Agenda

Assembly instructions

- Arithmetic
- Example: Calling a function
- Lea

Demo: The guessing game

- Dissassembly
- Buffer exploits

Assembly – Common instructions

Table 1. Most Common Instructions

Instruction	Translation
`mov S, D`	S → D (i.e, copies value of S into D)
`add S, D`	S + D → D (adds S to D and stores result in D)
`sub S, D`	D - S → D (subtracts S *from* D and stores result in D)

Example – Common instructions

```
mov -0x4(%rbp),%eax
add $0x2,%eax
```

Assembly: Arithmetic

Table 1. Common Arithmetic Instructions.

Instruction	Translation
`add S, D`	S + D → D
`sub S, D`	D - S → D
`inc D`	D + 1 → D
`dec D`	D - 1 → D
`neg D`	-D → D
`imul S, D`	S × D → D
`idiv S`	`%rax` / S: quotient → `%rax`, remainder → `%rdx`

Table 2. Bit Shift Instructions

Instruction	Translation	Arithmetic or Logical?
`sal v, D`	D `<<` v → D	arithmetic
`shl v, D`	D `<<` v → D	logical
`sar v, D`	D `>>` v → D	arithmetic
`shr v, D`	D `>>` v → D	logical

Table 3. Bitwise Operations

Instruction	Translation
`and S, D`	S `&` D → D
`or S, D`	S `|` D → D
`xor S, D`	S `^` D → D
`not D`	`~`D → D

Recall: Program Memory

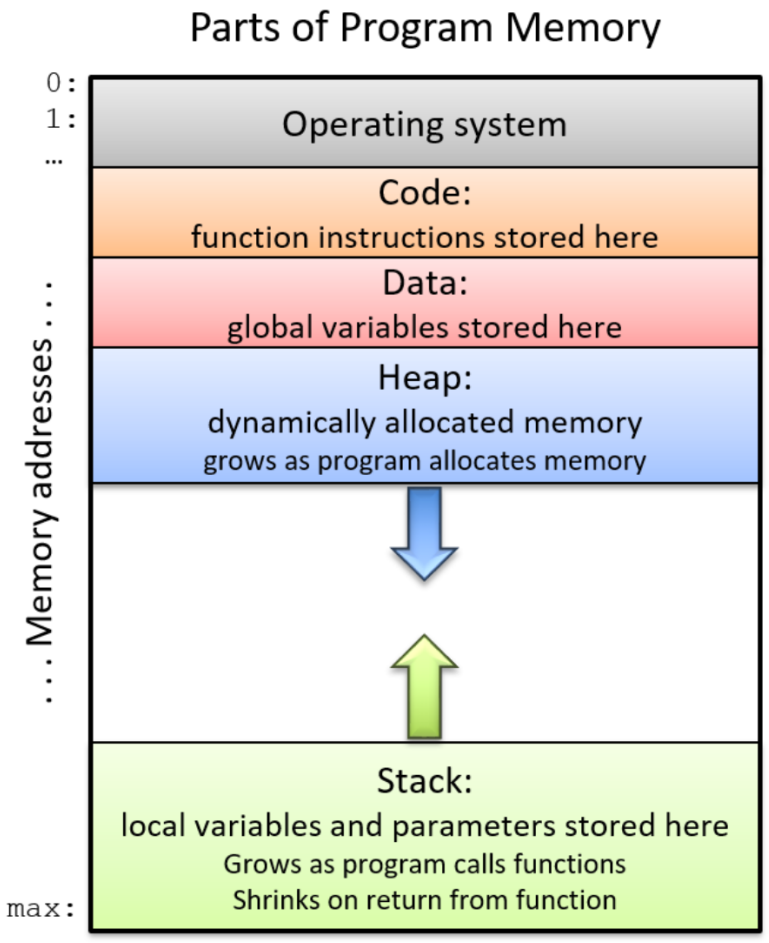


Figure 1. The parts of a program's address space

Table 2. Stack Management Instructions

Instruction	Translation

`push S`	Pushes a copy of S onto the top of the stack. Equivalent to:
	`sub $0x8, %rsp`
	`mov S, (%rsp)`

`pop D`	Pops the top element off the stack and places it in location D. Equivalent to:
	`mov (%rsp), D`
	`add $0x8, %rsp`

Recall: The Execution Stack

```
#include <stdio.h>

int adder2(int a) {
  return a + 2;
}

int main(){
  int x = 40;
  x = adder2(x);
  printf("x is: %d\n", x);
  return 0;
}
```

Assembly: Pop Quiz

What is %rsp?

What is %rbp?

Assembly: Functions

All currently executing functions are on the stack

- Only the topmost is running
- All other functions are waiting

Local variable are references with respect to the base pointer (%rbp)

When the stack is popped, nothing is cleaned!

- E.g. old values are still there

Assembly: Functions

Registers	
%rsp	0x40c
%rbp	0x418

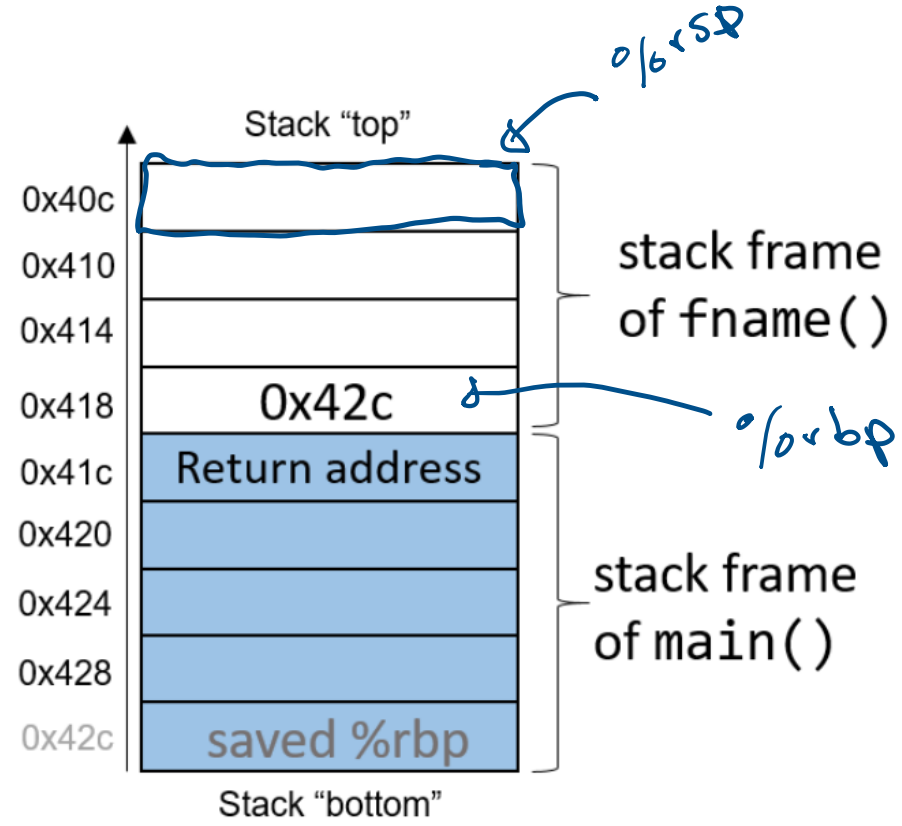


Figure 1. Stack frame management

Assembly: Function Stack Management

Table 2. Stack Management Instructions

Instruction	Translation

`push S`	Pushes a copy of S onto the top of the stack. Equivalent to:
	`sub $0x8, %rsp`
	`mov S, (%rsp)`

`pop D`	Pops the top element off the stack and places it in location D. Equivalent to:
	`mov (%rsp), D`
	`add $0x8, %rsp`

Assembly: Functions

Table 1. Common Function Management Instructions

Instruction	Translation

`leaveq`	Prepares the stack for leaving a function. Equivalent to:
	`mov %rbp, %rsp`
	`pop %rbp`

`callq addr <fname>`	Switches active frame to callee function. Equivalent to:
	`push %rip`
	`mov addr, %rip`

`retq`	Restores active frame to caller function. Equivalent to: `pop %rip`

Table 2. Locations of Function Parameters.

Parameter	Location
Parameter 1	%rdi
Parameter 2	%rsi
Parameter 3	%rdx
Parameter 4	%rcx
Parameter 5	%r8
Parameter 6	%r9
Parameter 7+	on call stack

Example – adder2

```
int adder2(int a) {
   return a + 2;
}
```

```
0000000000400526 <adder2>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       89 7d fc                mov    %edi,-0x4(%rbp)
  40052d:       8b 45 fc                mov    -0x4(%rbp),%eax
  400530:       83 c0 02                add    $0x2,%eax
  400533:       5d                      pop    %rbp
  400534:       c3                      retq
```

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	Junk
%edi	0x28
%rsp	0xd28
%rbp	0xd40
%rip	0x526

Stack

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Stack

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Stack

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Stack

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Stack

Address	Value
0xd28	Return Address
0xd30	

Example – adder2

```
0x526   push   %rbp
0x527   mov    %rsp, %rbp
0x52a   mov    %edi, -0x4(%rbp)
0x52d   mov    -0x4(%rbp), %eax
0x530   add    $0x2, %eax
0x533   pop    %rbp
0x534   retq
```

Registers	
%eax	
%edi	
%rsp	
%rbp	
%rip	

Stack

Address	Value
0xd28	Return Address
0xd30	

Assembly: lea

Load effective address (lea) is a fast instruction for computing memory addresses

Unlike *mov*, *lea* does not perform a memory lookup

Example: lea

Registers	
%rax	0x5
%rdx	0x4
%rcx	0x808

Instruction	Translation	Value
lea 8(%rax), %rax		
lea (%rax, %rdx), %rax		
lea (,%rax,4), %rax		
lea -0x8(%rcx), %rax		
lea -0x4(%rcx, %rdx, 2), %rax		

Real World Exploits: Buffer Overflow

Buffer is another term for array.

Buffer overflow occurs when we try to access memory outside of the bounds of an array.

Usually this results in a segmentation fault; however, buffer overflows can be exploited by hackers to make a program execute in a manner different from what was intended. This is called a **buffer overflow exploit.**

Example: Guessing Game

```
void endGame(void){
  printf("You win!\n");
  exit(0);
}

int main(){
  int guess, secret, len, x=3;
  char buf[12];
  printf("Enter secret number:\n");
  scanf("%s", buf);
  guess = atoi(buf);
  secret=getSecretCode();
  if (guess == secret)
    printf("You got it right!\n");
  else{
    printf("You are so wrong!\n");
    return 1;
  }
  printf("Enter the secret string to win:\n");
  scanf("%s", buf);
  guess = calculateValue(buf, strlen(buf));
  if (guess != secret){
    printf("You lose!\n");
    return 2;
  }
  endGame();
  return 0;
}
```

Suppose an attacker has access to the executable as well as the source code here.

How can they execute the code in endgame without knowing the secret code and secret value?

Reverse Engineering

Using tools such as GDB and hexedit, one can **reverse engineer** an executable to infer the original code.

From this, you could find the secret number and secret message if they were hard-coded.

gdb ./secret

disass main

Dump of assembler code for function main:

```
   0x0000000004006f2 <+0>:     push   %rbp
   0x0000000004006f3 <+1>:     mov    %rsp,%rbp
   0x0000000004006f6 <+4>:     sub    $0x20,%rsp
    … etc
```

Exploiting buffer vulnerabilities

```
void endGame(void){
  printf("You win!\n");
  exit(0);
}

int main(){
  int guess, secret, len, x=3;
  char buf[12];
  printf("Enter secret number:\n");
  scanf("%s", buf);
  guess = atoi(buf);
  secret=getSecretCode();
  if (guess == secret)
    printf("You got it right!\n");
  else{
    printf("You are so wrong!\n");
    return 1;
  }
  printf("Enter the secret string to win:\n");
  scanf("%s", buf);
  guess = calculateValue(buf, strlen(buf));
  if (guess != secret){
    printf("You lose!\n");
    return 2;
  }
  endGame();
  return 0;
}
```

Where are the potential lines where we could overrun a buffer?

Exploiting buffer vulnerabilities

```
main:
  <+0>: push   %rbp
  <+1>: mov    %rsp,%rbp
  <+4>: sub    $0x20,%rsp
  <+8>: movl   $0x3,-0x4(%rbp)
  <+15>:mov    $0x400873,%edi
  <+20>:callq  0x400500<printf@plt>
  <+25>:lea    -0x20(%rbp),%rax
  <+29>:mov    %rax,%rsi
  <+32>:movl   $0x400888,%edi
  <+37>:mov    $0x0,%eax
  <+42>:callq  0x400540<scanf@plt>
  <+47>:lea    -0x20(%rbp),%rax
  <+51>:mov    %rax,%rdi
  <+54>:callq  0x400530<atoi@plt>
```

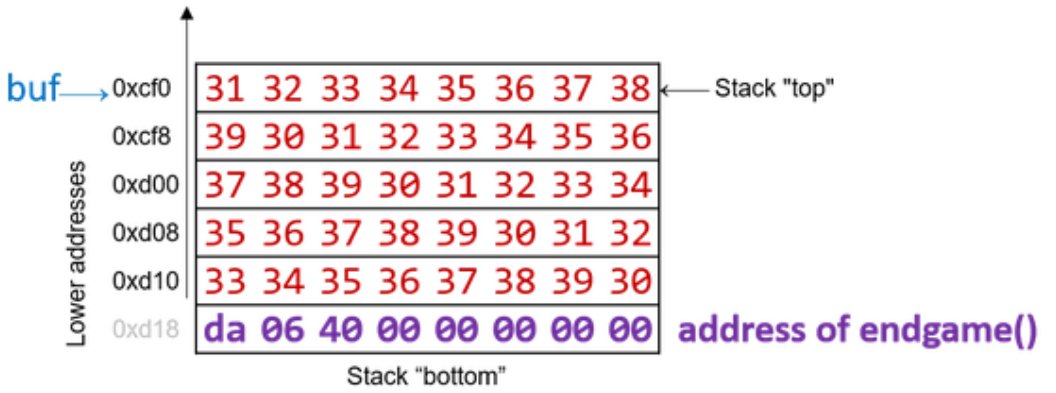
Registers	
%eax	0x0
%edi	0x400888
%rsi	0xcf0
%rsp	0xcf0
%rbp	0xd10

Immediately after call to `scanf()`

Memory	
0x400873	"Enter secret number"
0x400888	"%s"

Idea: Overrun the buffer given to *scanf* to overwrite the return address of *main* to go *endgame*.

See the book to try this yourself



call stack

Protecting against buffer overflow

Stack randomization randomizes the location of the call stack so hackers can rely on the same memory addresses being re-used when the program is run.

Stack corruption detection uses flags to detect mis-use of the stack

Limited executable regions restricts executable code to only be in certain regions of memory

All methods are still vulnerable to hackers

Stack randomization can be circumvented using a **NOP sled** which reruns the program until an unlucky address location results in execution landing in the hacker’s code.

Stack corruption detection can also be exploited by hackers if they know how and where the flags are implemented.

Limited executable regions can be exploited by hackers who invoke functions directly from the executable regions

Best Defense: Good programming

Use length specifiers whenever possible

Table 1. C Functions with Length Specifiers

Instead of:	Use:
`gets(buf)`	`fgets(buf, 12, stdin)`
`scanf("%s", buf)`	`scanf("%12s", buf)`
`strcpy(buf2, buf)`	`strncpy(buf2, buf, 12)`
`strcat(buf2, buf)`	`strncat(buf2, buf, 12)`
`sprintf(buf, "%d", num)`	`snprintf(buf, 12, "%d", num)`