Buffer Overflow Workshop

UC Davis Cybersecurity Club
Nate Buttke

April 19, 2022

Icebreaker

 Introduce yourself and describe the first computer that you can remember using

How does the computer execute programs?

- **Von Neumann architecture:** Memory and processor are separate. The processor has registers (and cache), though.
- The processor corresponds with the rest of the computer using the system bus, a group of parallel wires.
- sub-busses:
  - The **data bus** hat sends the contents of the memory between the CPU and RAM.
  - The **address bus** lets the CPU ask for a specific address in RAM
- One register, the instruction pointer (IP) contains the memory address of the current instruction

Registers (pdf in the tarball)

Intel Cheat Sheet

Registers

- Each of the following registers is 32 bits in size

Name	Meaning of Acronym	Usage
EAX	Accumulator	General Purpose Register
EBX	Base	General Purpose Register
ECX	Counter	General Purpose Register
EDX	Data	General Purpose Register
ESI	Source Index	General Purpose Register
EDI	Destination Index	General Purpose Register
EBP	Base Pointer	Used as the base pointer in C code. Can be used as a general register in stand alone code
ESP	Stack Pointer	The stack pointer.

- You can access smaller portions of EAX, EBX, ECX, and EDX
  - The following example only shows the syntax for eax but it does apply to the the above mentioned registers

Syntax	Size
EAX	32 bits
AX	Lower 16 bits of EAX
AH	The upper 8 bits of AX
AL	The lower 8 bits of AX

On Memory Addressing

- Have you ever used a 32 bit computer? Why can they only use 4 gigabytes of memory?
- Memory is byte-addressable. The smallest piece of memory you can ask for is a byte. (You can operate on bits, though.)
- Called "32 bits" because the registers are that size. This is also called the "word size"
- With 32 bits, you can enumerate $2^{32} \approx 4.29 \times 10^9$ bytes of memory (4.29 gigabytes).

Data representation

- Everything in the computer is represented as binary.
- *Everything:* `ints` `floats`, pointers, machine instructions

Endianness[1]

- Words are not ordered in the same way in every computer
- Little endian: least significant byte has the lowest address
- Big endian: Most significant byte has the lowest address

[1] Wikipedia: "The adjective endian comes from the 1726 novel Gulliver's Travels by Jonathan Swift where characters known as Lilliputians are divided into those breaking the shell of a boiled egg from the big end (Big-Endians) or from the little end (Little-Endians)."

More on data representation

- From *Below C Level* by Norm Matloff, a program's variables are contiguous in memory. Local variables should be in reverse order to their appearance in the C program, and global variables in normal order.

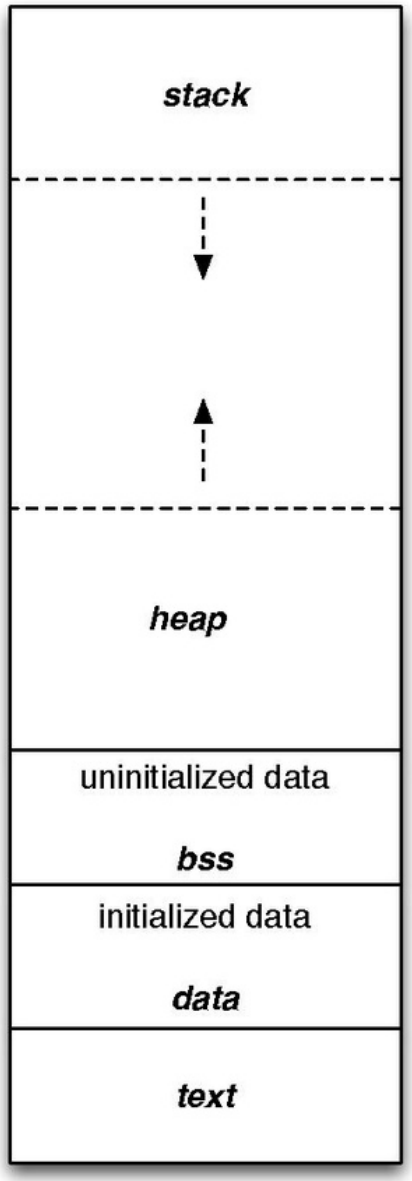
Not on my machine? Take this with a grain of salt

```
(gdb) print (X + 0)
$1 = (int *) 0x7fffffffe860
(gdb) print (Y + 0)
$2 = (int *) 0x7fffffffe880
(gdb)
```

```
int main(void){
  int X[5],Y[20], I;
```

Overflow: introductory example

- Structs are more reliable for placing variables next to each other in memory.
- Look at `stack0.c`. Can you figure out how to change `changeme`? What makes the difference between inputs that change or do not change the variable?



A closer look

- It should take one char (byte) over 64 to write to `changeme`
- How can we see this in action? That is, how can we debug with only a binary?
- In gdb we can use `disassemble main` to view the corresponding machine instrucitons

```
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x0000555555555159 <+0>:     push   rbp
   0x000055555555515a <+1>:     mov    rbp,rsp
   0x000055555555515d <+4>:     sub    rsp,0x60
   0x0000555555555161 <+8>:     mov    DWORD PTR [rbp-0x54],edi
   0x0000555555555164 <+11>:    mov    QWORD PTR [rbp-0x60],rsi
   0x0000555555555168 <+15>:    mov    rax,QWORD PTR fs:0x28
   0x0000555555555171 <+24>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555175 <+28>:    xor    eax,eax
   0x0000555555555177 <+30>:    mov    DWORD PTR [rbp-0x10],0x0
   0x000055555555517e <+37>:    lea    rax,[rbp-0x50]
   0x0000555555555182 <+41>:    mov    rdi,rax
   0x0000555555555185 <+44>:    call   0x555555555040 <gets@plt>
   0x000055555555518a <+49>:    mov    eax,DWORD PTR [rbp-0x10]
   0x000055555555518d <+52>:    test   eax,eax
   0x000055555555518f <+54>:    je     0x5555555551a2 <main+73>
   0x0000555555555191 <+56>:    lea    rax,[rip+0xe70]        # 0x555555556
   0x0000555555555198 <+63>:    mov    rdi,rax
   0x000055555555519b <+66>:    call   0x555555555030 <puts@plt>
   0x00005555555551a0 <+71>:    jmp    0x5555555551b1 <main+88>
   0x00005555555551a2 <+73>:    lea    rax,[rip+0xe97]        # 0x555555556
   0x00005555555551a9 <+80>:    mov    rdi,rax
   0x00005555555551ac <+83>:    call   0x555555555030 <puts@plt>
   0x00005555555551b1 <+88>:    mov    edi,0x0
   0x00005555555551b6 <+93>:    call   0x555555555050 <exit@plt>
End of assembler dump.
(gdb)
```

assembly operations

Operations

- In operations that require two operands the left is called the source and the right the destination

Operation	Result	Affects Flag Register?
mov src, dest	dest = src	No
add src, dest	dest = dest + src	Yes
sub src, dest	dest = dest - src	Yes
cmp src, dest	dest – src (result not stored)	Yes
inc dest	dest = dest + 1	Yes
dec dest	dest = dest - 1	Yes
and src, dest	dest = dest & src	Yes
or src, dest	dest = dest | src	Yes
xor src, dest	Dest = dest ^ src	Yes
not dest	dest = ~dest	Yes
shr src, dest	dest = dest >> src (brings in 0s)	Yes
sar src, dest	dest = dest >> src (brings in bit with same value as MSB	Yes
shl src, dest	dest = dest << src	Yes

Watching the stack with gdb

- We can set breakpoints by copying and pasting the hex for instructions
- To print stack data: `x/<number><width><format> <address>`, e.g. `x/24wx $esp`.
- `x/10i $rip` prints 10 instructions after IP
- `define hook-stop` to set commands that will run after each pause (end your input with `end`).

Watching the stack with gdb

Cool! we are getting somewhat comfortable with binary programs in gdb Can we use what we learned in `stack0.c` to complete `stack1.c`?

- `strcpy` is insecure like `gets`.
- *Fun fact*: `git` bans some C functions, including `strcpy` from its source.[2]

Watch out for endianness

[2] `https://github.com/git/git/blob/master/banned.h`

epic. I bet you can figure out `stack2.c`, too. And maybe if we have time we can try to insert a function call into the stack.