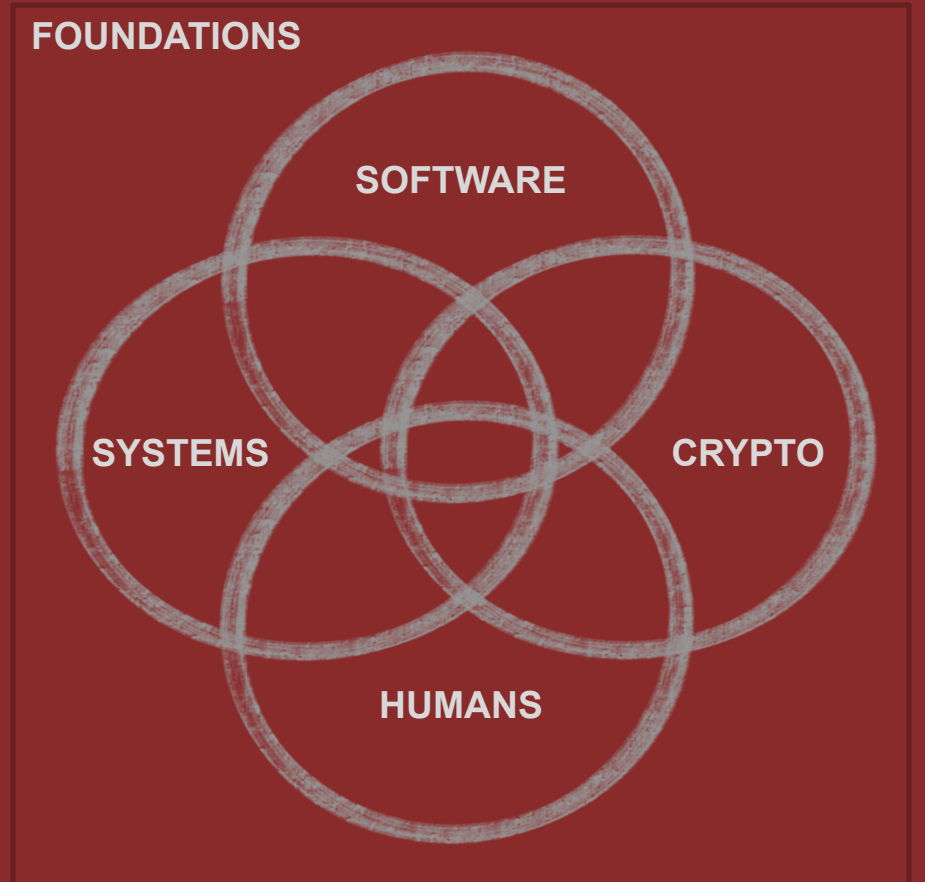


Διάλεξη #2 - x86 Basics and Buffer Overflows

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στην Ασφάλεια

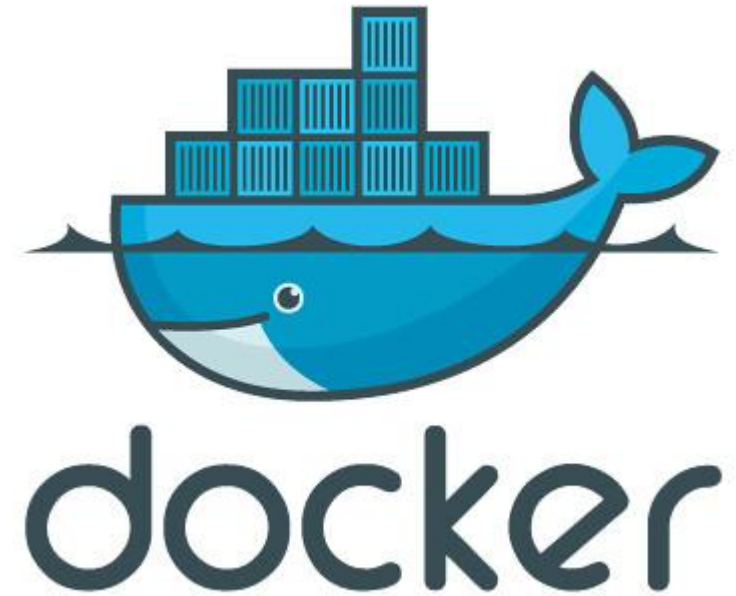
Θανάσης Αυγερινός



Huge thank you to [David Brumley](#) from Carnegie Mellon University for the guidance and content input while developing this class

Την Προηγούμενη Φορά

1. Docker



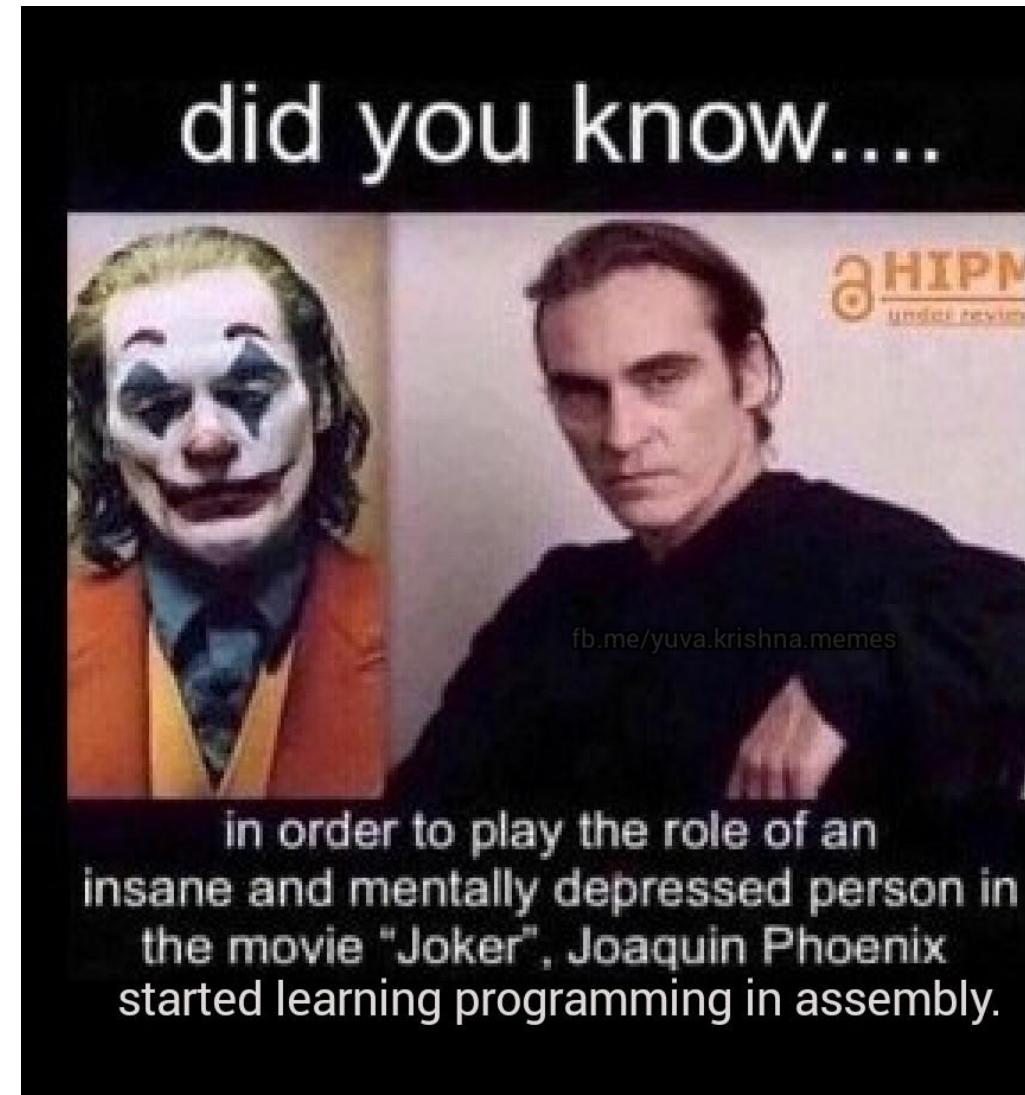
Ανακοινώσεις / Διευκρινίσεις

Ερωτήσεις:

- Γιατί είναι τεράστιο το docker image ethan42/iwconfig;
 - Περιέχει debugging εργαλεία προεγκατεστημένα + python3
- Γιατί το export δεν τρέχει στο image αλλά στο container;
 - Μπορείτε να χρησιμοποιήσετε docker save για images

Σήμερα

- x86 Fundamentals
 - Call Return Semantics
- Basics of buffer overflow attacks



Υπεθύμιση για Περιεχόμενα Μνήμης
(aka speedrunning progintro)

Η Μνήμη Οργανώνεται σε Bytes (Υπενθύμιση)

Το μέγεθος της μνήμης μετράται σε Bytes:

- 1 KB (KiloByte) = 1.000 Bytes
- 1 MB (MegaByte) = 1.000.000 Bytes
- 1 GB (GigaByte) = 1.000.000.000 Bytes

Μνήμη με
N Bytes

Byte 0

0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
1	1	1	0	0	0	1	1

Byte 1

Byte 2

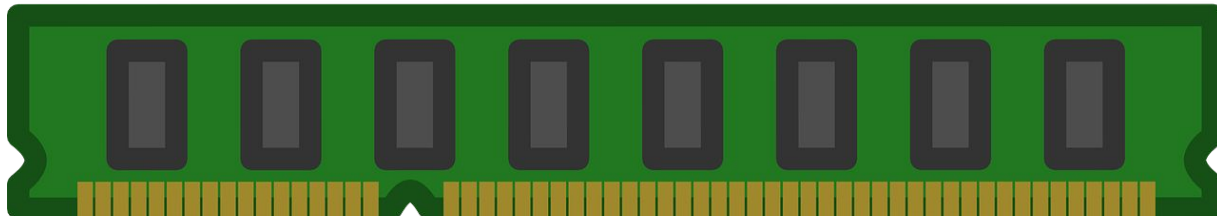
...

...

Byte N-1

0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0

Byte N



Κατηγορίες Μνήμης

Υπάρχουν 3 κατηγορίες μνήμης:

1. Η στοίβα (stack)
2. Ο σωρός (heap)
3. Η παγκόσμια / στατική μνήμη (global / static memory)

Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.



Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

Byte 31996

Byte 31997

Byte 31998

Byte 31999

Byte 32000

63

62

61

Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

```
...
```

```
char d = 64;
```

Byte 1

Byte 2

Byte 3

Byte 31996

Byte 31997

Byte 31998

Byte 31999

Byte 32000

63

62

61

Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

...

```
char d = 64;
```

Byte 1

Byte 2

Byte 3

Byte 31996

Byte 31997

Byte 31998

Byte 31999

Byte 32000

64

63

62

61

Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

Byte 31996

Byte 31997

Byte 31998

Byte 31999

Byte 32000

63

62

61

Η στοίβα (Stack)

Η **στοίβα (stack)** είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62;
```

Byte 1

Byte 2

Byte 3

Byte 31996

Byte 31997

Byte 31998

Byte 31999

Byte 32000

62

61

Τι αποθηκεύεται συνήθως στην στοίβα;

2: Ορίσματα που περνάμε στην συνάρτηση

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

1: Τοπικές
Μεταβλητές
ορισμένες
μέσα στην
συνάρτηση

3: Προσωρινά δεδομένα
(συνήθως μερικά bytes) που
αποθηκεύει ο μεταγλωττιστής

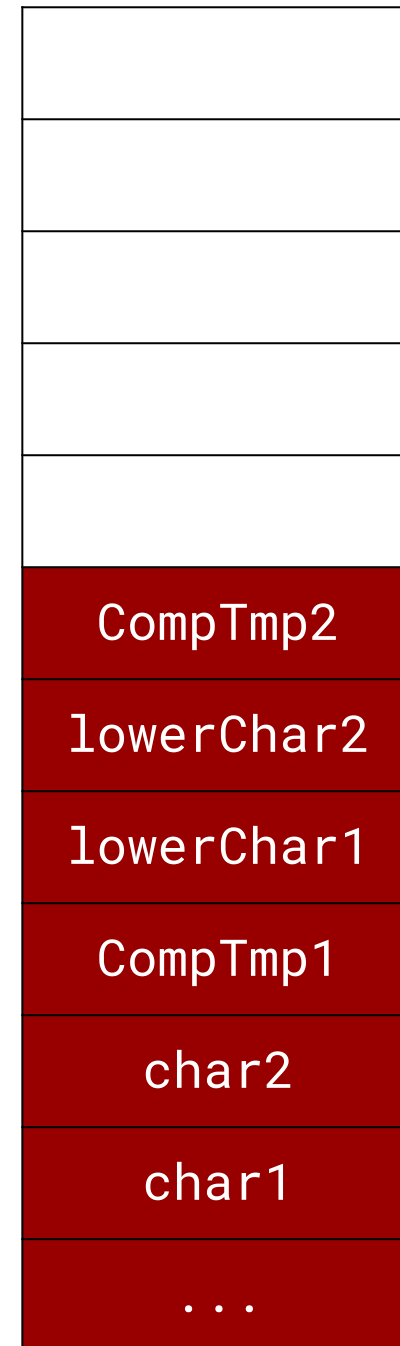
Τι αποθηκεύεται συνήθως στην στοίβα;

2: Ορίσματα που περνάμε στην συνάρτηση

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

1: Τοπικές Μεταβλητές ορισμένες μέσα στην συνάρτηση

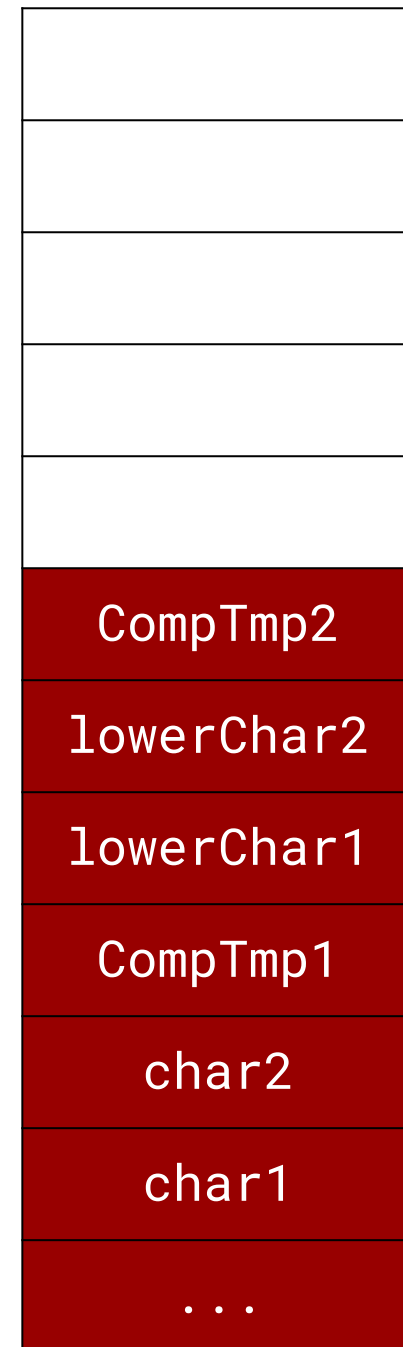
3: Προσωρινά δεδομένα (συνήθως μερικά bytes) που αποθηκεύει ο μεταγλωττιστής



Τι αποθηκεύεται συνήθως στην στοίβα;

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Χώρος που δεσμεύεται στην στοίβα σε **κάθε** κλήση της συνάρτησης `equalIgnoreCase`. Αυτό το κομμάτι μνήμης λέγεται και διάγραμμα ενεργοποίησης (**activation record** ή **stack frame**) της συνάρτησης



Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}
```

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

equalIgnoreCase

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

Τι θα συμβεί όταν κληθεί η συνάρτηση tolower την πρώτη φορά;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}
```

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```



tolower

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}
```



```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```



tolower

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

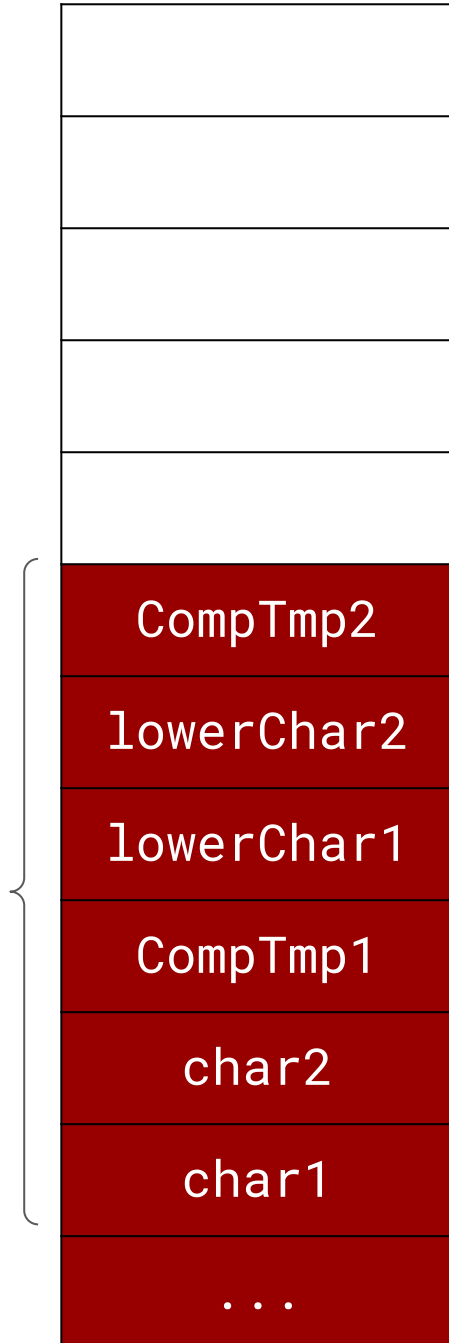
Τι θα συμβεί όταν εκτελεστεί η εντολή return;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```



equalIgnoreCase



Τι θα συμβεί όταν εκτελεστεί η δεύτερη κλήση tolower;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}
```



```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```



tolower

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

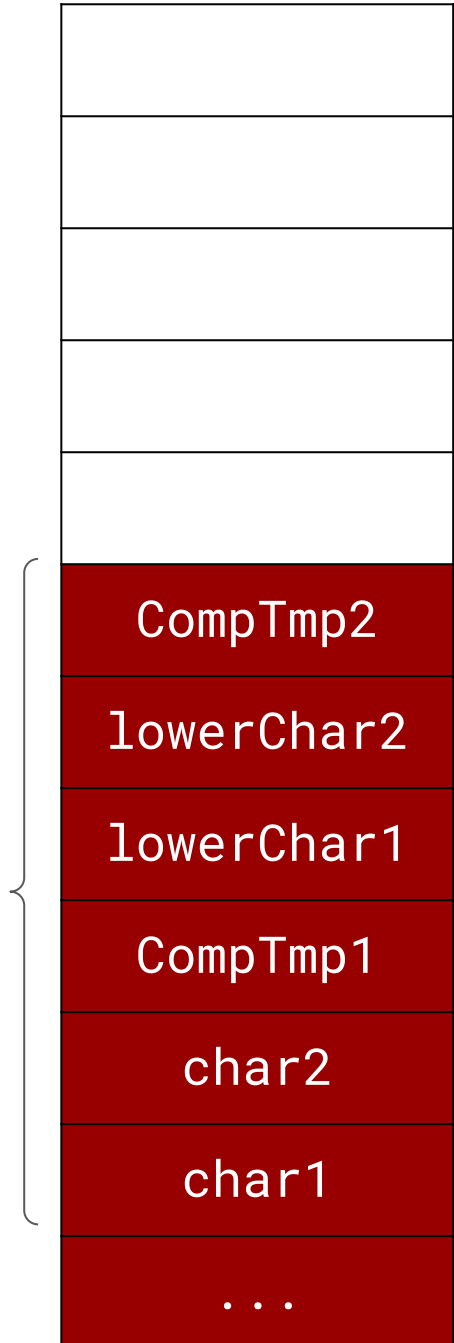
Τι θα συμβεί όταν εκτελεστεί η εντολή return;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```



equalIgnoreCase



Τι θα συμβεί όταν εκτελεστεί η return της equalIgnoreCase;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Τι θα συμβεί όταν εκτελεστεί η return της equalIgnoreCase;



Πόσα bytes έχει ένας int;

Endianness

Endianness λέμε τον τρόπο με τον οποίο οι ακέραιοι αποθηκεύονται στην μνήμη. Οι ακέραιοι αποτελούνται από πολλά bytes και επομένως πρέπει να αποφασίσουμε αν τους αποθηκεύουμε από το μικρότερο στο μεγαλύτερο (little endian) ή από το μεγαλύτερο στο μικρότερο (big endian).

68 ('h')	65 ('e')	6c ('l')	6c ('l')
----------	----------	----------	----------

Το πρώτο byte στην μνήμη είναι το μικρότερο byte του αριθμού

6c6c6568

Παράδειγμα Endianness

Τι θα τυπώσει το παρακάτω:

```
#include <stdio.h>

int main() {
    int x = 0x42434445;
    char * bytes = (char*)&x;
    int i;
    for(i = 0; i < sizeof(int) / sizeof(char); i++)
        printf("%02x\n", bytes[i]);
    return 0;
}
```

Παράδειγμα Endianness

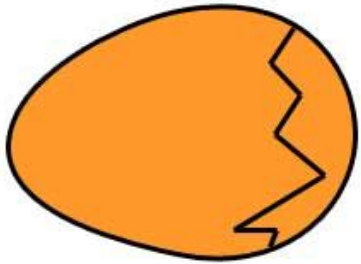
Τι θα τυπώσει το παρακάτω:

```
#include <stdio.h>

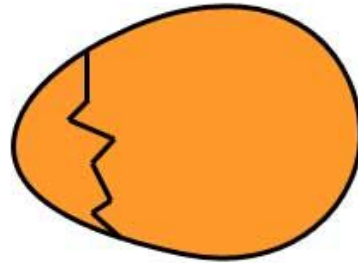
int main() {
    int x = 0x42434445;
    char * bytes = (char*)&x;
    int i;
    for(i = 0; i < sizeof(int) / sizeof(char); i++)
        printf("%02x\n", bytes[i]);
    return 0;
}
```

```
$ ./int
45
44
43
42
```

ENDIANNESS



BIG ENDIAN - The way
people always broke
their eggs in the
Lilliput land




LITTLE ENDIAN - The
way the king then
ordered the people to
break their eggs

- Little endian (x86/x86_64/arm/...):
LSB stored in smallest memory
address
- Big endian (PPC/SPARC/...):
MSB stored in smallest memory
address
- Bi-endian (Some arm and mips):
Can switch dynamically

Credit:

<http://flickeringtubelight.net/blog/2004/05/big-endian-and-little-endian-storage-schemes-how-to-remember/>



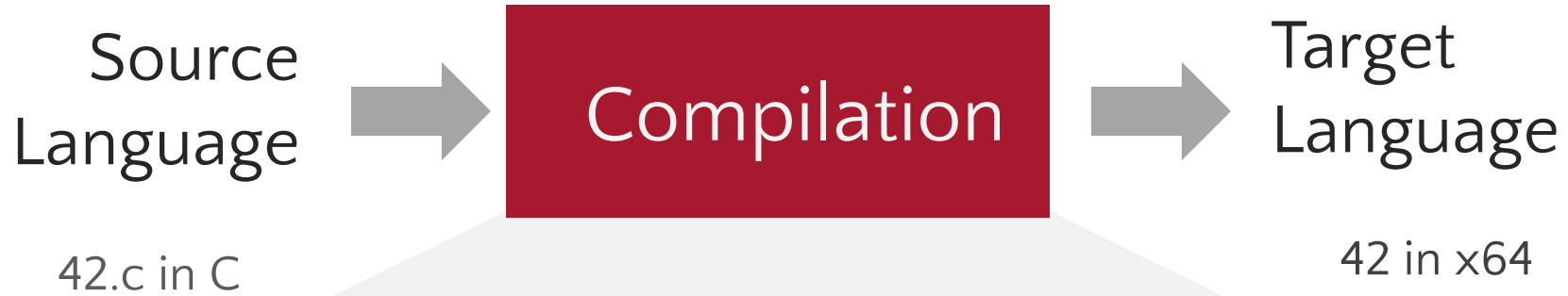
**Compilation
Workflow &
Execution
Semantics**

To answer

Is this program safe?

We must first know

What will executing it do?



> Preprocessor

\$ gcc -E

> Compiler

> Assembler

> Linker

```
#include <stdio.h>

void answer_function(char *name, int x)
{
    printf("%s answered %d\n", name, x);
}

int main(int argc, char *argv[])
{
    answer_function("hello", 42);
}
```

#include expansion

#define substitution

> Preprocessor

```
$ gcc -S
```

> Compiler

> Assembler

> Linker

```
answer_function:  
.LFB0:  
.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)  
movl %esi, -12(%rbp)  
movl -12(%rbp), %edx
```

Generate assembly code

> Preprocessor

> Compiler

> Assembler

> Linker

```
$ as hello.s -o hello.o
$ file hello.o
hello.o: ELF 64-bit LSB relocatable,
x86-64, version 1 (SYSV), not stripped
```

Create object code
(the ones and zeros a computer executes)

\$ ld <opts>
(this one is complicated; see gcc -v)

> Preprocessor

> Compiler

> Assembler

> Linker

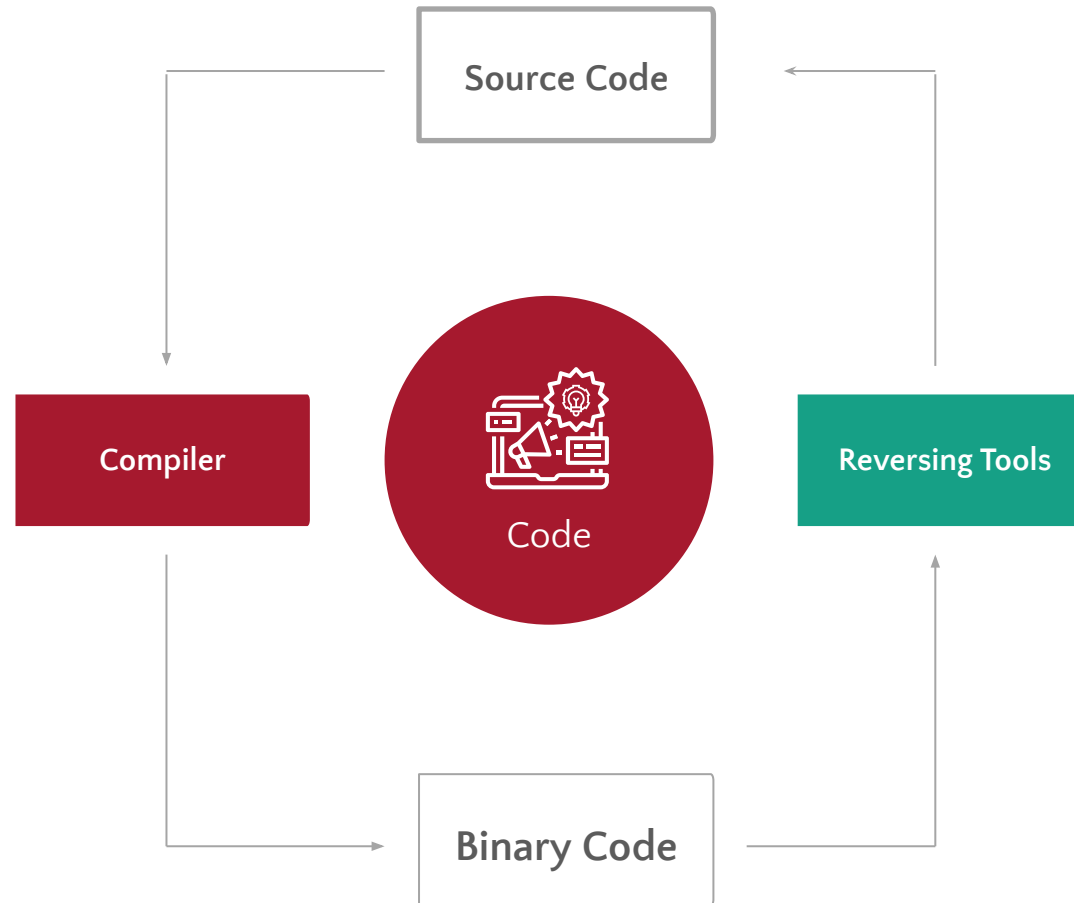
```
gcc -v hello.c
...
/usr/lib/gcc/x86_64-linux-gnu/10/collect2 -plugin
/usr/lib/gcc/x86_64-linux-gnu/10/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper
-plugin-opt=-fresolution=/tmp/ccLvq7Mj.res -plugin-opt=-pass-through=-lgcc
-plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc
-plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr
-m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/Scrt1.o
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/10/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-gnu/10
-L/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/10/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/./lib
-L/usr/lib/x86_64-linux-gnu -L/usr/lib/./lib -L/usr/lib/gcc/x86_64-linux-gnu/10/../../../../
/tmp/ccRnv5wm.o -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state
--as-needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-linux-gnu/10/crtendS.o
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/crtn.o
```

Linker links with other compilation units (.o) and libraries (.a) to produce an executable.

Compilation is not necessarily invertible

Compilation

Turn source code into executable code using programmer-defined abstractions as a guide translation.



Reversing

Best case effort to recover source code from binary code.

Interview Question: How would you see the content of a binary program?

Binary

Code Segment
(.text)

Data Segment
(.data)

...

The final **executable binary**

- Text segment
- Data for constants like “hello world” and globals
- And more...



```
# read the various sections. Also see `nm`  
$ readelf -S <file>
```

```
# -d: disassemble  
# -S: match symbol locations to source  
$ objdump <-d> <-S> file
```



**BASIC
EXECUTION
MODEL**

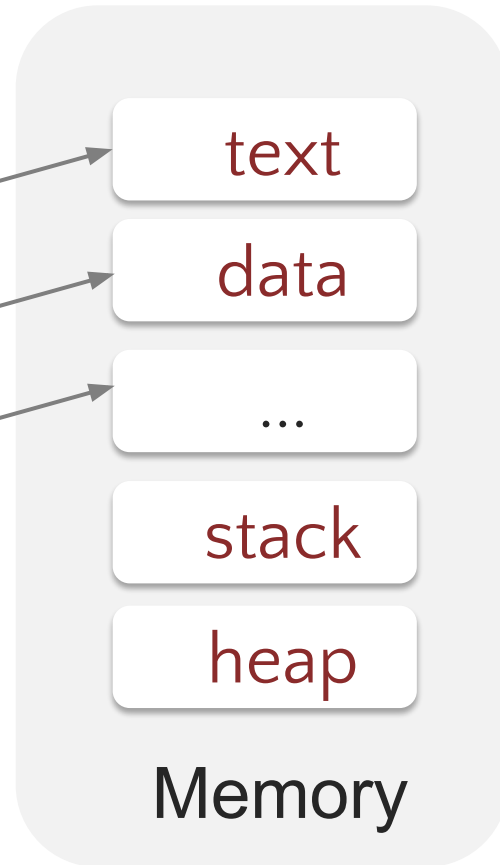
1

Executable file segments



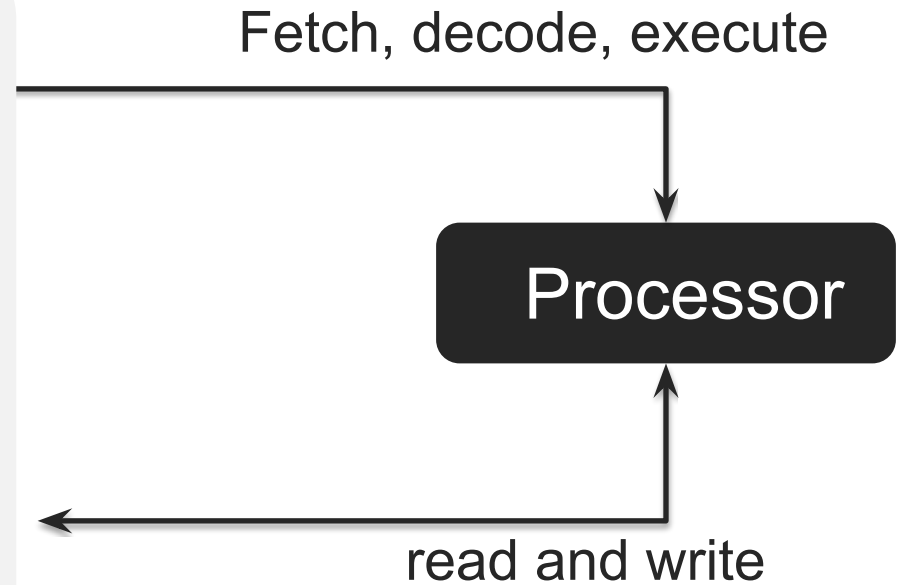
2

Segments loaded and process memory created

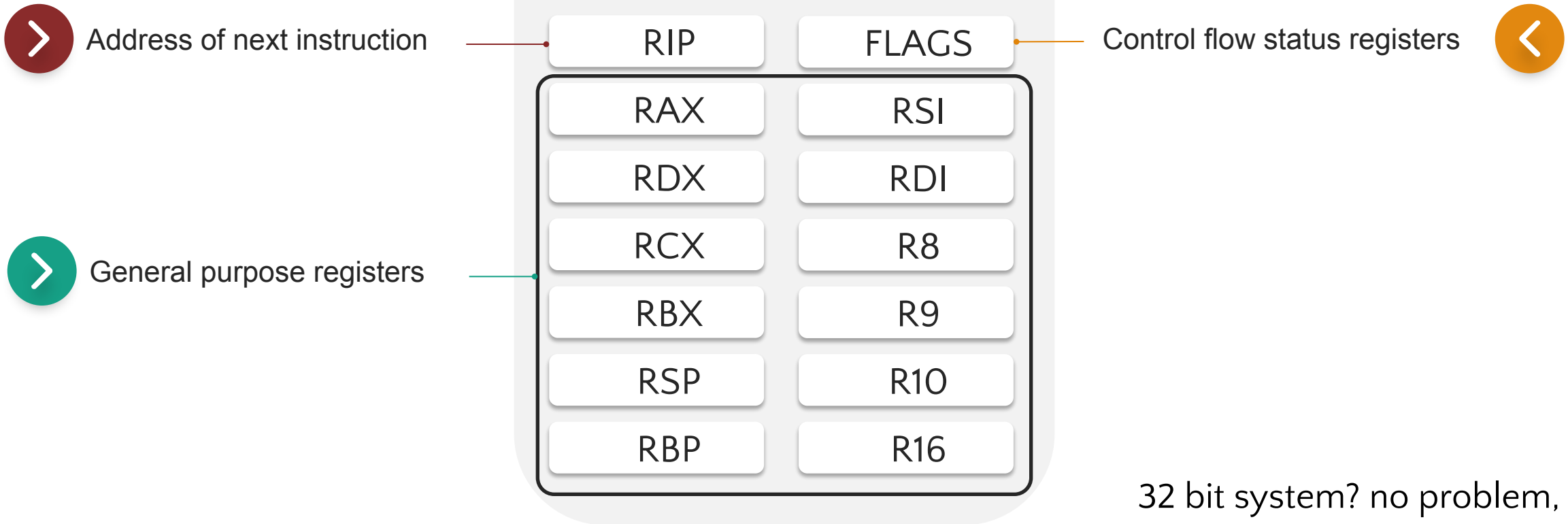


3

Processor executes process in memory

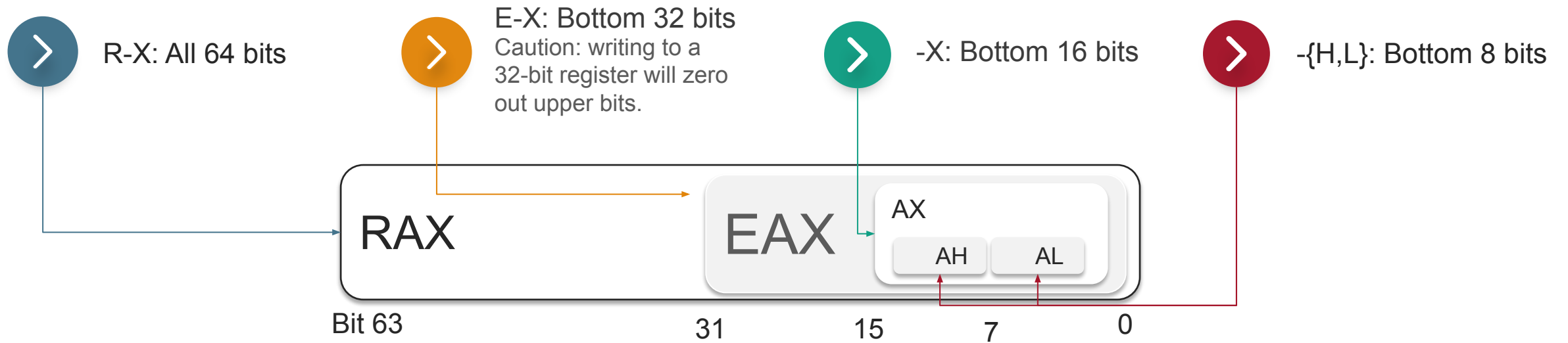


x64 Processor



32 bit system? no problem,
just EIP instead of RIP, EAX
instead of RAX, etc

Parts of a register can be addressed directly



Basic Ops and AT&T Vs. Intel Syntax

Meaning	AT&T	Intel
<code>rbx = rax</code>	<code>movq %rax, %rbx</code>	<code>mov rbx, rax</code>
<code>rax = rax + rbx</code>	<code>addq %rbx, %rax</code>	<code>add rax, rbx</code>
<code>rcx = rcx << 2</code>	<code>shl \$2, %rcx</code>	<code>shl rcx, 2</code>



AT&T is at odds w/ assignment order

- Default for objdump
- Default in UNIX



Intel mirrors assignment order

- `objdump -M intel`
- Default for windows





Memory Operations

Memory Loads

What do these instructions do?

1. `movb (%rax), dl`

2. `mov (%rax), edx`

↑
src

↑
dst

- Bracket “[]” indicates Intel
- Paren “()” indicates AT&T

Register	Value
rax	0x3
rdx	0x0
rbx	0x5

Byte	Address
0xff	6
0xee	5
0xdd	4
0xcc	3
0xbb	2
0xaa	1
0x00	0

Memory Loads

What do these instructions do?

1. `movb (%rax), dl`

1. `mov (%rax), edx`

Register	Value
<code>rax</code>	<code>0x3</code>
<code>rdx</code>	<code>0x0</code>
<code>rbx</code>	<code>0x5</code>

Byte	Address
<code>0xff</code>	6
<code>0xee</code>	5
<code>0xdd</code>	4
<code>0xcc</code>	3
<code>0xbb</code>	2
<code>0xaa</code>	1
<code>0x00</code>	0



1. Moves `0xcc` into `dl`
2. Moves `0xcc – 0xff` into `edx`.
But what number is this?

EXAMPLE

We'll assume x86/x86_64 from now on:

- Address a goes into register bits 0-7
- Address $a+1$ in the 8-15
- And so on

```
mov (rax), rcx
```

Register **Value**

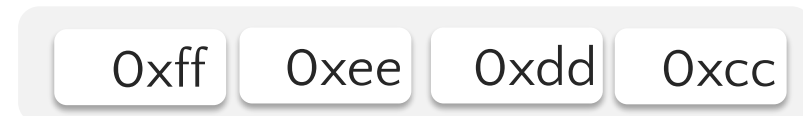
rax 0x3

rdx 0xcc

rbx 0x5


Address	
0xff	6
0xee	5
0xdd	4
0xcc	3
0xbb	2
0xaa	1
0x00	0

RDX



32

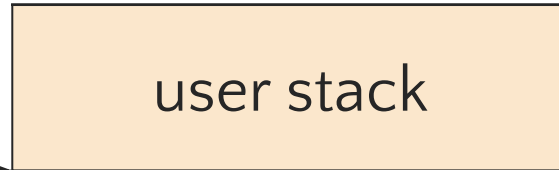
0



Process Memory Organization

> Stack Grows Down

%rsp



> Shared libraries go up



> Heap (brk syscall) grows up

brk



0x00007FFFFFFFFFFFFF
(128TB)

64-bit pointers and upper bits must match 47!



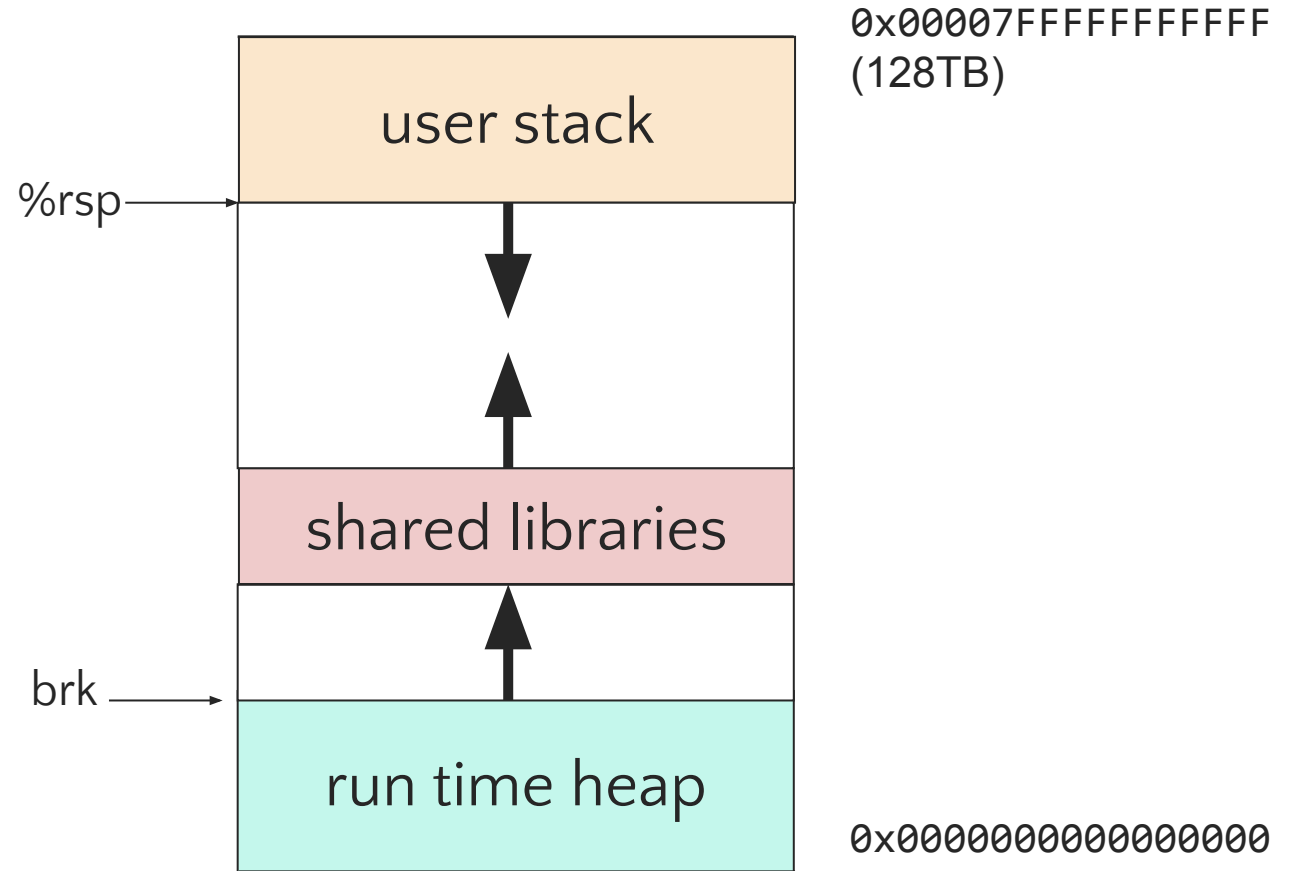
0x0000000000000000

On the stack

- Local variables
- Lifetime: stack frame

On the heap

- Dynamically allocated via new/malloc/etc.
- Lifetime: until freed



Procedures

- Assembly code doesn't have procedures
- Compilers *implement* procedures
 - On the stack
 - Following the call/return stack discipline

The compiler needs to implement a call stack, calling convention, and a way to pass arguments for code being compiled, third-party binary libraries, and the OS

```
int f1(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = f2(c, buf);
    return d;
}
```



Need to access arguments



Need space to store local variables (buf, c, and d)



Need space for call arguments



Need a place to return values

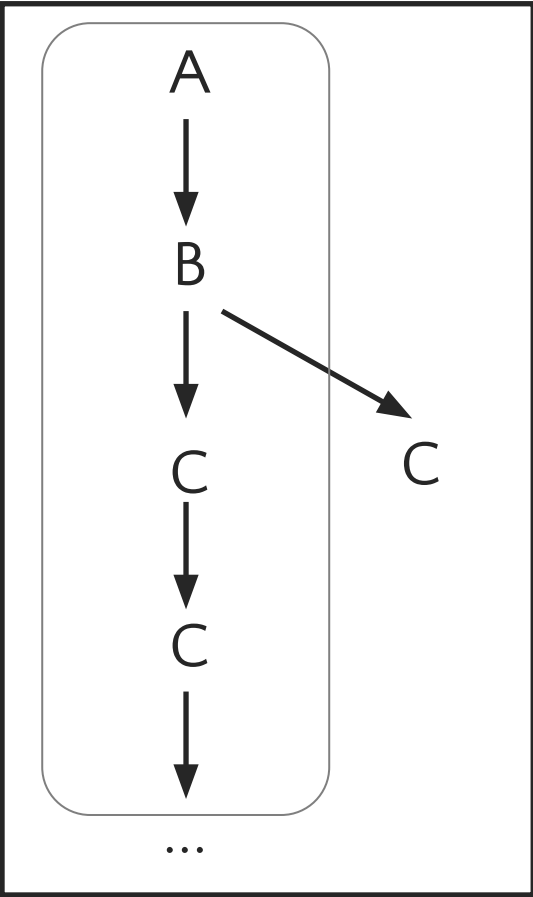
Recall the basic stack semantics of function calls

```
A(...)  
{  
  ...  
  B()  
  ...  
}
```

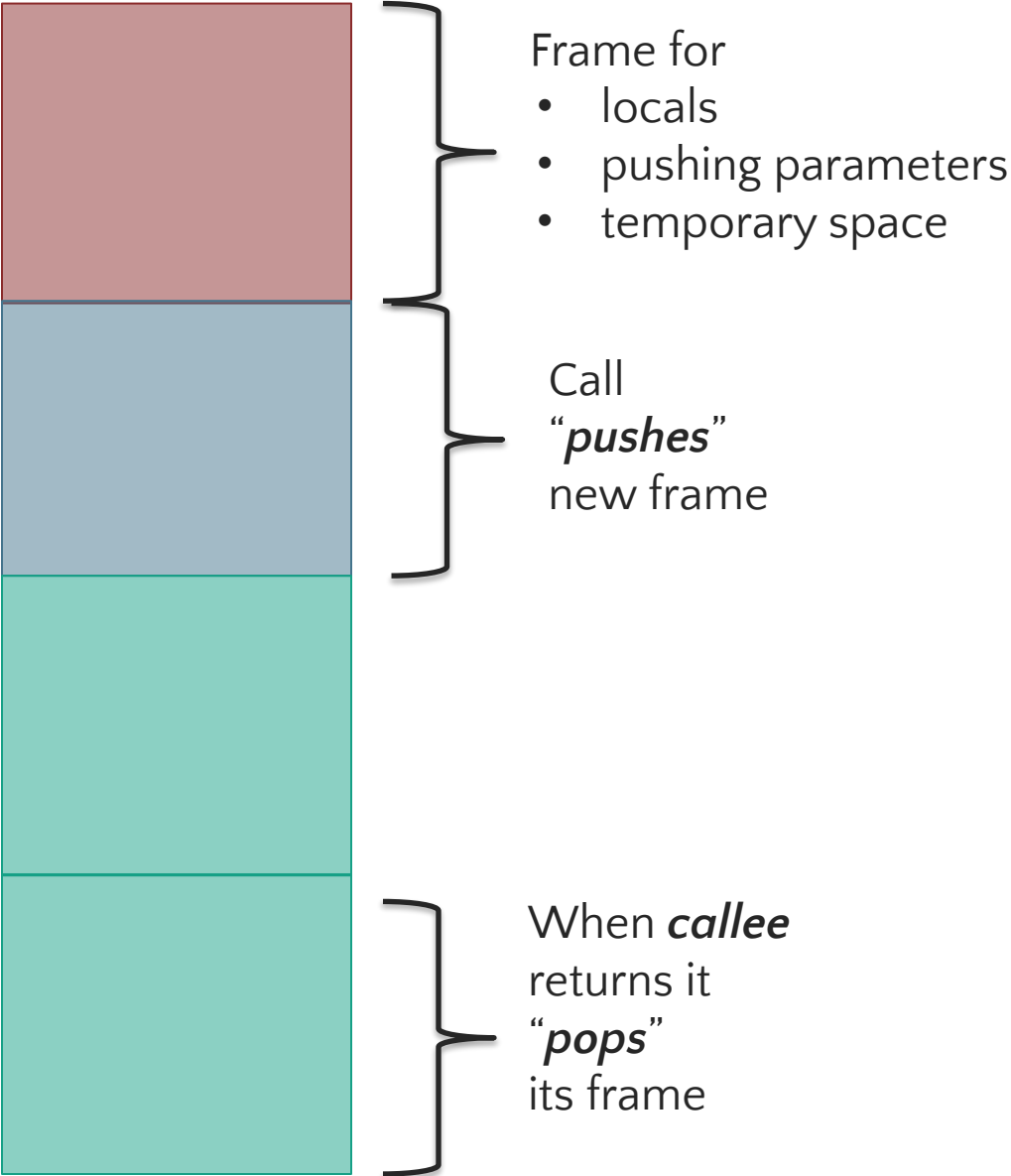
```
B(...)  
{  
  ...  
  C()  
  ...  
  C()  
}
```

```
C(...)  
{  
  ...  
  C()  
  ...  
}
```

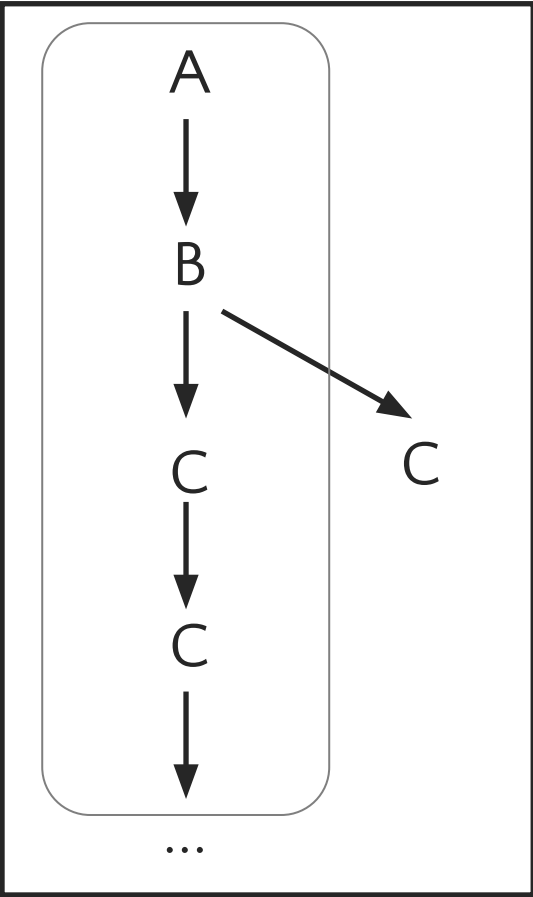
Function Call Chain



The Stack



Function Call Chain



A Toy Example

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

A Toy Example

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

Transfer control to functions using the call instruction

```
080491d7 <print>:
...
80491f2:    e8 69 fe ff ff    call 8049060 <strcpy@plt>
80491f7:    83 c4 10          add $0x10,%esp
...
8049202:    e8 69 fe ff ff    call 8049070 <puts@plt>
8049207:    83 c4 10          add $0x10,%esp
...
804920f:    c3              ret

08049210 <main>:
...
804925e:    e8 74 ff ff ff    call 80491d7 <print>
8049263:    83 c4 10          add $0x10,%esp
...
8049274:    c3              ret
```

Return to the caller using the ret instruction

Call - Ret Semantics

Call semantics: Store address of next instruction (the return address) where the current stack pointer is pointing and then jump to target address. call target is equivalent to:

```
push next_address  
jmp target
```

`mem[sp] = next_addr; sp -= sizeof(next_addr)`

Ret semantics: Read address of next instruction from the stack (the return address) and jump to it. ret is equivalent to:

```
pop next_address  
jmp next_address
```

`next_addr = mem[sp]; sp += sizeof(next_addr)`

A Toy Example

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

```
804925e:    e8 74 ff ff ff    call 80491d7 <print>
esp = 0xffffd640; eip = 0x804925e
```

0xffffd66c

...

main return addr

main locals vars

0xffffd640

A Toy Example

```
void print(char * message) {  
    char buffer[64];  
    strcpy(buffer, message);  
    printf("%s\n", buffer);  
}  
  
int main(int argc, char ** argv) {  
    ...  
    print(argv[1]);  
    return 0;  
}
```

```
80491d7:    53      push   %ebx  
esp = 0xffffd63c; eip = 0x80491d7
```

0xffffd66c

...

main return addr

main locals vars

0xffffd640

print return addr
0x8049263

A Toy Example

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

```
8049202:    e8 69 fe ff ff  call    8049070 <puts@plt>
```

```
esp = 0xffffd5e0; eip = 0x8049202
```

0xffffd66c

...

main return addr

main locals vars

0xffffd640

print return addr
0x8049263

print locals
including buffer
(at least 64
bytes)

0xffffd5e0

A Toy Example

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```



```
804920f:    c3                ret
esp = 0xffffd63c; eip = 0x804920f
```

0xffffd66c

...

main return addr

main locals vars

0xffffd640

print return addr
0x8049263

0xffffd5e0

A Toy Example

```
void print(char * message) {  
    char buffer[64];  
    strcpy(buffer, message);  
    printf("%s\n", buffer);  
}  
  
int main(int argc, char ** argv) {  
    ...  
    print(argv[1]);  
    return 0;  
}
```

8049263: 83 c4 10

esp = 0xffffd640; eip = 0x8049263

0xffffd66c

...

main return addr

main locals vars

0xffffd640

0xffffd5e0

add \$0x10, %esp

62



Buffer Overflows

What Are Buffer Overflows?

A *buffer overflow* occurs when data is written outside of the space allocated for the buffer

- C does not check that writes are in-bounds
1. Stack-based
 - what we'll cover today
 2. Heap-based
 - more advanced
 - very dependent on system and library version

A Toy Example

Input: ./bof0 "ABCD"

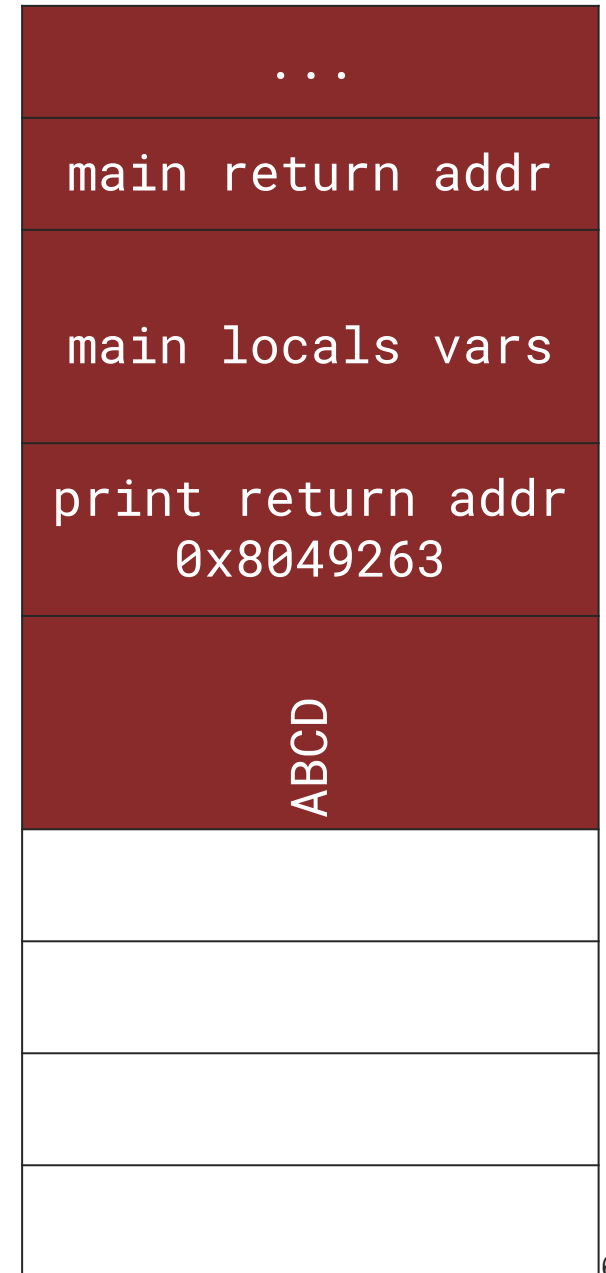
```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

0xffffd66c

0xffffd640

0xffffd5e0



8049202: e8 69 fe ff ff call 8049070 <puts@plt>

esp = 0xffffd5e0; eip = 0x8049202

A Toy Example

Input: `./bof0 "AAAAAA...AAA"`

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

0xffffd66c

...

main return addr

main locals vars

0xffffd640

AAAAA...AAAA

0xffffd5e0

8049202: e8 69 fe ff ff call 8049070 <puts@plt>

esp = 0xffffd5e0; eip = 0x8049202

A Toy Example

Input: `./bof0 "AAAAAA...AAAAA...AAA"`

```
void print(char * message) {  
    char buffer[64];  
    strcpy(buffer, message);  
    printf("%s\n", buffer);  
}  
  
int main(int argc, char ** argv) {  
    ...  
    print(argv[1]);  
    return 0;  
}
```

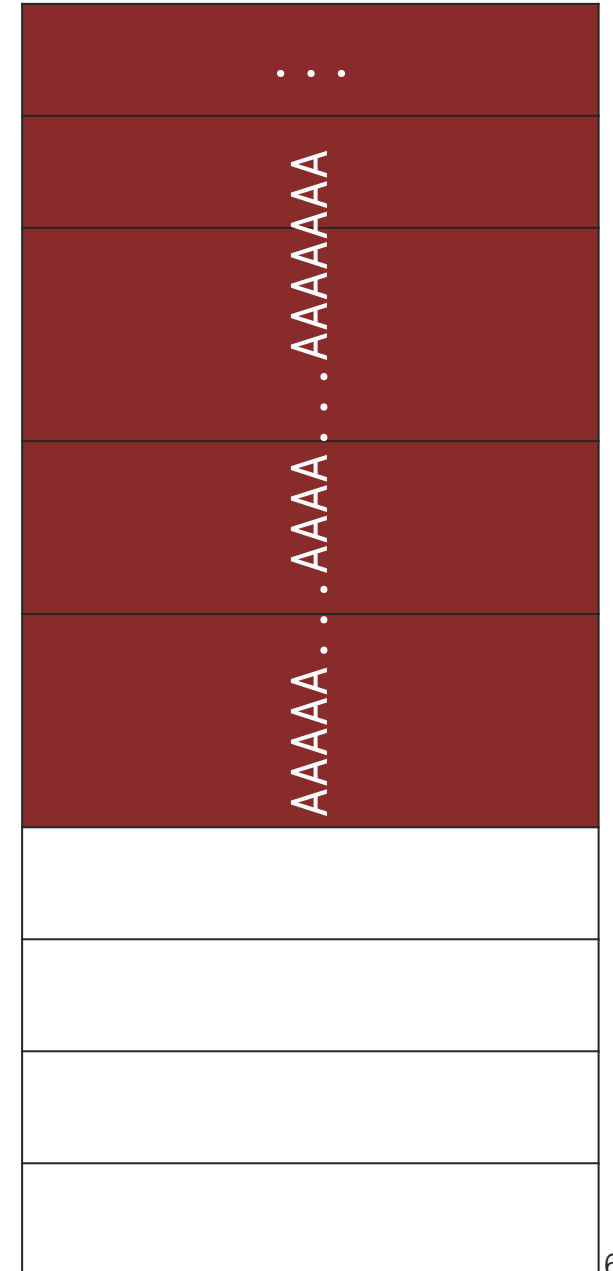
0xffffd66c

0xffffd640

0xffffd5e0

8049202: e8 69 fe ff ff call 8049070 <puts@plt>

esp = 0xffffd5e0; eip = 0x8049202



A Toy Example

Input: ./bof0 "AAAAAA...AAAAA...AAA"

```
void print(char * message) {
    char buffer[64];
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main(int argc, char ** argv) {
    ...
    print(argv[1]);
    return 0;
}
```

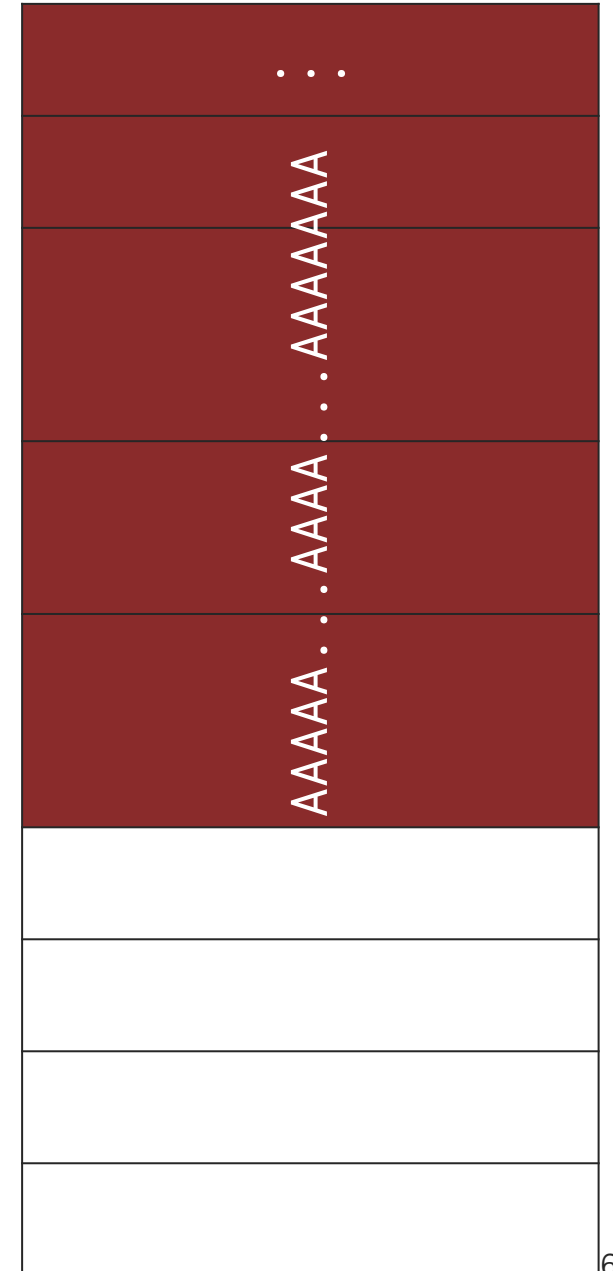
804920f: c3 ret

esp = 0xffffd63c; eip = 0x804920f

0xffffd66c

0xffffd640

0xffffd5e0



A Toy Example

Input: `./bof0 "AAAAAA...AAAAA...AAA"`

```
void print(char * message) {  
    char buffer[64];  
    strcpy(buffer, message);  
    printf("%s\n", buffer);  
}  
  
int main(int argc, char ** argv) {  
    ...  
    print(argv[1]);  
    return 0;  
}
```

0xffffd66c

0xffffd640

0xffffd5e0

...

AAAAA...AAAAA...AAAAA

Program received signal SIGSEGV, Segmentation fault.

0x41414141 in ?? ()

(gdb) i r eip

eip

0x41414141

0x41414141

A Toy Example

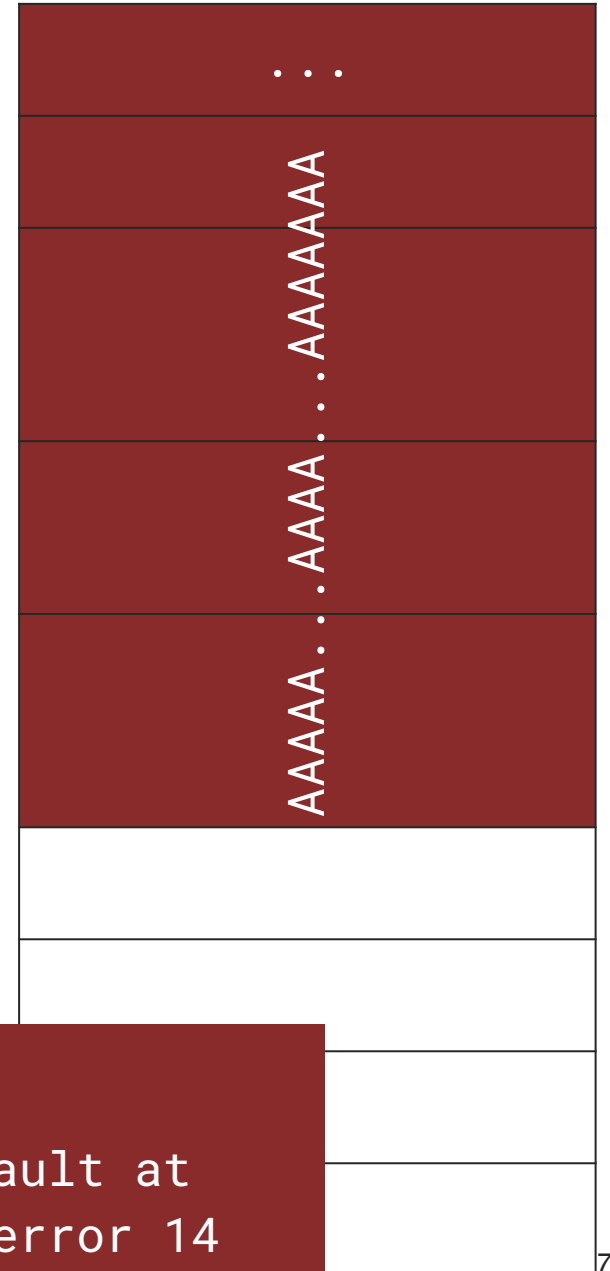
```
Input: ./bof0 "AAAAAA...AAAAA...AAA"
```

```
void print(char * message) {  
    char buffer[64];  
    strcpy(buffer, message);  
    printf("%s\n", buffer);  
}  
  
int main(int argc, char ** argv) {  
    ...  
    print(argv[1]);  
    return 0;  
}
```

0xffffd66c

0xffffd640

0xffffd5e0



```
$ dmesg -T | grep segfault  
[Thu Mar 21 10:58:52 2024] bof0_real[21893]: segfault at  
41414141 ip 0000000041414141 sp 00000000ffffd640 error 14  
in libc.so.6[f7d85000+20000]
```

Can we return control to another function?

Let's do it Live!

The slide features a central white circle containing the text "Shellcode Injection". This circle is surrounded by a light gray ring, which is further enclosed by a dark red outer ring. The text is centered and rendered in a bold, black, sans-serif font.

Shellcode Injection

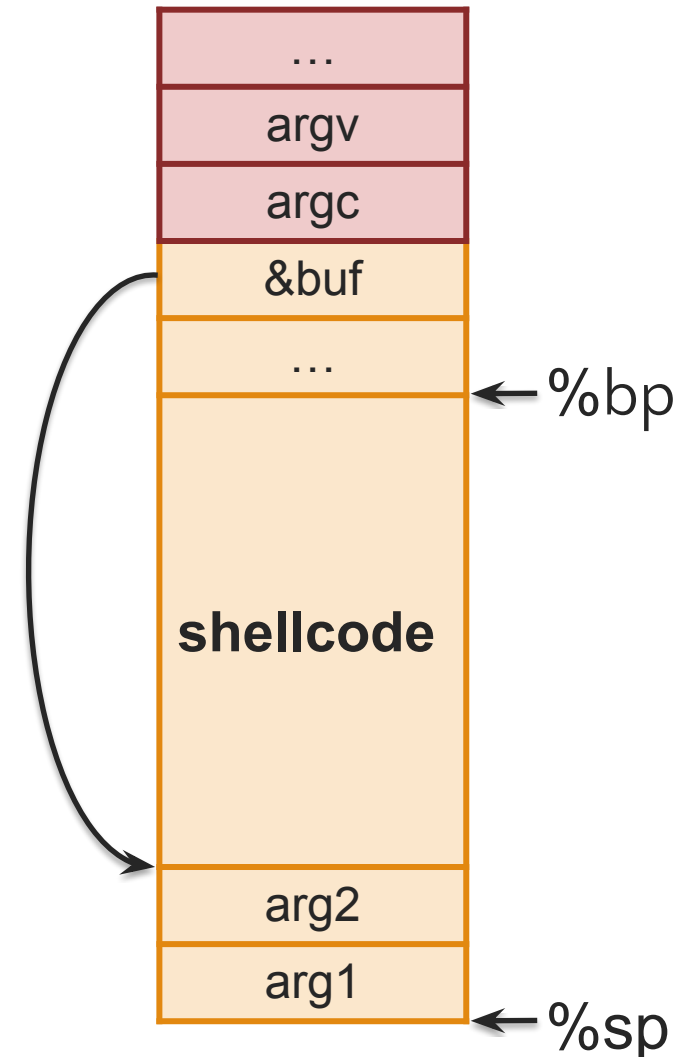
Shellcode

Traditionally exploits injected assembly instructions for `exec("/bin/sh")` into buffer.

Data Execution Prevention and other defenses have made this exploitation technique ineffective on consumer commercial OSes for over a decade.

Sadly, this is still applicable in areas like IoT, energy, and so on.

- Considered a basic skill for exploitation (even if not on your latest OS)
- See *"Smashing the stack for fun and profit"* for one string
- or search online OR *write it yourself!*



Recap

To generate *exploit* for a basic buffer overflow:

Determine size of **stack frame up to head of buffer**

1. Overflow buffer with the right size



computation

+

control

Το Bonus #0 μόλις ανέβηκε

Προθεσμία: Δευτέρα 1η Απριλίου, 23:59

[Bonus #0](#)

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!