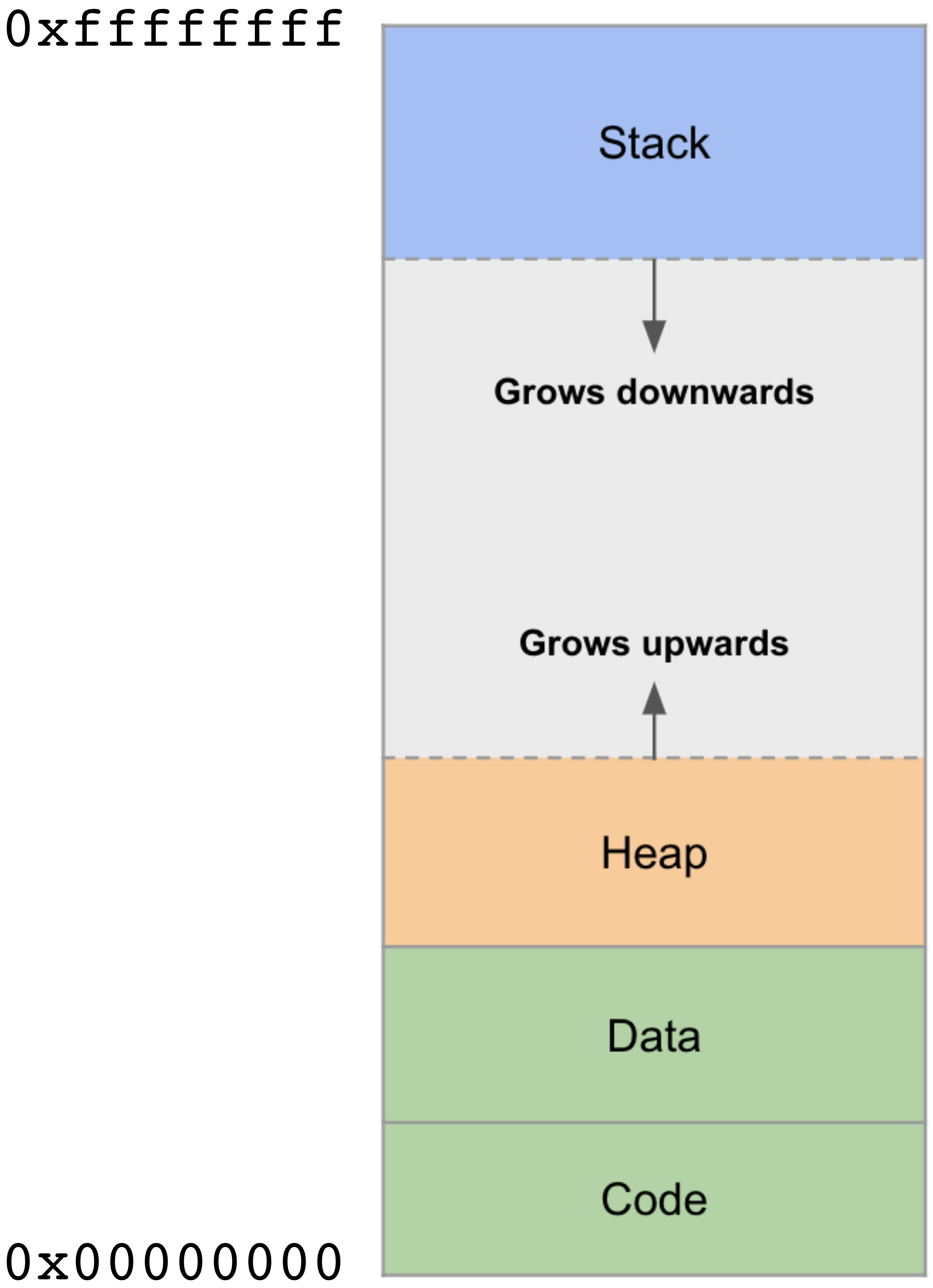
- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)

- **Pointer Authentication:** place some secret value in the pointers
  - In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits
  - Before using the pointer in memory, check if the PAC is still valid
    - Invalid: crash the program
    - Valid: restore unused bits, use the address normally

Properties of PAC

- Each possible address has its own PAC
- Message Authentication Code (MAC) in the cryptography lectures
- Only someone who knows the CPU’s master secret can generate a PAC for an address
- The CPU’s master secret is not accessible to the program
    - Leaking program memory will not leak the master secret

Address Space Layout Randomization



0xffffffff

Stack

Grows downwards

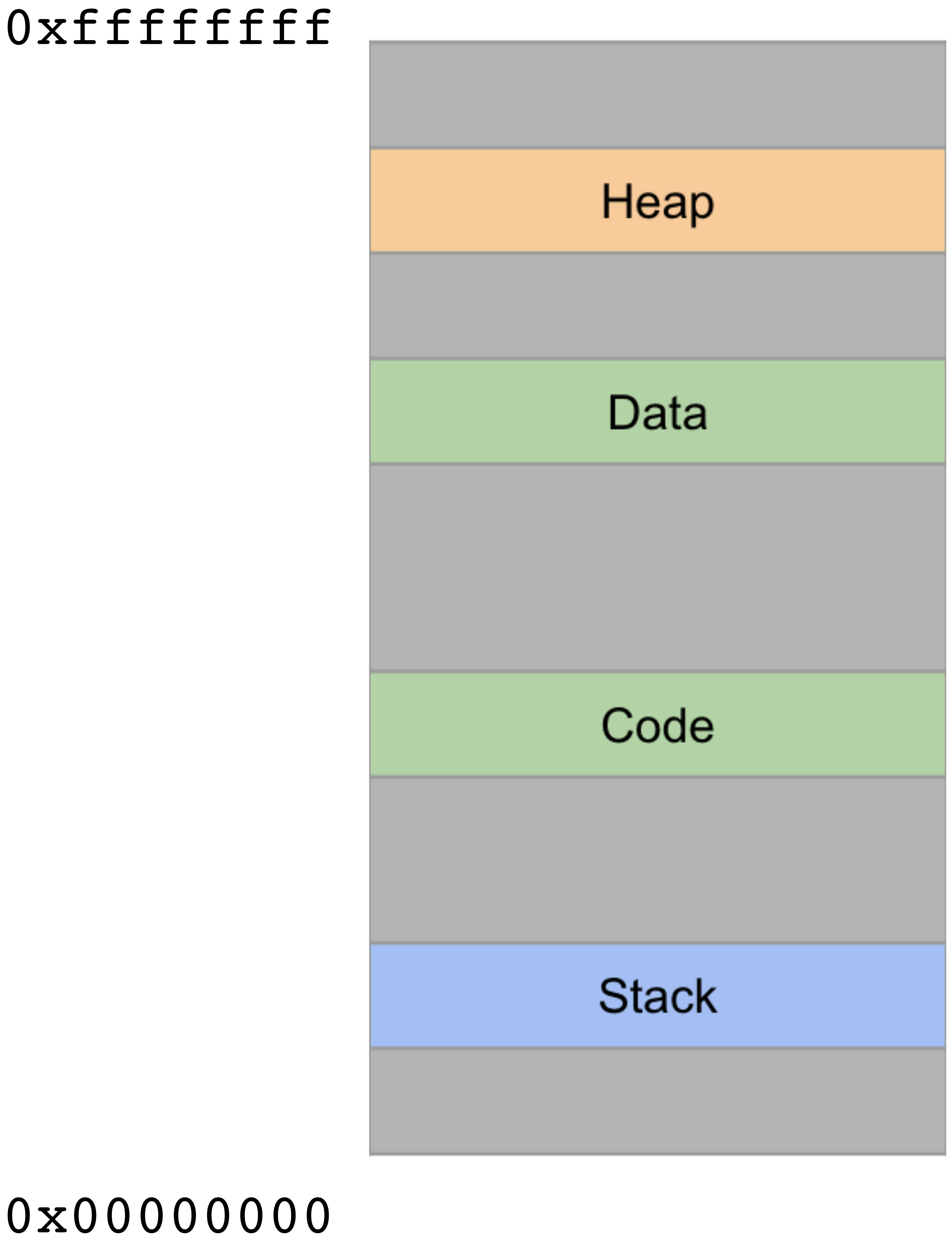
Grows upwards

Heap

Data

Code

0x00000000



0xffffffff

Heap

Data

Code

Stack

0x00000000

Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead

Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead
- However…
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)
  - Leak the address of a pointer, whose address relative to your shellcode is known (stack pointer, RIP)
  - Guess the address of your shellcode: Brute-force

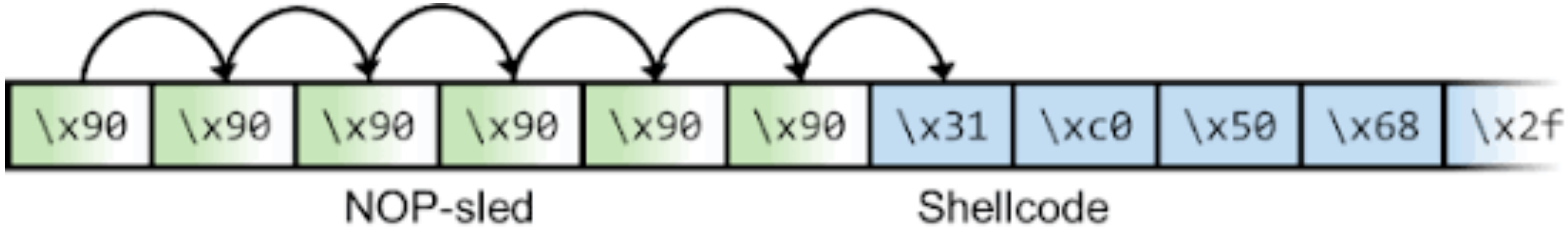
Combining Mitigations

- **Defense in depth**
- Example: Combining ASLR and non-executable pages
- To defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities
  - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
  - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)

Agenda

- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations
- Demo related to Project 1

NOP Slide



- NOP: no operation
  - "slide" the CPU's instruction execution flow to its final, desired destination
- Return instruction pointer to anywhere in NOP can then execute the Shellcode

https://www.coengoedegebure.com/buffer-overflow-attacks-explained/

https://en.wikipedia.org/wiki/NOP_slide

`run <input>`

Run the program with `input` as the command-line arguments

`print <var>`
(or just "**`p <var>`**")

Print the value of variable `var`
(Can also do some operations: `p  &x`)

`b <function>`

Set a breakpoint at `function`

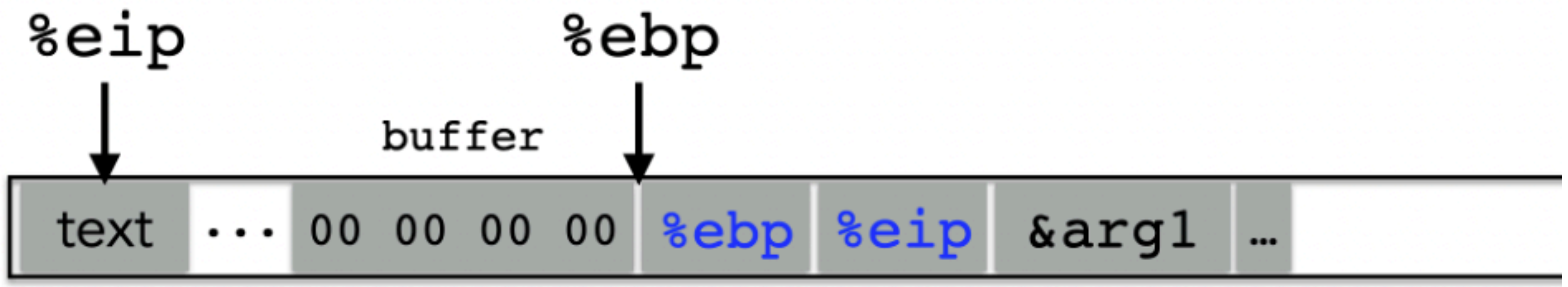
`s`
`c`

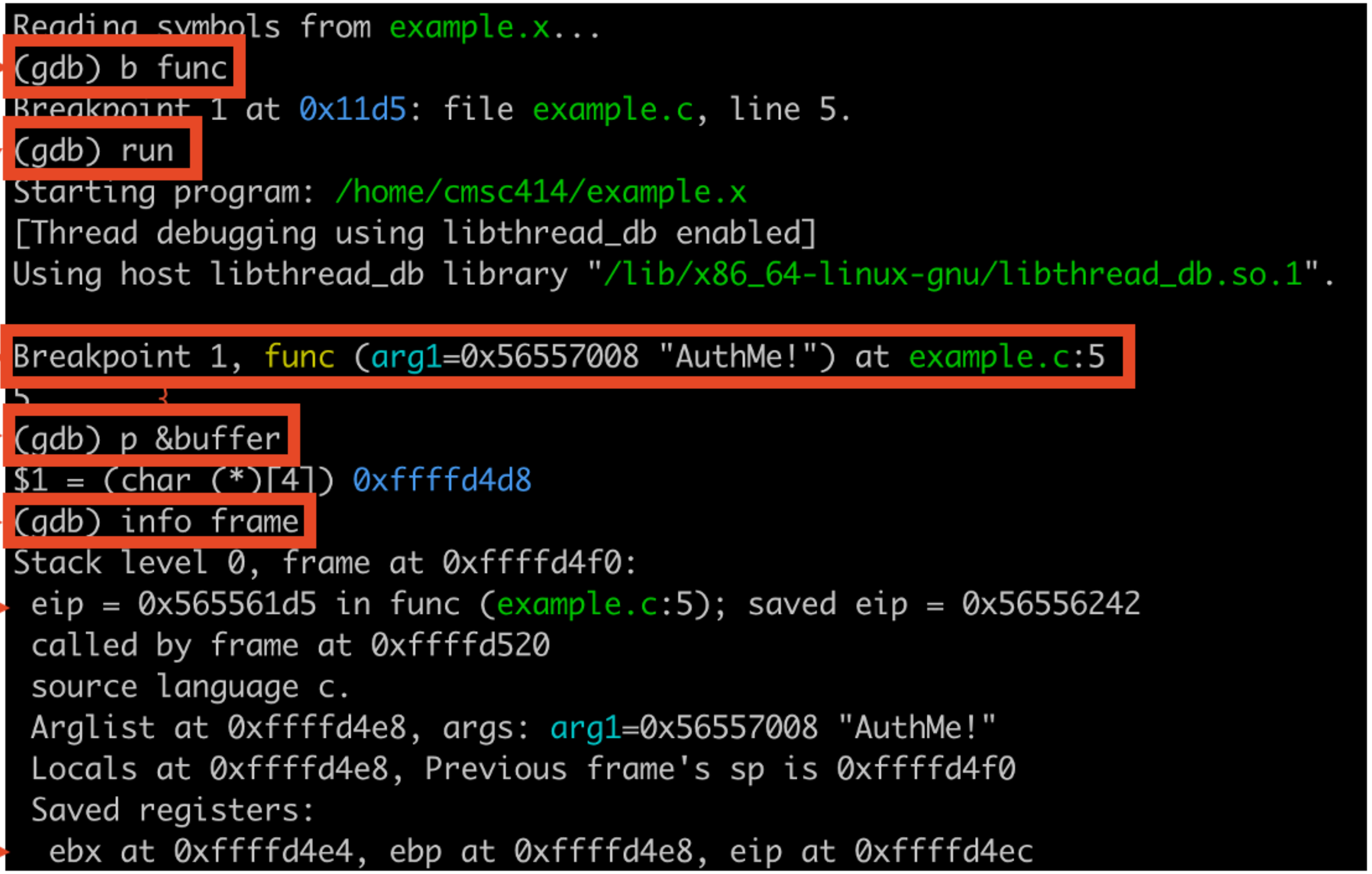
step through execution (into calls)
continue execution (no more stepping)

`info frame` (or just "`i f`")	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`info reg` (or just "`i r`")	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>

gdb example

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```





Set a breakpoint at func()

Run the program

Breakpoint reached

Print buffers' addr

Frame info

Current/saved eip

Where on the stack registers are saved

```
Reading symbols from example.x...
(gdb) b func
Breakpoint 1 at 0x11d5: file example.c, line 5.
(gdb) run
Starting program: /home/cmsc414/example.x
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (arg1=0x56557008 "AuthMe!") at example.c:5
5
(gdb) p &buffer
$1 = (char (*)[4]) 0xffffd4d8
(gdb) info frame
Stack level 0, frame at 0xffffd4f0:
 eip = 0x565561d5 in func (example.c:5); saved eip = 0x56556242
 called by frame at 0xffffd520
 source language c.
 Arglist at 0xffffd4e8, args: arg1=0x56557008 "AuthMe!"
 Locals at 0xffffd4e8, Previous frame's sp is 0xffffd4f0
 Saved registers:
  ebx at 0xffffd4e4, ebp at 0xffffd4e8, eip at 0xffffd4ec
```

Tutorial on Computer