Operating Systems and C
Fall 2022, Security-Track
6. Stack-Based Exploits

Parallel Tracks

exciting part of the course!

perflab:

- have two matrix multiplication procedures (rotate, smooth), have to rewrite it.
- about optimization techniques; blocking, loop unrolling, etc.

performance-track lecture nr. 1:
what you'll need for perflab.
array layout, what it means for performance, cache hierarchy, how associativity is organized in the cache. (important stuff for anyone)

performance-track lecture nr. 2:
what you'll need for perflab. optimizations. how to write code so compiler can derive performant code. manual transformations, blocking, loop unrolling

attacklab:

- you have an executable, have to attack it.
- about the stack; code injection (smashing the stack), return-oriented programming (find interesting code in other programs).

security-track lecture nr. 1:
what you'll need for attacklab.

Outline

basically
what you need
for the **attacklab**

(next week: not
important for attacklab
(culture-stuff))

Brief review of assembly

Arrays, strings and structs in assembly

alignment; needed in
assignment

Structure of an .s program

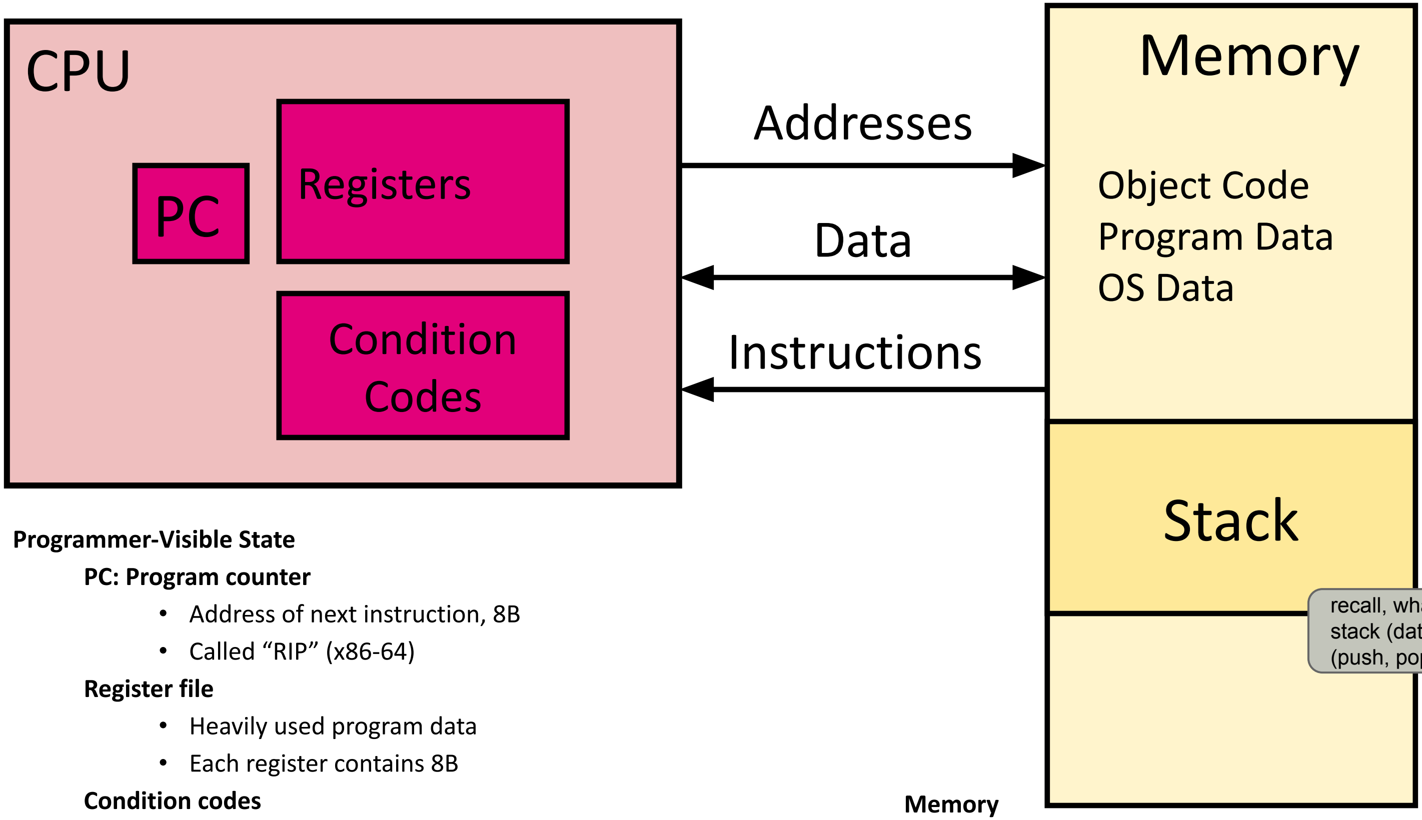
you'll need to
disassemble a binary

Stack-based exploits and counter-measures

X86-64 Assembly

von Neumann Architecture

instruction either
- op on registers (state), or
- transfers data to/from mem



recall, what is a stack (data structure)? (push, pop)

Programmer-Visible State

PC: Program counter
- Address of next instruction, 8B
- Called "RIP" (x86-64)

Register file
- Heavily used program data
- Each register contains 8B

Condition codes
- Store status information about most recent arithmetic operation
- Used for conditional branching

Memory
- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Registers

10s

aka. **p**rogram **c**ounter (**pc**)

The **%rip** register is the current **i**nstruction **p**ointer.
Contains address of next instruction to be executed.

most instructions implicitly increment it.
explicitly updated ⇒ change in control flow.

There are **16 general purpose registers** in x86-64.
Additional registers for floating point, SIMD, …
16 registers: r0, r1, …, r15

“register file”

General-Purpose Registers

For historical reasons, r0-r7 are called **original registers**.
They have the following names:

- ax: register a
- bx: register b
- cx: register c
- dx: register d

top & bottom
for a given frame

- bp: register **b**ase **p**ointer (start of stack)
- sp: register **s**tack **p**ointer (current location in stack, grow downwards)

- si: register source index (source for data copies)
- di: register destination index (destination for data copies)

usually first
parameters of
functions
(if not arrays)

General-Purpose Registers

memory is byte-addressable

why: e.g. C short is 2B

Register values can be accessed at different levels of granularity:

- **8B**:
  - original registers: **prefix r** rax, rsp, rsi
  - other registers: no suffix r8, r15
- **4B**:
  - original registers: **prefix e** eax, esp, esi
  - other registers: **suffix d** r8d, r15d
- **2B:**
  - original registers: **no prefix** ax, sp, si
  - other registers: **suffix w** r8w, r15w
- **1B** (high byte):
  - original registers (bits 8-15 from ax-dx) ah, bh, ch, dh
- **1B** (low byte):
  - original registers (bits 0-7 from ax-dx) al, bl, cl, dl
  - other registers: suffix b r8b, r15b

Instructions

Three classes of instructions:

1. Transfer between memory and register
   - Load/store data: register <-> memory (e.g. mov)
   - Push/pop: register <-> stack

we didn't talk much about these, except we walked about how these are built from logic gates

2. Arithmetic and comparison functions

3. Transfer control
   - Jumps to/from procedures
   - Conditional branches

can loop

can if/then

AT&T syntax

10s

The GNU tools (gcc, gdb) use AT&T Syntax for assembly.

example: `movq %rsp, %rbp`

Syntax is of the form

OPERATOR source, destination

never more than 2 operands.
when there are 2, this is the form.

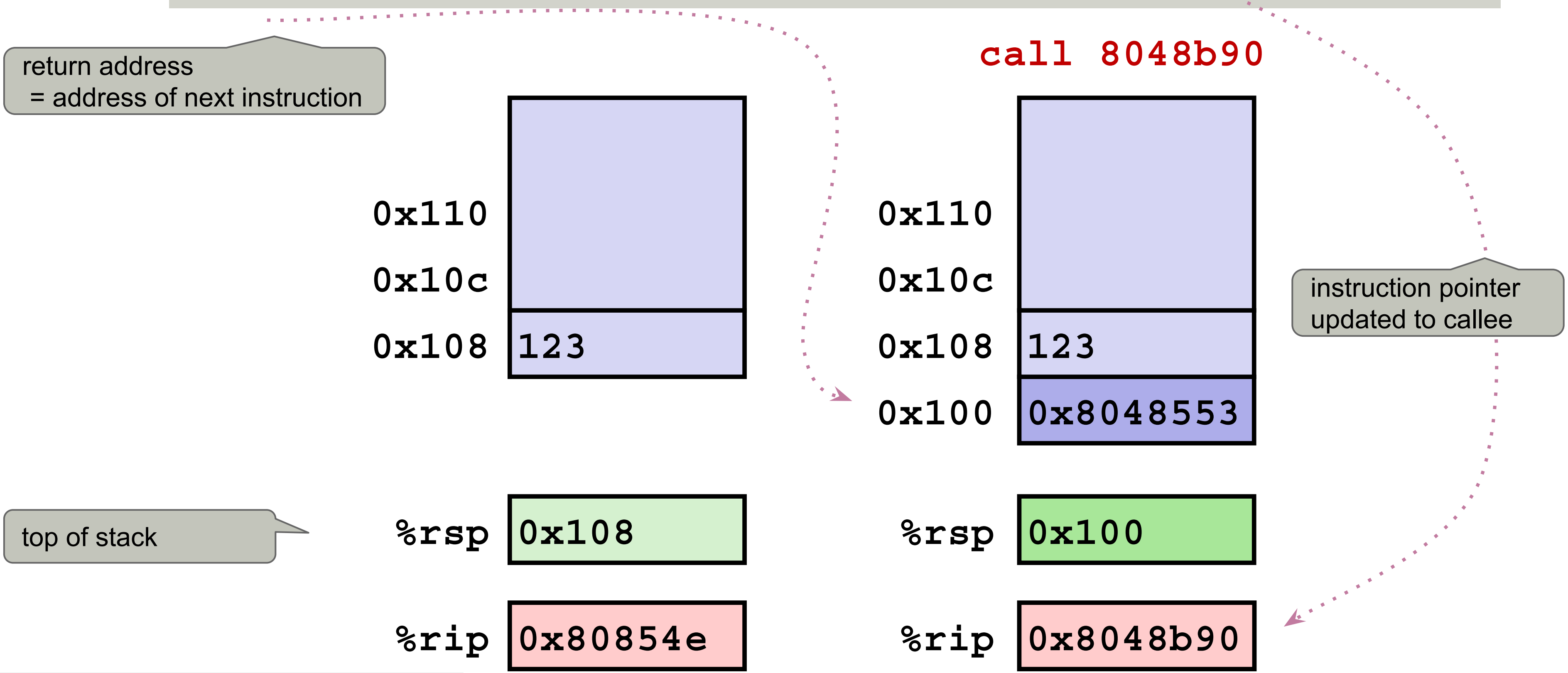
Register names are prefixed with **%**

The **alternative** is the **Intel syntax** (on windows): MOVQ EBP, ESP – no %

Look for % in the assembly code, if they are present you are dealing with AT&T syntax

Procedure Call Example

```
804854e:  e8 3d 06 00 00    call   8048b90 <main>
8048553:  50                pushl  %eax
```



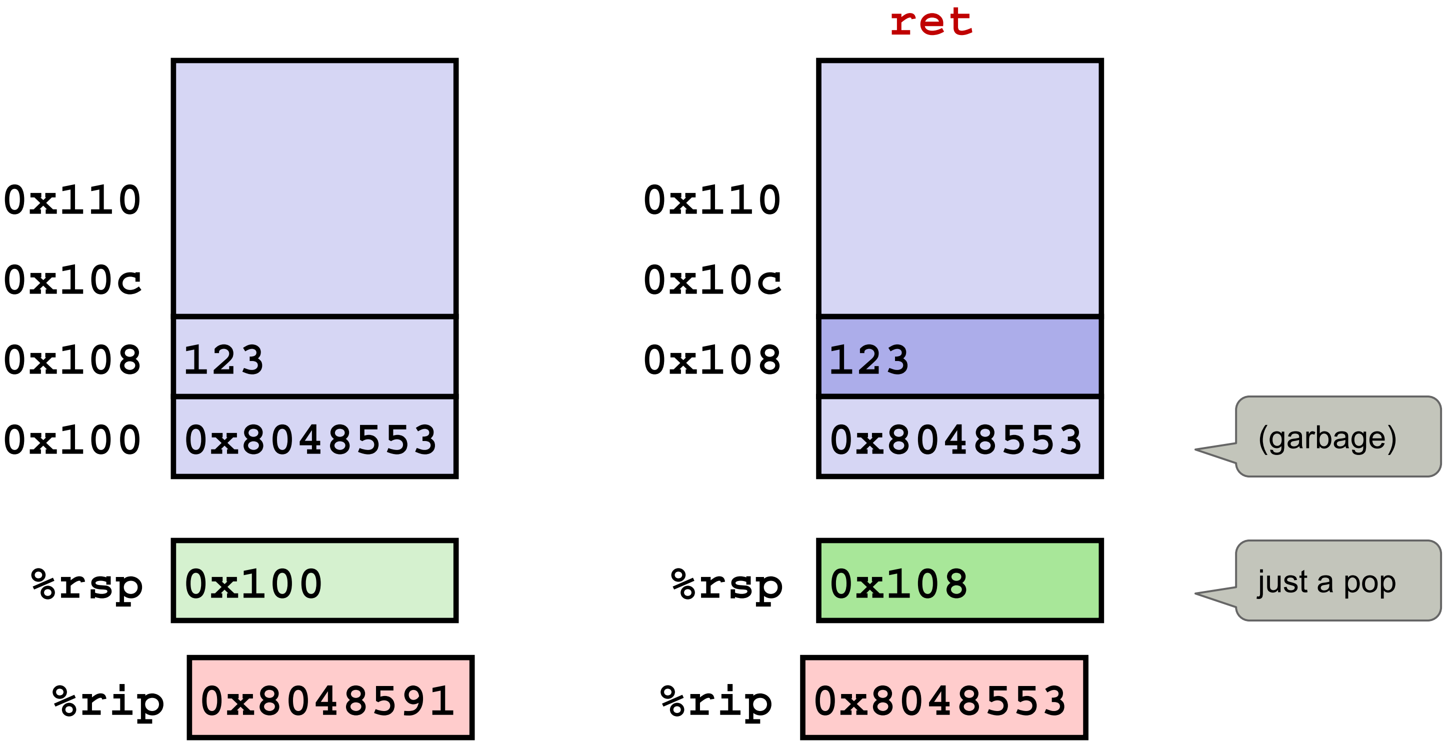
return address
= address of next instruction

top of stack

instruction pointer
updated to callee

Procedure Call Example

```
8048591:  c3                ret
```



Q: what happens if user input overwrites stack pointer, or return address?
A: user has full control over control flow. (exploit)

Stack Frame

stack frame

- caller's saved registers(1)
- local vars
- args
- return address(2)

(1) : to restore caller state on return
(bottom frame has no caller).
(2) : pushes it when it calls
(top frame does not need this)

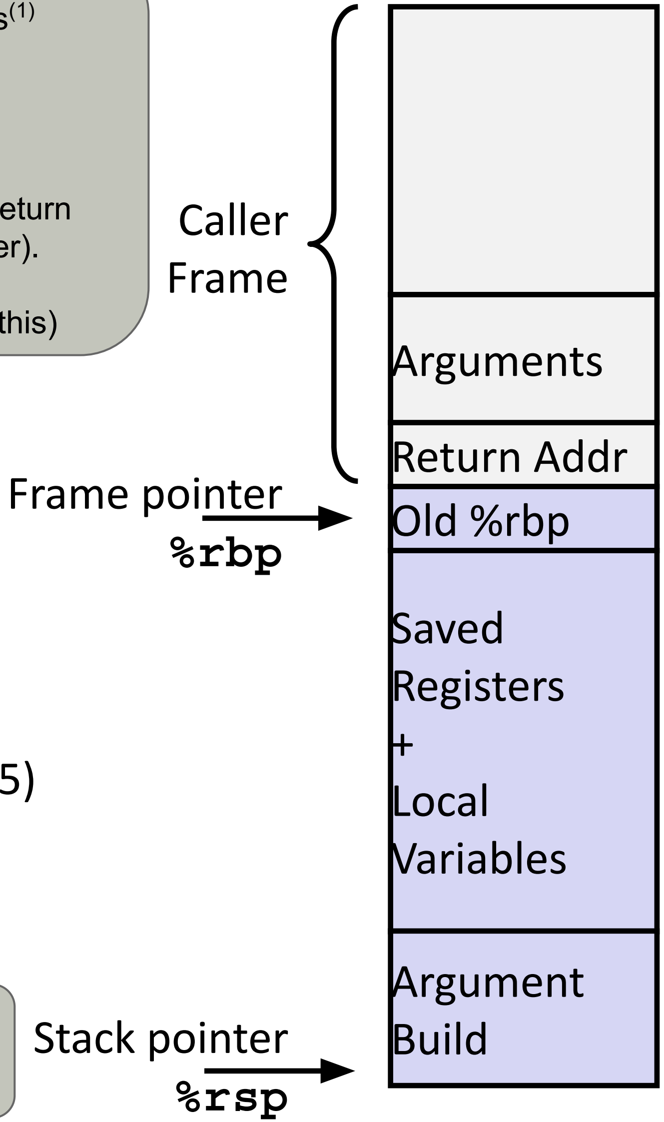
Caller:

- Arguments
  - pushed by program (if needed)
- Return address
  - pushed by call

Callee:

- Previous frame pointer (%rbp)
- Other callee-save registers (%rbx, %r12-15)
- Space for local variables
- Arguments for next function
(when about to call another function)

extremely important for assignment; you'll spend hours seeing what stack looks like.



Caller Frame

Arguments

Return Addr

Frame pointer `%rbp`

Old %rbp

Saved Registers + Local Variables

Argument Build

Stack pointer `%rsp`

Outline

Brief review of assembly

Arrays, strings and structs in assembly

Structure of an .s program

Stack-based exploits and counter-measures

Array Allocation

compiler is going to reserve space for the array.

Basic Principle

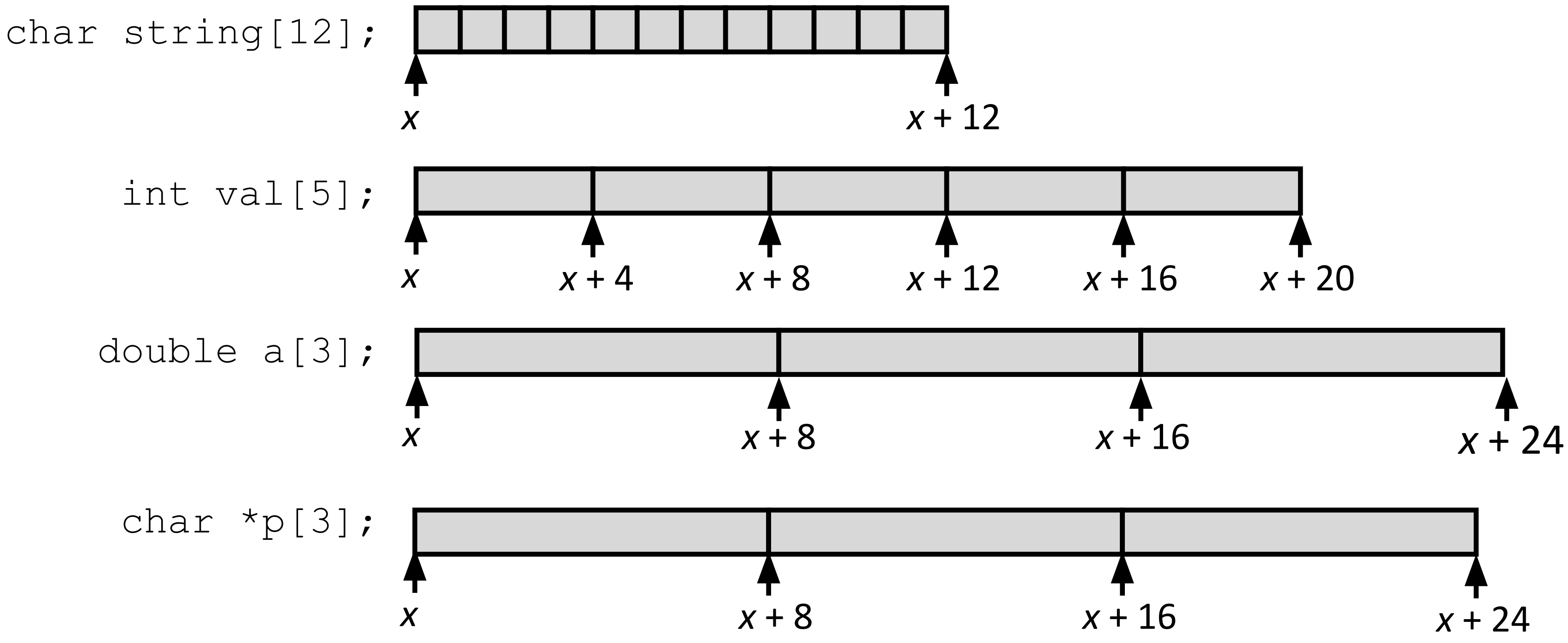
T **`A[`***L***`]`***;*

A is an Array of data type *T* and length *L*

Contiguously allocated region of *L* * **`sizeof`**（*T*） bytes

examples follow

Array Allocation



`char string[12];`

`int val[5];`

`double a[3];`

`char *p[3];`

8 bytes = 64 bits
(address size =
word size)

Array Access

int **A[*5*] = {0, 1, 2, 3, 4};**

Array of data type *int* and length *5*

Identifier **A** can be used as a pointer to array element 0: Type *int**

val from
previous slide

Reference	Type	Value
val[4]	**int**	**4**
val	**int ***	x
val+1	**int ***	$x + 4$
&val[2]	**int ***	$x + 8$
val[5]	**int**	??
***(val+1)**	**int**	**1**
val + ***i***	**int ***	$x + 4\,i$

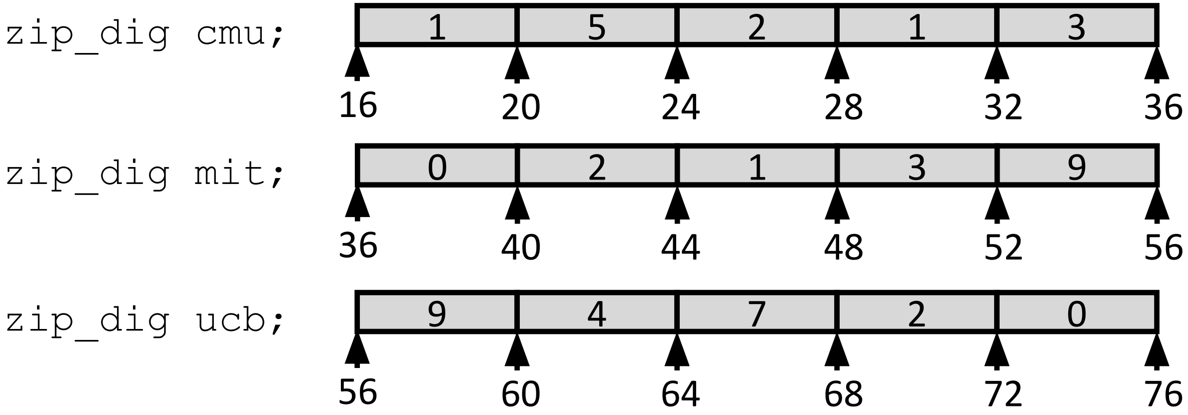
undefined
behavior

Array Example

type alias

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

(ucb = berkeley)



20 = 5*4

- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Example

```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

addressing mechanism:
z + 4*i

```
  # rdx = z
    movq     $0, %rax                 #    %rax = i
.L4:                                  # loop:
    addq     $1, (%rdx,%rax,4)        #    z[i]++
    addq     $1, %rax                 #    i++
    cmpl     $5, %rax                 #    i:ZLEN (ZLEN==5)
    jne .L4                           #    if !=, goto loop
```

$1 is
constant 1

%r is
register r

Strings

Declared as read-only data.

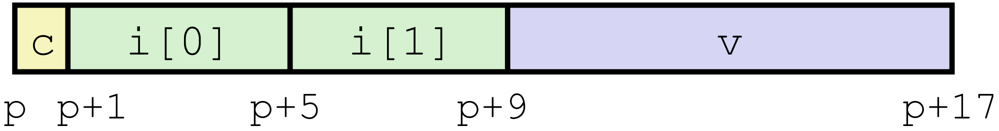
```
    .section      .rodata
.Label:
    .string    ”String constant\n”
```

1 byte per character, terminated by \0

we won’t spend more time on strings, because you won’t need them in the assignment.

Structures & Alignment

Unaligned Data



not a power of 2.
moving data in memory,
and to/from register,
is a mess. instead:

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

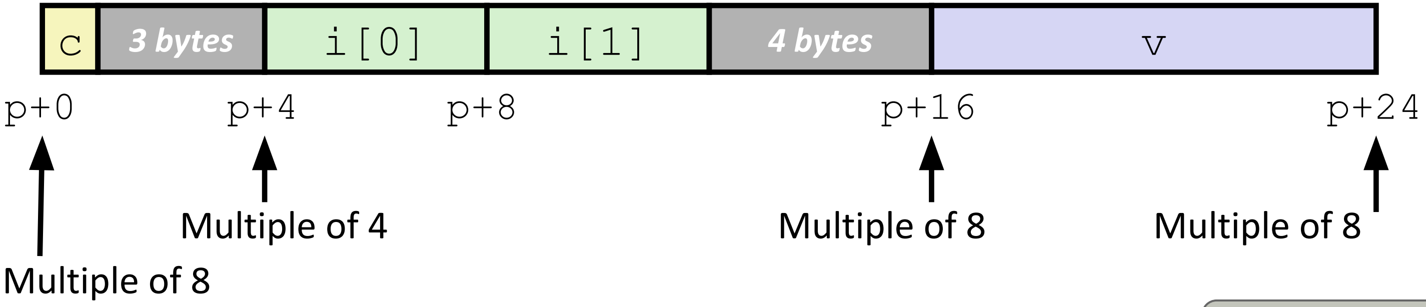
Aligned Data

Primitive data type requires *K* bytes ⇒

Its address must be multiple of *K*

starting address must be multiple of K (1, 2, 4, 8, 16, …)

whole data structure is padded up to multiple of its largest K



gcc stack is 16-byte aligned. remember that.

Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …

  no restrictions on address
- 2 bytes: **`short`**, …

  lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …

  lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, **`char *`**, …

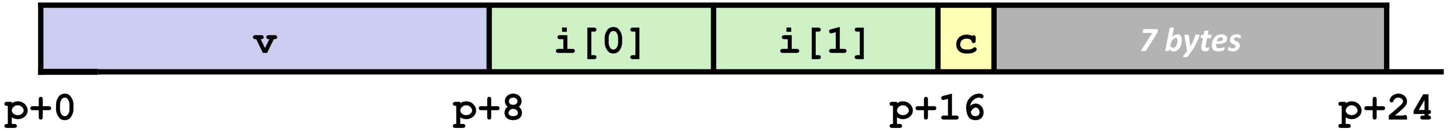
  lowest 3 bits of address must be 000_2

Q: but Willard, that's wasteful.
A: when we talk dynamic memory allocation (heap), we'll see these bits used for something else (tags).

Meeting Overall Alignment Requirement

For largest alignment requirement K
Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



order in descending order of size makes it easier for compiler to store your struct more compactly.

Outline

Brief review (x86-64 + ex 3.67)

Arrays, strings and structs in assembly

Structure of an .s program

what transformation of my code does my compiler do? that's why we want to be able to read assembly.

Stack-based exploits and counter-measures

Virtual Memory

you have
different **sections**
in your assembly program,
which corresponds to
different regions
in virtual memory.

4 sections

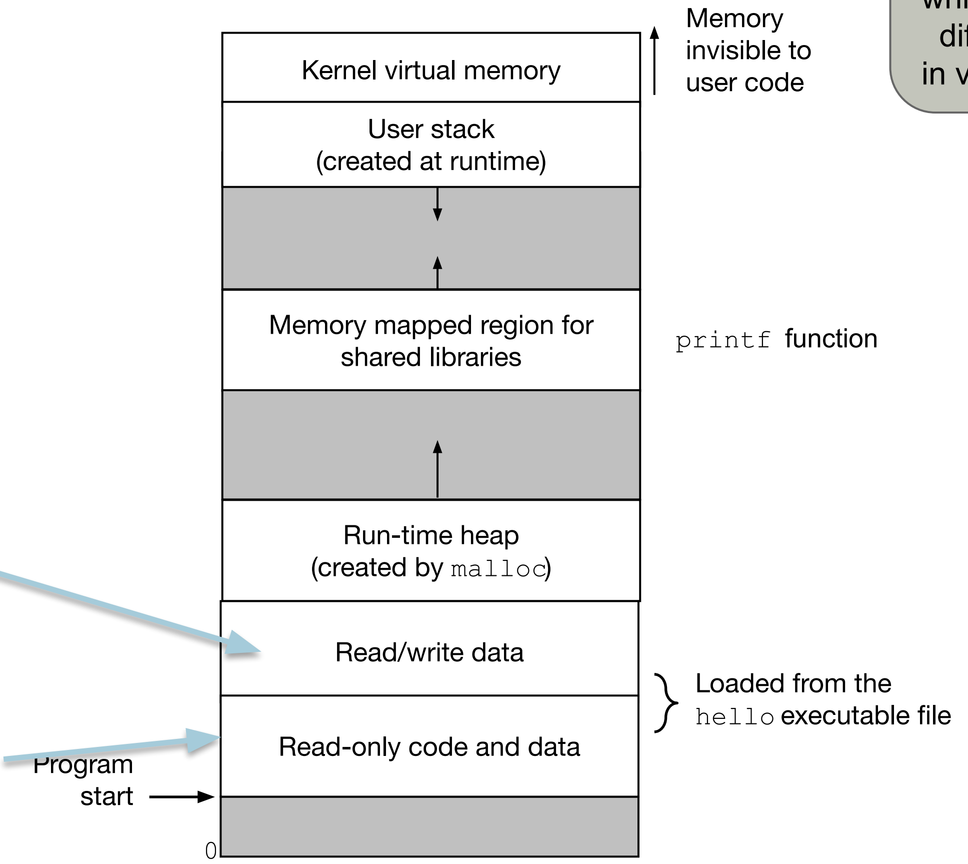
defined, but not initialized

(uninitialised) .bss:

(initialised read-write) .data:

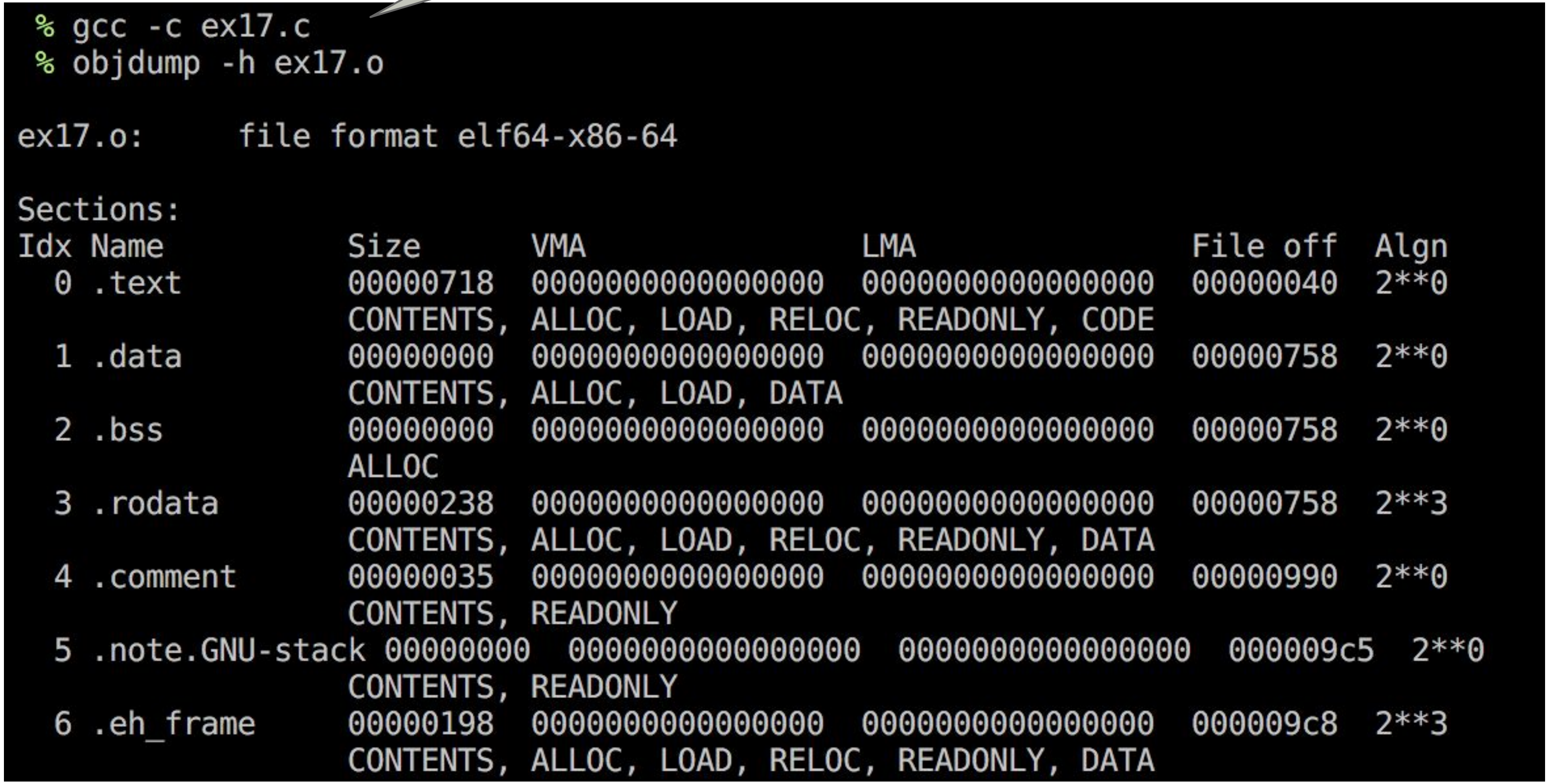
(initialised read-only) .rodata:

(program) .text:



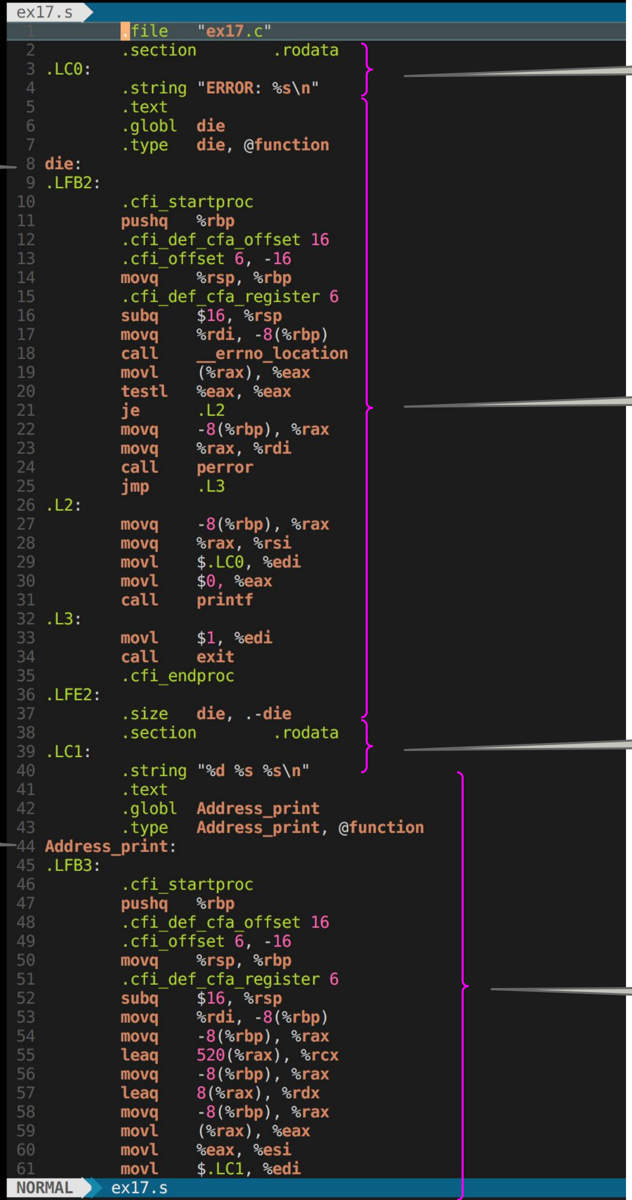
Object file

-h (headers) reveals structure of object file (the sections, their size, etc.).



```
 % gcc -c ex17.c
 % objdump -h ex17.o

ex17.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000718  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000758  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000758  2**0
                  ALLOC
  3 .rodata       00000238  0000000000000000  0000000000000000  00000758  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .comment      00000035  0000000000000000  0000000000000000  00000990  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000009c5  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000198  0000000000000000  0000000000000000  000009c8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

Assembly file



```
        .file   "ex17.c"
        .section        .rodata
.LC0:
        .string "ERROR: %s\n"
        .text
        .globl  die
        .type   die, @function
die:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    %rdi, -8(%rbp)
        call    __errno_location
        movl    (%rax), %eax
        testl   %eax, %eax
        je      .L2
        movq    -8(%rbp), %rax
        movq    %rax, %rdi
        call    perror
        jmp     .L3
.L2:
        movq    -8(%rbp), %rax
        movq    %rax, %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
.L3:
        movl    $1, %edi
        call    exit
        .cfi_endproc
.LFE2:
        .size   die, .-die
        .section        .rodata
.LC1:
        .string "%d %s %s\n"
        .text
        .globl  Address_print
        .type   Address_print, @function
Address_print:
.LFB3:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        leaq    520(%rax), %rcx
        movq    -8(%rbp), %rax
        leaq    8(%rax), %rdx
        movq    -8(%rbp), %rax
        movl    (%rax), %eax
        movl    %eax, %esi
        movl    $.LC1, %edi
```

read-only data

a function

program

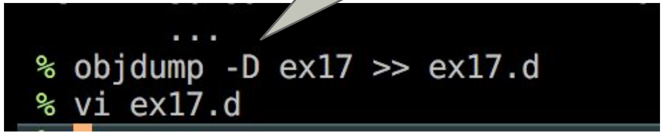
read-only data

a function

program

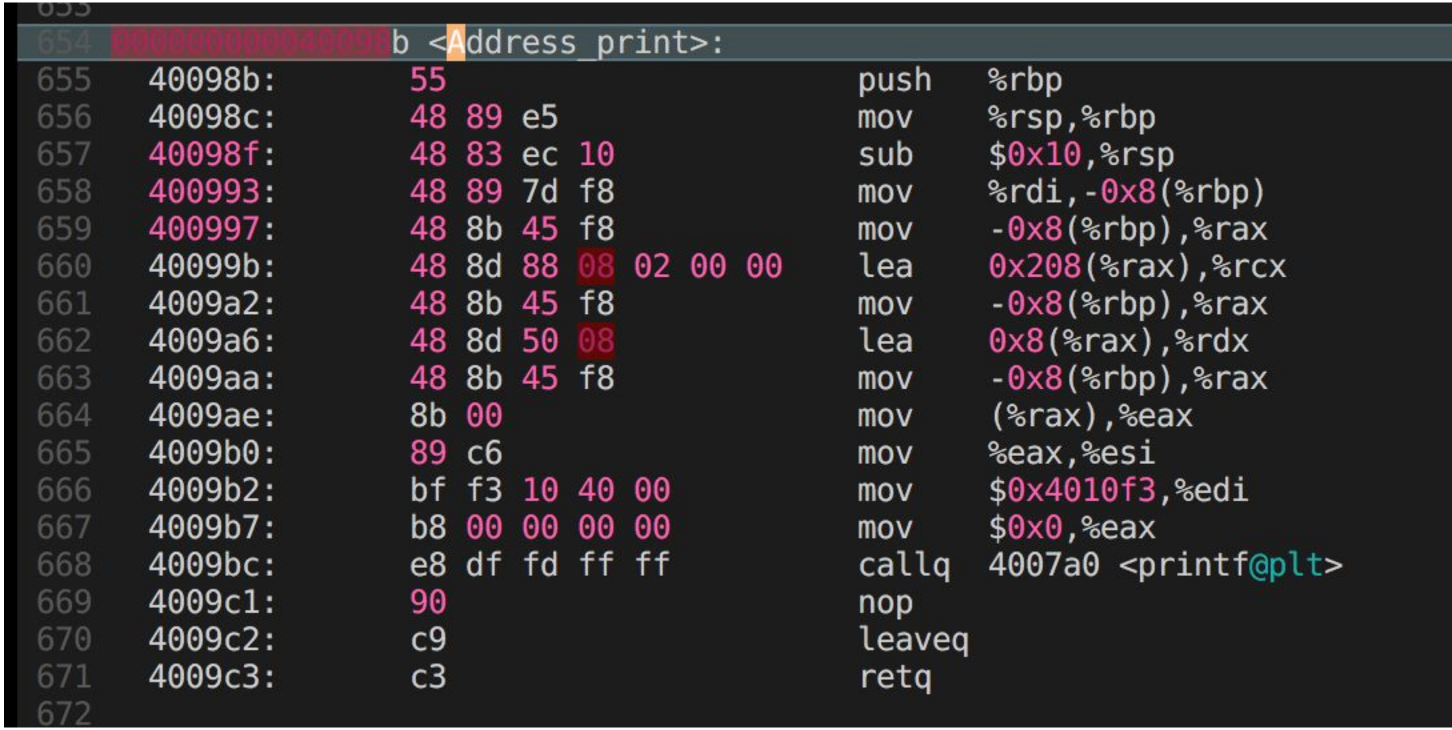
Object file

disassemble all sections,
not just those containing instructions



```
   ...
% objdump -D ex17 >> ex17.d
% vi ex17.d
```

reveals the actual
address of each instruction



```
000000000040098b <Address_print>:
  40098b:       55                      push   %rbp
  40098c:       48 89 e5                mov    %rsp,%rbp
  40098f:       48 83 ec 10             sub    $0x10,%rsp
  400993:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
  400997:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  40099b:       48 8d 88 08 02 00 00    lea    0x208(%rax),%rcx
  4009a2:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  4009a6:       48 8d 50 08             lea    0x8(%rax),%rdx
  4009aa:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  4009ae:       8b 00                   mov    (%rax),%eax
  4009b0:       89 c6                   mov    %eax,%esi
  4009b2:       bf f3 10 40 00          mov    $0x4010f3,%edi
  4009b7:       b8 00 00 00 00          mov    $0x0,%eax
  4009bc:       e8 df fd ff ff          callq  4007a0 <printf@plt>
  4009c1:       90                      nop
  4009c2:       c9                      leaveq
  4009c3:       c3                      retq
```

last 2 phases of
assignment: need
to use this to find
patterns in the code

Outline

Brief review (x86-64 + ex 3.67)

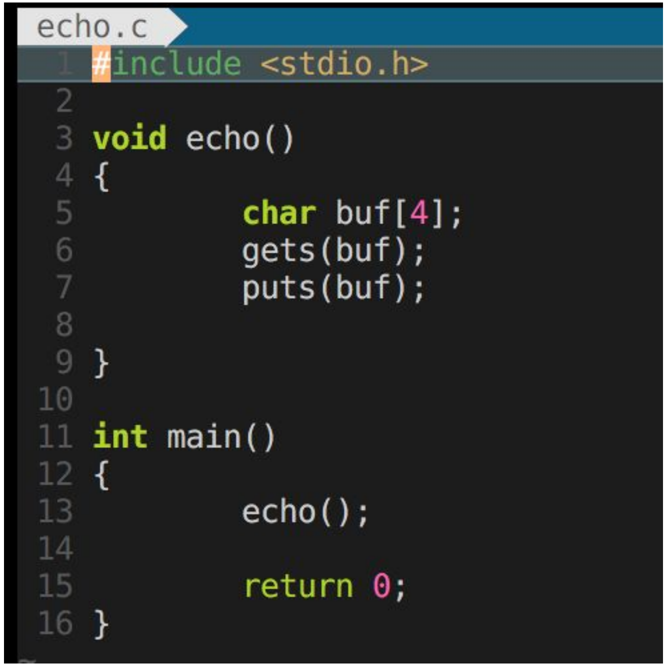
Arrays, strings and structs in assembly

Structure of an .s program

Stack-based exploits and counter-measures

Vulnerable C code

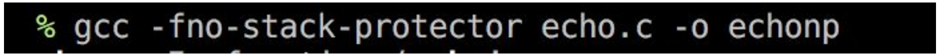
example of vulnerable C code



echo.c

```c
#include <stdio.h>

void echo()
{
        char buf[4];
        gets(buf);
        puts(buf);

}

int main()
{
        echo();

        return 0;
}
```

disable stack protection
(more on that later today)
(so it’s not really a problem
today; compilers prevent
problem by default)



```
% gcc -fno-stack-protector echo.c -o echonp
```

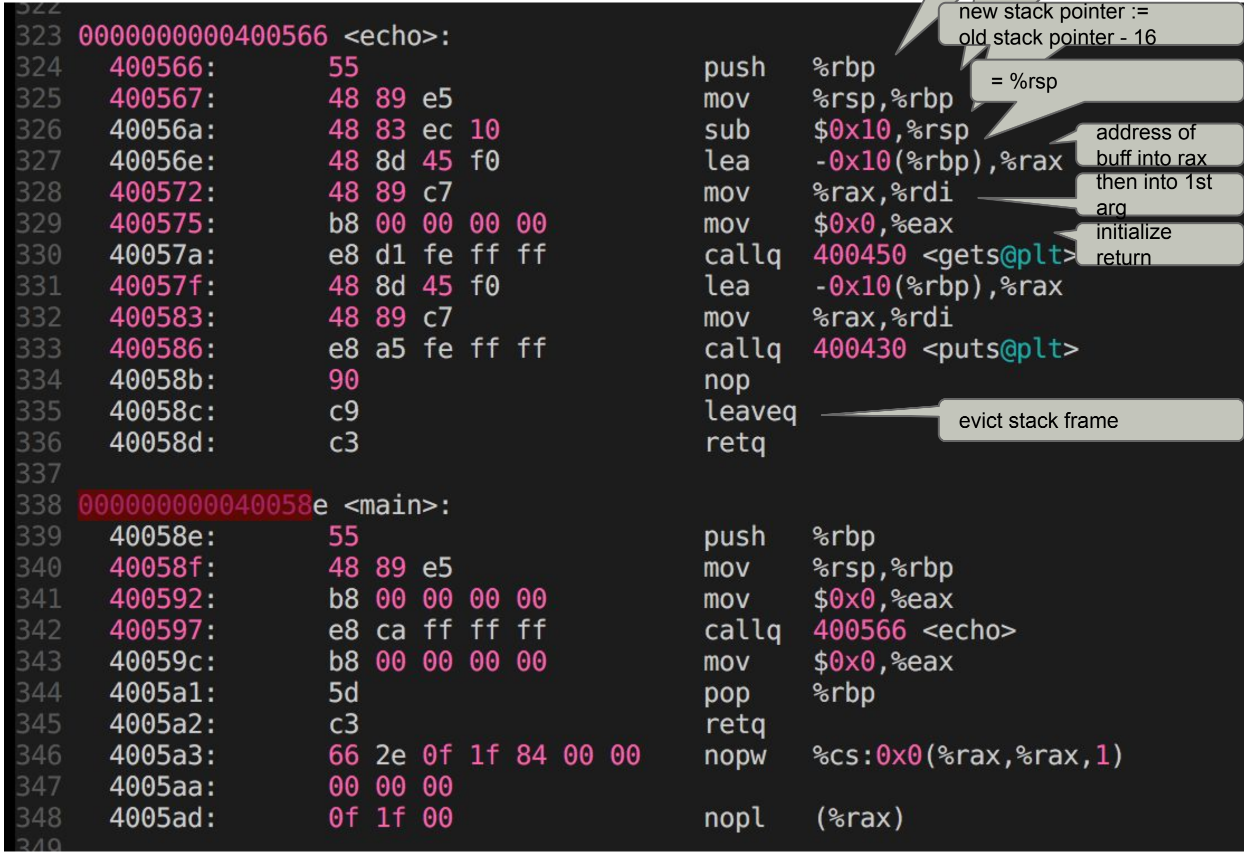
```
 % ./echonp
12345678
12345678
 % ./echonp
123456789
123456789
 % ./echonp
12345678901234567890123
12345678901234567890123
 % ./echonp
123456789012345678901234567890123
123456789012345678901234567890123
[1]    19108 segmentation fault (core dumped)  ./echonp
```

process tried accessing
something outside itself.
what’s happening?
let’s investigate.

Vulnerable C code

tip: for assignment, always use objdump. with it, you get assembly, and the object, and the address where it is

```
% objdump -D echonp >> echonp.d
```



```
0000000000400566 <echo>:
  400566:       55                      push   %rbp
  400567:       48 89 e5                mov    %rsp,%rbp
  40056a:       48 83 ec 10             sub    $0x10,%rsp
  40056e:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400572:       48 89 c7                mov    %rax,%rdi
  400575:       b8 00 00 00 00          mov    $0x0,%eax
  40057a:       e8 d1 fe ff ff          callq  400450 <gets@plt>
  40057f:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400583:       48 89 c7                mov    %rax,%rdi
  400586:       e8 a5 fe ff ff          callq  400430 <puts@plt>
  40058b:       90                      nop
  40058c:       c9                      leaveq
  40058d:       c3                      retq

000000000040058e <main>:
  40058e:       55                      push   %rbp
  40058f:       48 89 e5                mov    %rsp,%rbp
  400592:       b8 00 00 00 00          mov    $0x0,%eax
  400597:       e8 ca ff ff ff          callq  400566 <echo>
  40059c:       b8 00 00 00 00          mov    $0x0,%eax
  4005a1:       5d                      pop    %rbp
  4005a2:       c3                      retq
  4005a3:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4005aa:       00 00 00
  4005ad:       0f 1f 00                nopl   (%rax)
```

push old base pointer on stack

new base pointer := old stack pointer

new stack pointer := old stack pointer - 16

= %rsp

address of buff into rax then into 1st arg

initialize return

evict stack frame

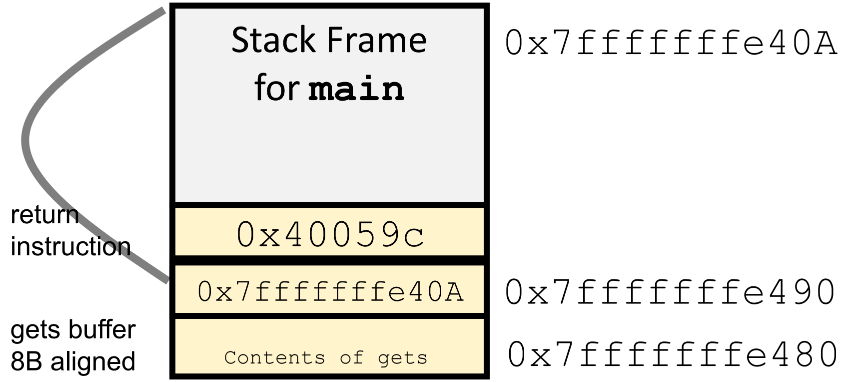
The stack is 16B aligned
=>
rsp is decreased with 16B
sub $0x10, %rsp

enough space for buf

Vulnerable code

so, why vulnerable?

Before call to gets



Stack Frame for **`main`**

0x7fffffffe40A

return instruction

0x40059c

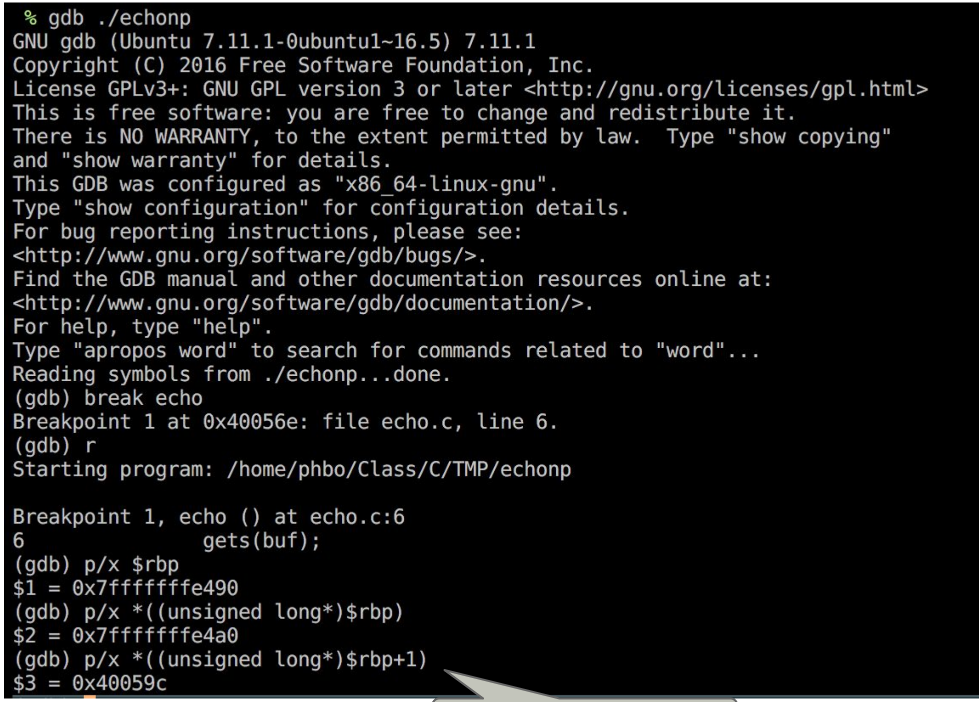
0x7fffffffe40A

0x7fffffffe490

gets buffer 8B aligned

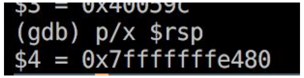
Contents of gets

0x7fffffffe480



```
 % gdb ./echonp
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./echonp...done.
(gdb) break echo
Breakpoint 1 at 0x40056e: file echo.c, line 6.
(gdb) r
Starting program: /home/phbo/Class/C/TMP/echonp

Breakpoint 1, echo () at echo.c:6
6               gets(buf);
(gdb) p/x $rbp
$1 = 0x7fffffffe490
(gdb) p/x *((unsigned long*)$rbp)
$2 = 0x7fffffffe4a0
(gdb) p/x *((unsigned long*)$rbp+1)
$3 = 0x40059c
```

%rbp+1 is return instruction



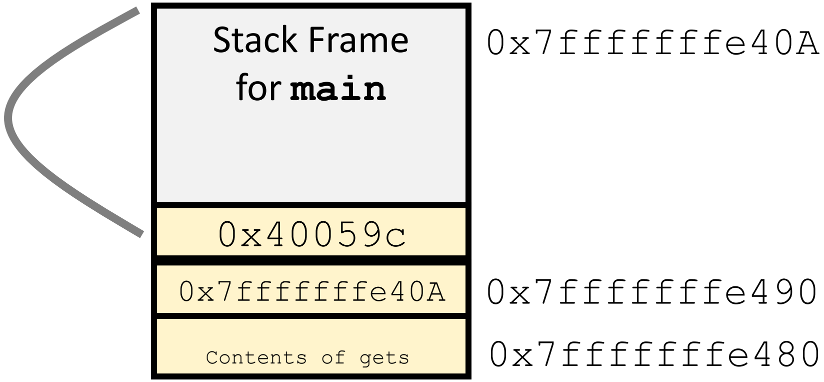
```
$3 = 0x40059c
(gdb) p/x $rsp
$4 = 0x7fffffffe480
```

if I write more than 16 bytes,
then I overwrite return address.
I take over the machine then
(or provoke segmentation fault)

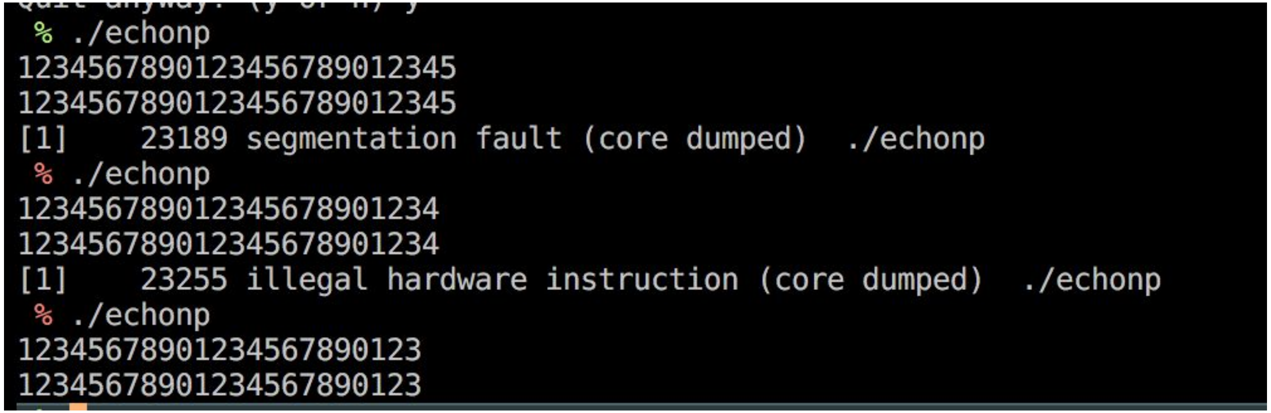
```
  400597:       e8 ca ff ff ff          callq  400566 <echo>
  40059c:       b8 00 00 00 00          mov    $0x0,%eax
```

Buffer Overflow Stack Example

Before call to gets

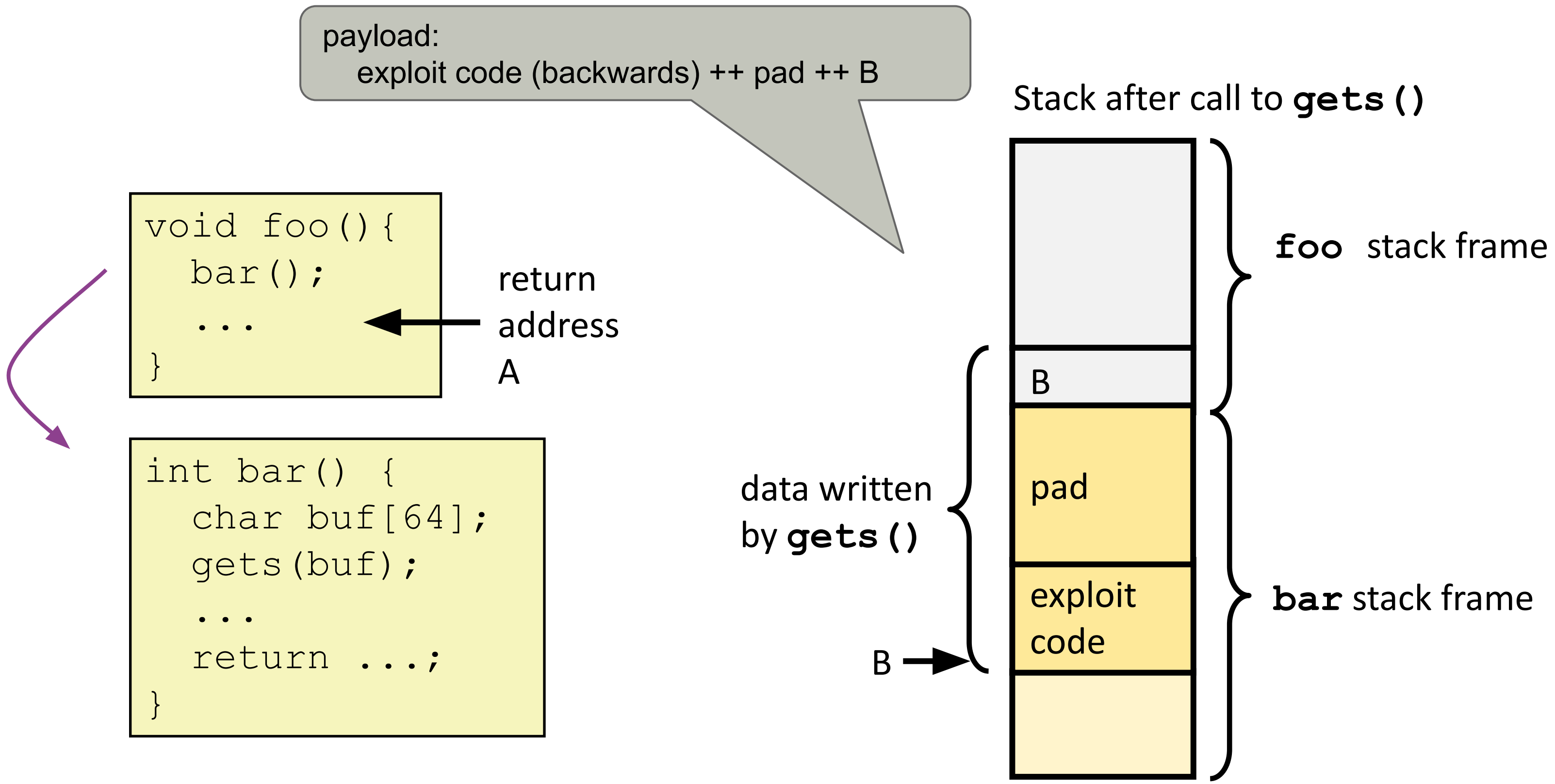


depending on input,
different termination behavior.



```
Quit anyway? (y or n) y
 % ./echonp
1234567890123456789012345
1234567890123456789012345
[1]    23189 segmentation fault (core dumped)  ./echonp
 % ./echonp
123456789012345678901234
123456789012345678901234
[1]    23255 illegal hardware instruction (core dumped)  ./echonp
 % ./echonp
12345678901234567890123
12345678901234567890123
```

Malicious Use of Buffer Overflow



payload:
exploit code (backwards) ++ pad ++ B

```
void foo(){
  bar();
  ...
}
```

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Input string contains byte representation of executable code
Overwrite return address with address of exploit code
When **`bar()`** executes **`ret`**, will jump to exploit code

Exploits Based on Buffer Overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

Internet worm

Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:

- **`finger droh@cs.cmu.edu`**

Worm attacked fingerd server by sending phony argument:

- **`finger "exploit-code padding new-return-address"`**
- exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Morris worm, 1988

how to stop this:
- want to make it impossible to execute code on the stack.
- want to make it hard to figure out where exploit code starts.
- want to protect return address.

Avoiding Overflow Vulnerability

use a function that checks how much you read.

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

Use library routines that limit string lengths

- **`fgets`** instead of **`gets`**
- **`strncpy`** instead of **`strcpy`**
- Don't use **`scanf`** with **`%s`** conversion specification
  - Use **`fgets`** to read the string
  - Or use **`%ns`** where **`n`** is a suitable integer

System-Level Protections

Randomized stack offsets

At start of program, allocate random amount of space on stack

Makes it difficult for hacker to predict beginning of inserted code

Nonexecutable code segments

In traditional x86, can mark region of memory as either “read-only” or “writeable”

- Can execute anything readable

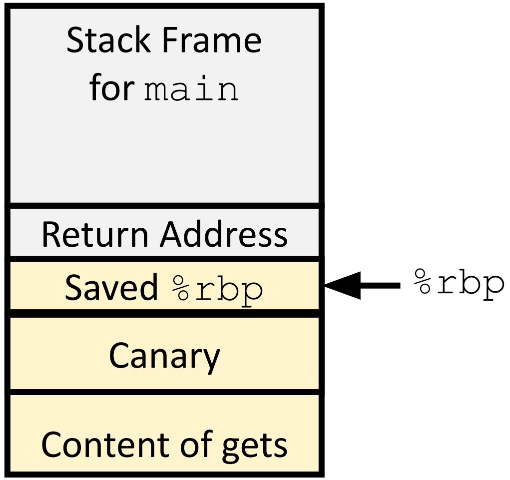
X86-64 added explicit “execute” permission

Stack Canaries

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**
  - **`-fstack-protector-all (default)`**

Setting Up Canary

Before call to gets



Stack Frame
for `main`

Return Address

Saved `%rbp`

Canary

Content of gets

`%rbp`

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov  %fs:20, %rax       # Get canary
    mov  %rax, -8(%rbp)     # Put on stack
    xor  %eax, %eax         # Erase canary
    . . .
```

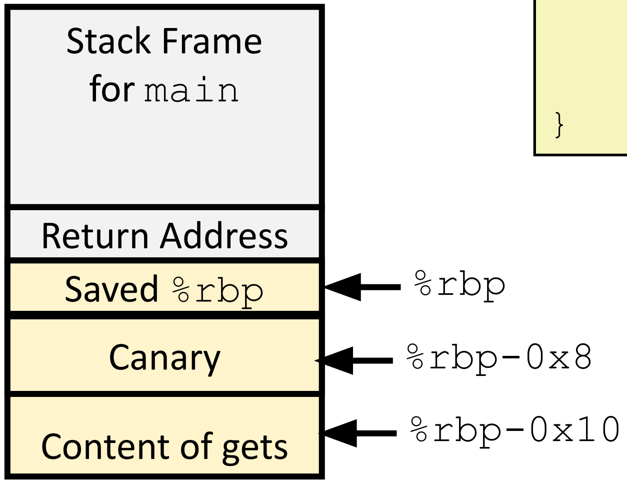
random part of memory into rax

put canary just below base pointer

must overwrite canary to overwrite return address

Checking Canary

Before call to gets



Stack Frame for `main`

Return Address

Saved `%rbp`

Canary

Content of gets

`%rbp`

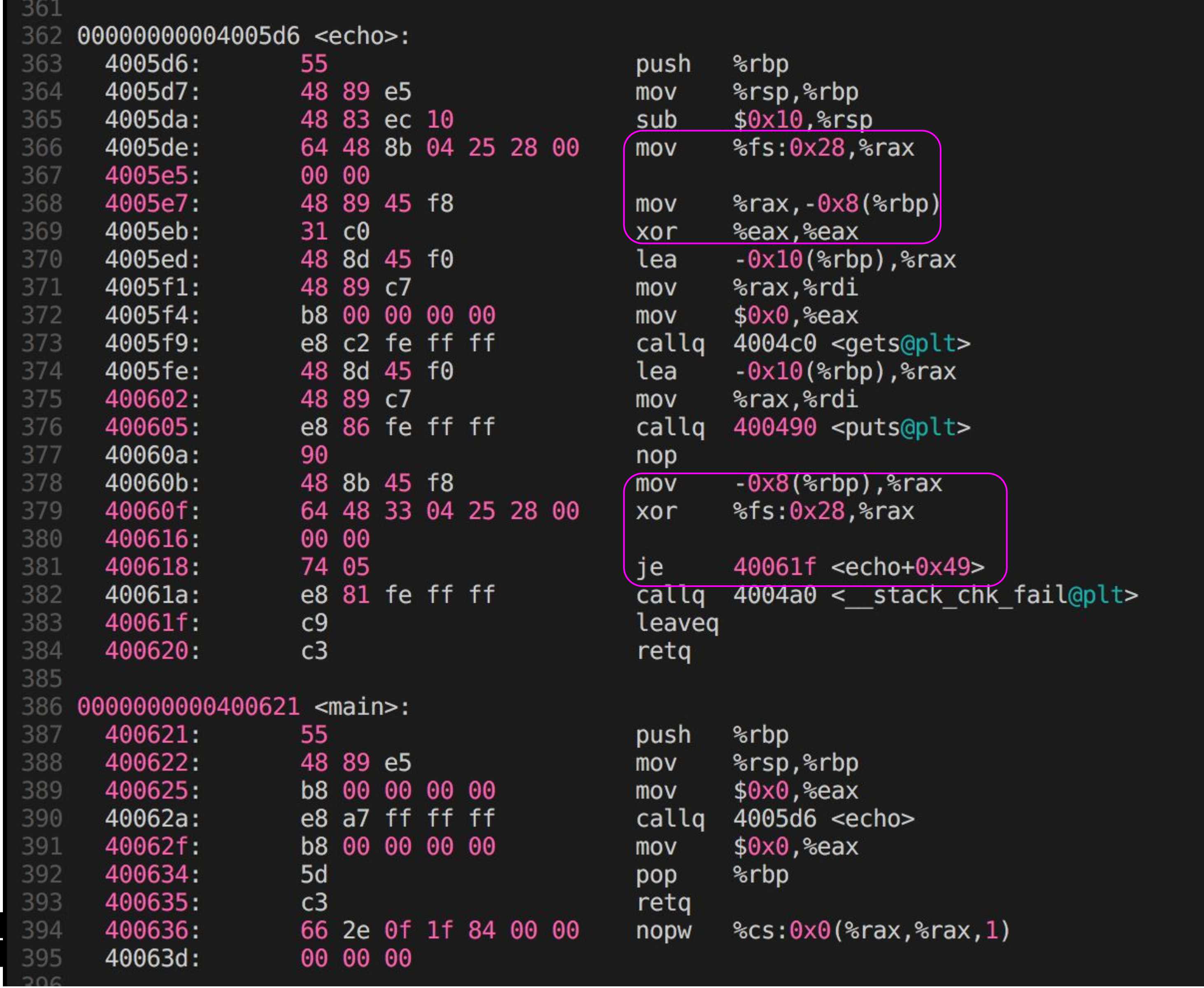
`%rbp-0x8`

`%rbp-0x10`

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

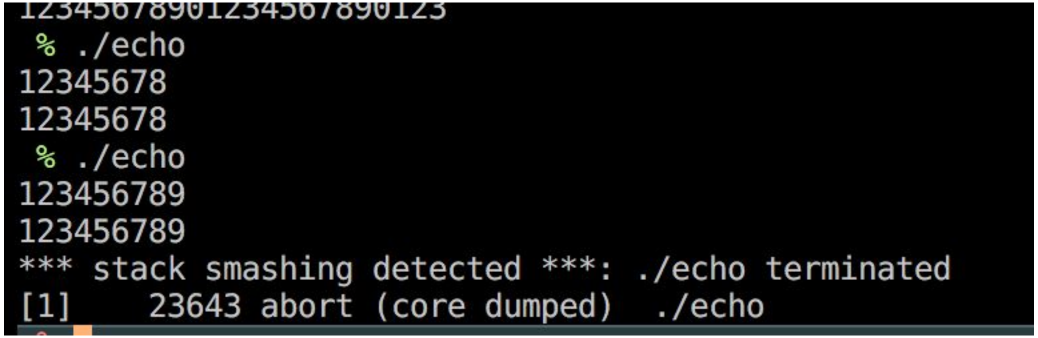
```
echo:
    . . .
    mov  -8(%rbp), %rax     # Retrieve from
stack
    xor  %fs:20, %rax       # Compare with
Canary
    je   .L24               # Same: skip ahead
    call __stack_chk_fail   # ERROR
.L24:
    . . .
```

Canary Example



```
00000000004005d6 <echo>:
  4005d6:       55                      push   %rbp
  4005d7:       48 89 e5                mov    %rsp,%rbp
  4005da:       48 83 ec 10             sub    $0x10,%rsp
  4005de:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4005e5:       00 00
  4005e7:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4005eb:       31 c0                   xor    %eax,%eax
  4005ed:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  4005f1:       48 89 c7                mov    %rax,%rdi
  4005f4:       b8 00 00 00 00          mov    $0x0,%eax
  4005f9:       e8 c2 fe ff ff          callq  4004c0 <gets@plt>
  4005fe:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400602:       48 89 c7                mov    %rax,%rdi
  400605:       e8 86 fe ff ff          callq  400490 <puts@plt>
  40060a:       90                      nop
  40060b:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  40060f:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  400616:       00 00
  400618:       74 05                   je     40061f <echo+0x49>
  40061a:       e8 81 fe ff ff          callq  4004a0 <__stack_chk_fail@plt>
  40061f:       c9                      leaveq
  400620:       c3                      retq

0000000000400621 <main>:
  400621:       55                      push   %rbp
  400622:       48 89 e5                mov    %rsp,%rbp
  400625:       b8 00 00 00 00          mov    $0x0,%eax
  40062a:       e8 a7 ff ff ff          callq  4005d6 <echo>
  40062f:       b8 00 00 00 00          mov    $0x0,%eax
  400634:       5d                      pop    %rbp
  400635:       c3                      retq
  400636:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40063d:       00 00 00
```

Canary Example

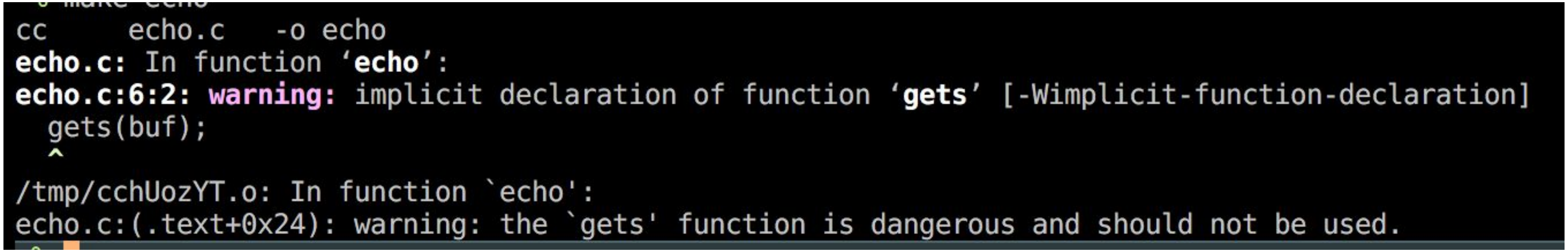


```
12345678901234567890123
 % ./echo
12345678
12345678
 % ./echo
123456789
123456789
*** stack smashing detected ***: ./echo terminated
[1]    23643 abort (core dumped)  ./echo
```

when compiled w/ protection

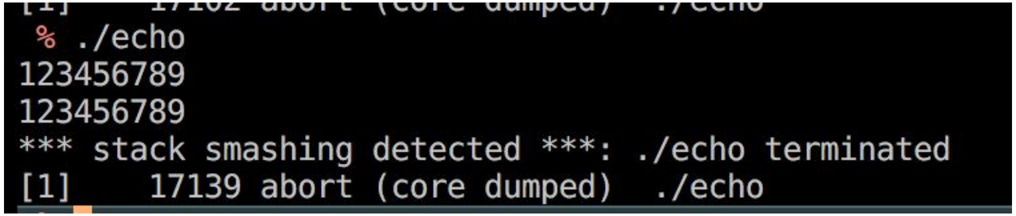
GCC protections

Compile-time



```
cc      echo.c   -o echo
echo.c: In function ‘echo’:
echo.c:6:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
  gets(buf);
  ^
/tmp/cchUozYT.o: In function `echo':
echo.c:(.text+0x24): warning: the `gets' function is dangerous and should not be used.
```

Run-time



```
 % ./echo
123456789
123456789
*** stack smashing detected ***: ./echo terminated
[1]    17139 abort (core dumped)  ./echo
```

pretty clear that you shouldn’t use **gets**.

shameless self-promotion

goal: tools that developers can use to write secure SW.

sample research (past supervisions):

- analyze binaries for information leaks
- reduce timing leaks in the Linux kernel
- automatically fix vulnerabilities in JavaScript
- automatically generate (i.e. synthesize) a secure program from formal specification
- assess privacy risk in analytics programs (data scientists; Google search for “Privugger”)

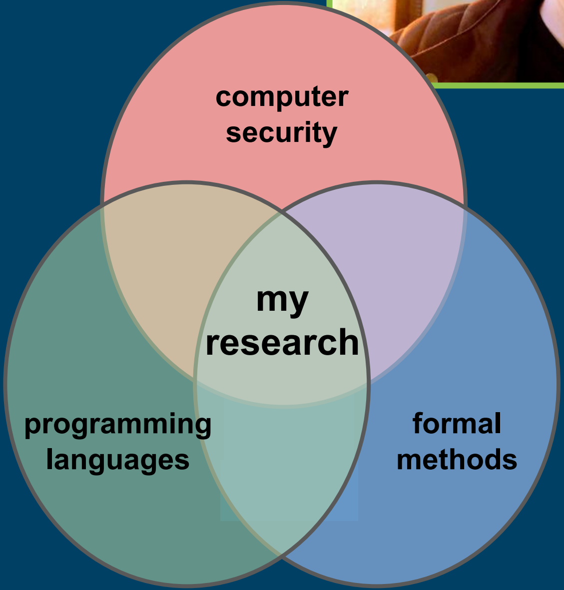
I like code, and I like proofs.

I created the “Applied Information Security” course.

I’m a barista in Analog.

Willard Rafnsson
IT University of Copenhagen

wilr@itu.dk
https://www.willardthor.com/



Take-Aways

Alignment is necessary when working with structs

Object files are sequences of hex codes, that can be mapped to assembly instructions

Buffer overflows cause vulnerabilities that can be exploited maliciously

Counter measures include (i) randomized stack addresses, (ii) non executable code on stack and (iii) canaries.