Memory Corruption

Memory corruption: because sometimes your variables just want to explore the stack on their own. 🎉

Assumptions going forward

- You have can use linux commands and work within the system
- You understand assembly or can figure it out
- You can debug using gdb and figure out what a program is doing

Class Format

- Lecture followed by demos
- Ideally one of the classes per week is dedicated to working through any problems. If there is a lot of content we will have additional short lectures.
- Office hours:
    - Monday - me
    - Wednesday - Jonah
    - Friday - Mohamed

Today -

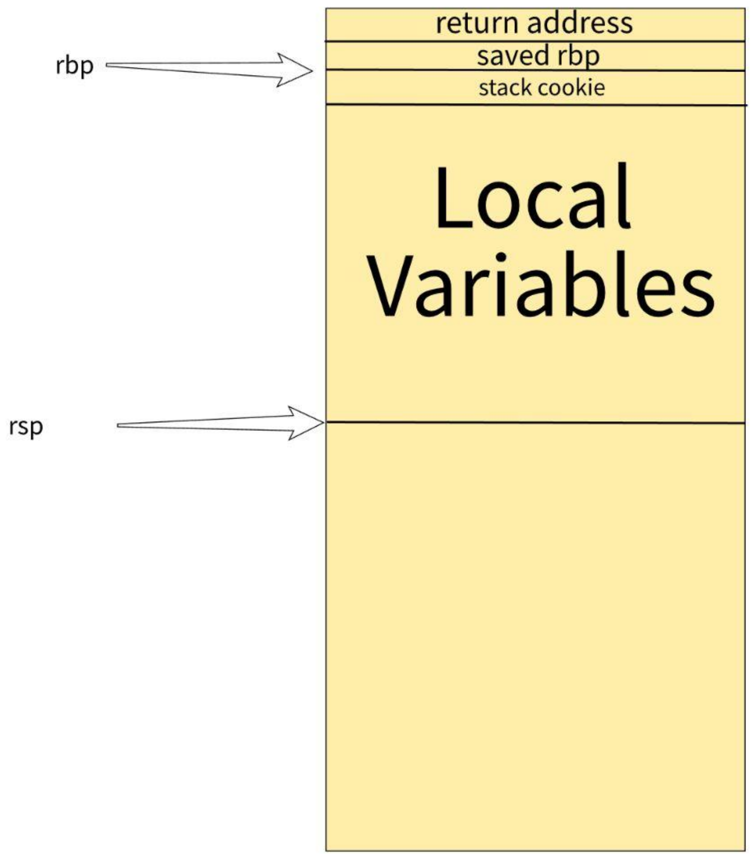
- checksec
- Overflow local variable
- Overflow return address
- Overflow return address with conditions.
- Overflow return address middle.
- ASLR/PIE
- ASLR Defeats

- `checksec` is a tool to analyze binary protections.
- Displays details about:
  - **NX** (Non-Executable Stack)
  - **ASLR** (Address Space Layout Randomization)
  - **Stack Canaries**
  - **RELRO** (Relocation Read-Only)

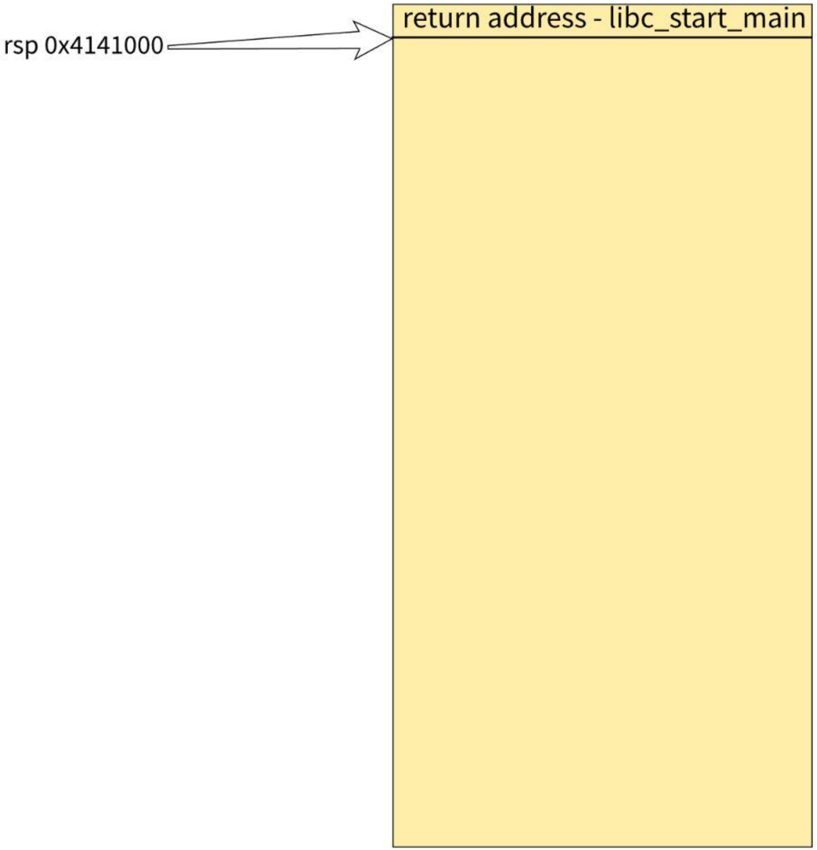
Generic stack in a function

```c
int main(int argc, char **argv) {
  int local_integer;
  char local_char;

  return 0;
}
```


return address
saved rbp
rbp
stack cookie
Local Variables
rsp

overflow_local

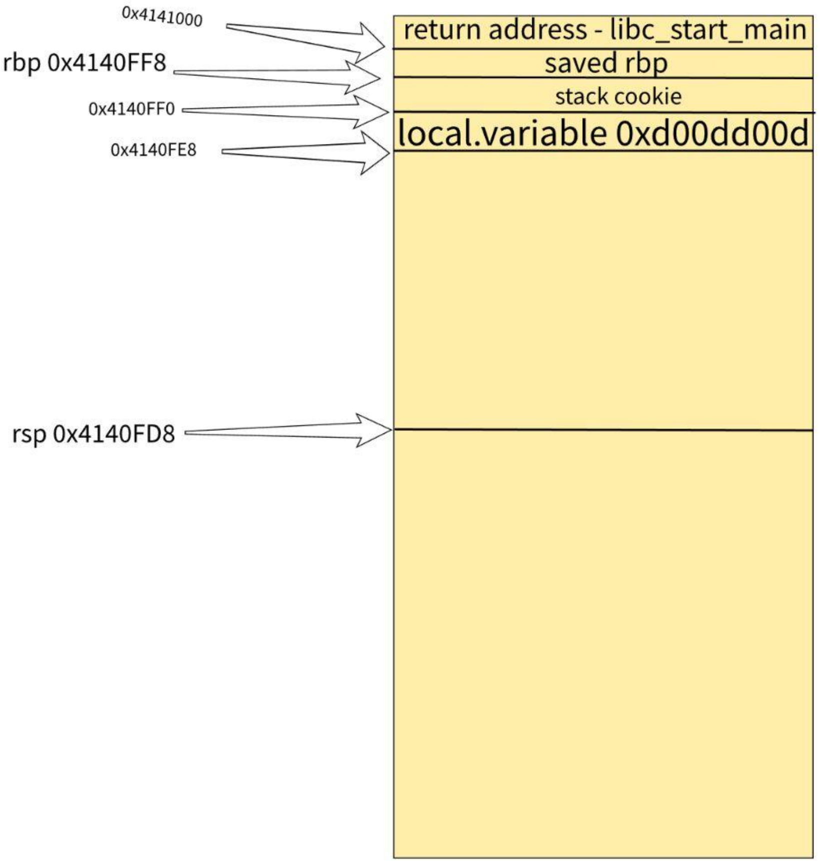

return address - libc_start_main
rsp 0x4141000

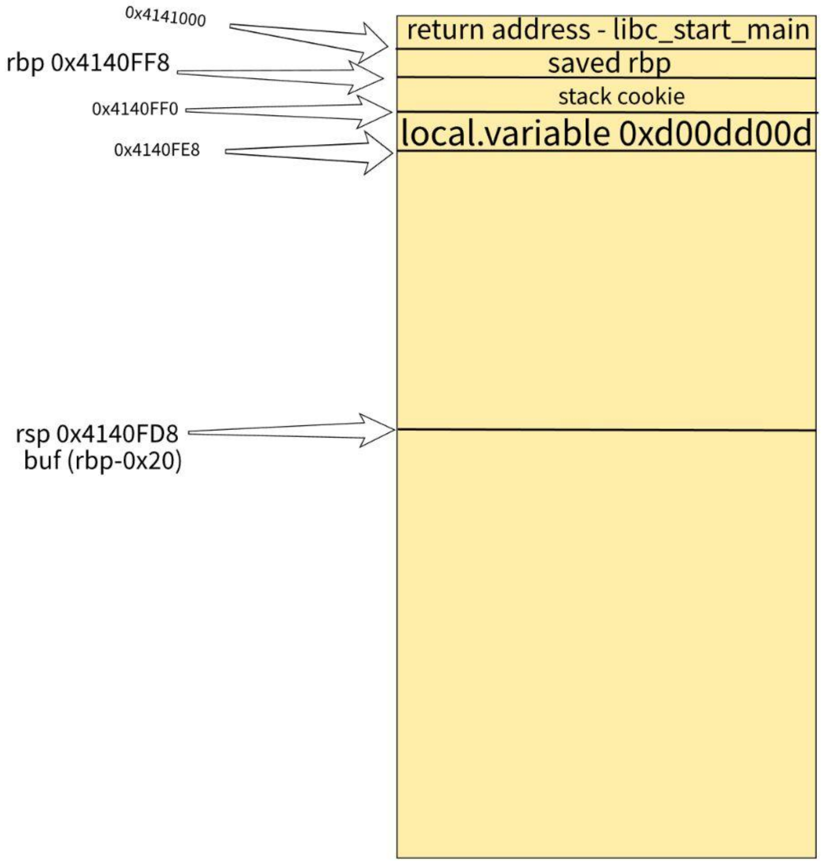

return address - libc_start_main
saved rbp
rsp 0x4140FF8

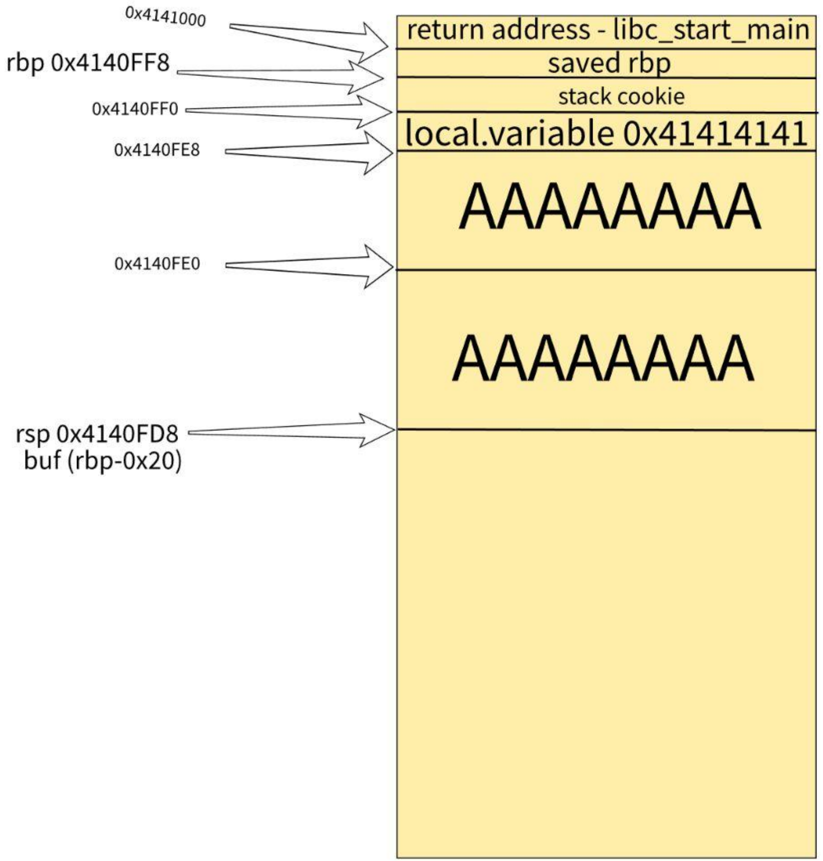

return address - libc_start_main
saved rbp
rbp 0x4140FF8
rsp 0x4140FF8


return address - libc_start_main
saved rbp
rbp 0x4140FF8
rsp 0x4140FD8

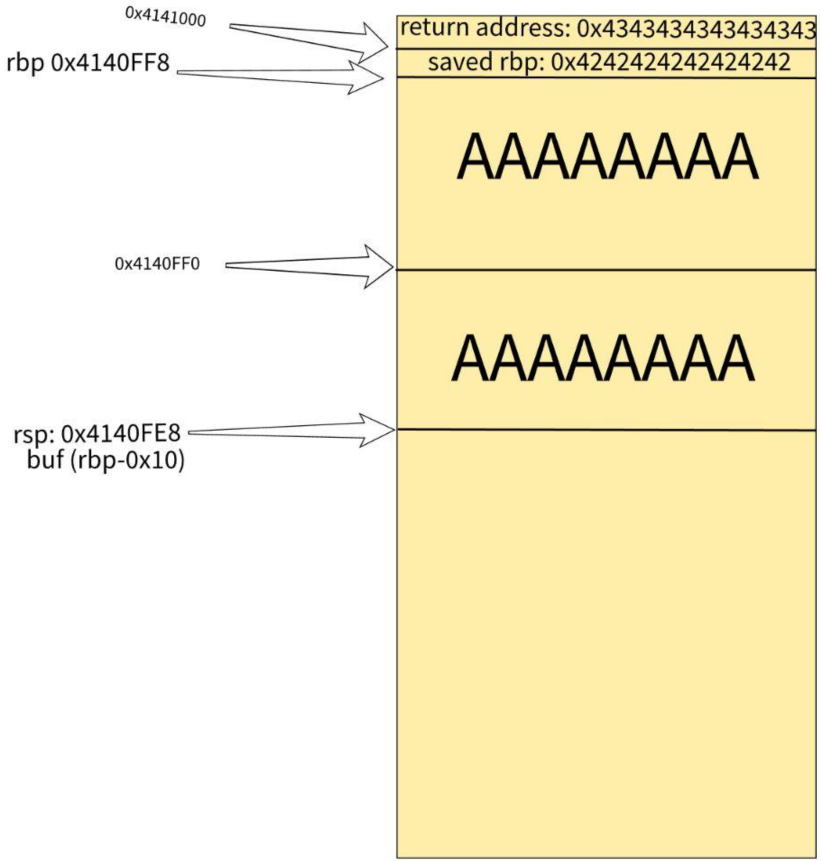

return address - libc_start_main
saved rbp
stack cookie
rbp 0x4140FF8
rsp 0x4140FD8

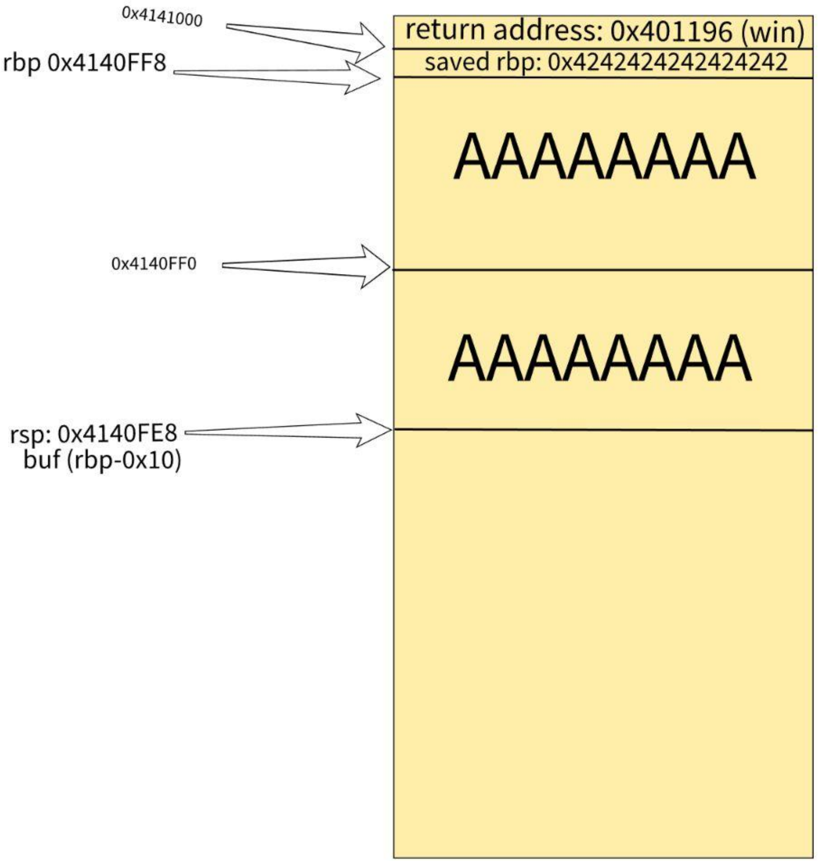

0x4141000
rbp 0x4140FF8
0x4140FF0
0x4140FE8
rsp 0x4140FD8
return address - libc_start_main
saved rbp
stack cookie
local.variable 0xd00dd00d


0x4141000
rbp 0x4140FF8
0x4140FF0
0x4140FE8
rsp 0x4140FD8
buf (rbp-0x20)
return address - libc_start_main
saved rbp
stack cookie
local.variable 0xd00dd00d


0x4141000
rbp 0x4140FF8
0x4140FF0
0x4140FE8
0x4140FE0
rsp 0x4140FD8
buf (rbp-0x20)
return address - libc_start_main
saved rbp
stack cookie
local.variable 0x41414141
AAAAAAAA
AAAAAAAA

overflow_return


0x4141000
rbp 0x4140FF8
return address: 0x4343434343434343
saved rbp: 0x4242424242424242
AAAAAAAA
0x4140FF0
AAAAAAAA
rsp: 0x4140FE8
buf (rbp-0x10)


0x4141000
rbp 0x4140FF8
return address: 0x401196 (win)
saved rbp: 0x4242424242424242
AAAAAAAA
0x4140FF0
AAAAAAAA
rsp: 0x4140FE8
buf (rbp-0x10)

overflow_return_with_conditions

overflow_return_middle

Address Space Layout Randomization

Position Independent Executable

Introduction to ASLR and PIE

What is ASLR?

- **ASLR (Address Space Layout Randomization)** is a security feature that randomizes memory addresses to make exploitation harder.
- Prevents predictable memory layout attacks.

What is PIE?

- **PIE (Position Independent Executable)** allows executables to be loaded at different addresses, enabling ASLR for the main binary.
- Ensures the entire program benefits from ASLR, not just shared libraries.

How ASLR Works

Memory Layout with ASLR Enabled

- When a program runs, ASLR randomizes memory locations of:
  - **Stack** 🏗️ (Function calls, local variables)
  - **Heap** 💾 (Dynamically allocated memory)
  - **Shared libraries** 📦 (e.g., libc, libm)
  - **Executable binary (if PIE is enabled)** 🔄

How PIE Works

Non-PIE Executable (Fixed Address)

- Traditional executables have a fixed base address.
- The binary is loaded at the same location every time.
- Example:

0x400000 -> main binary (fixed location)

PIE Executable (Randomized Address)

- Compiling with PIE allows ASLR to randomize the base address of the binary.
- Example:

0x5f0000 -> main binary (different location every time)

Summary

✅ **ASLR randomizes memory addresses** to prevent predictable exploits.
✅ **PIE enables ASLR for the binary itself** by making it position-independent.
✅ **Check protections with `checksec` and GDB mappings**.
✅ **Bypassing requires memory leaks, partial overwrites, or brute force techniques**.

Defeats:

- Partial Overwrites
- Data Leaks

Partial Overwrites

Check out the address of win()

What changes?
What is constant?

How can we leverage this?

What is a big problem though?

Leaking Data

- The flag itself
- Addresses
- Canaries/Cookies

Unitialized data

- Leak info
- Cause effects

Like a child, the stack doesn’t clean up after itself.

- Cookie leak

- Cause effect

Midterm:

- Drops @ 3pm EST on Friday February 7th via pwn.college
- Will be open until 3 pm EST Friday February 14th
- 5 questions
- 30% of your final grade

Midterm: What you need to know

- Assembly programming
- Reverse Engineering
- Memory Corruption
- All Assembly and RE pwn college challenges
- I highly recommend you study memory corruption
    - 6.0/6.1
    - 7.0/7.1
    - 10.0/10.1
    - 12.0/12.1

Midterm: Resources for you

- John office hours Mon: 2-4
- Jonah office hours Wed: 1-3
- Mohamed office hours Fri:
- Thursday we will work together in class to clear up any issues
- Discord