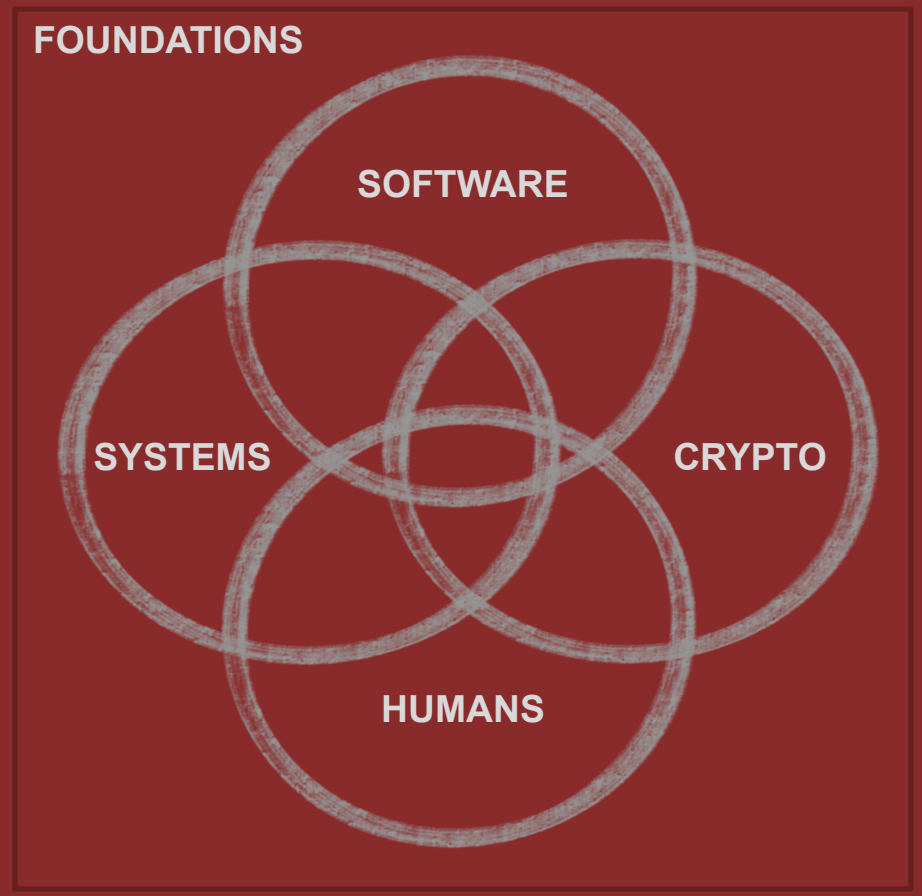
Διάλεξη #2 - x86 Basics and Buffer Overflows

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στην Ασφάλεια

Θανάσης Αυγερινός



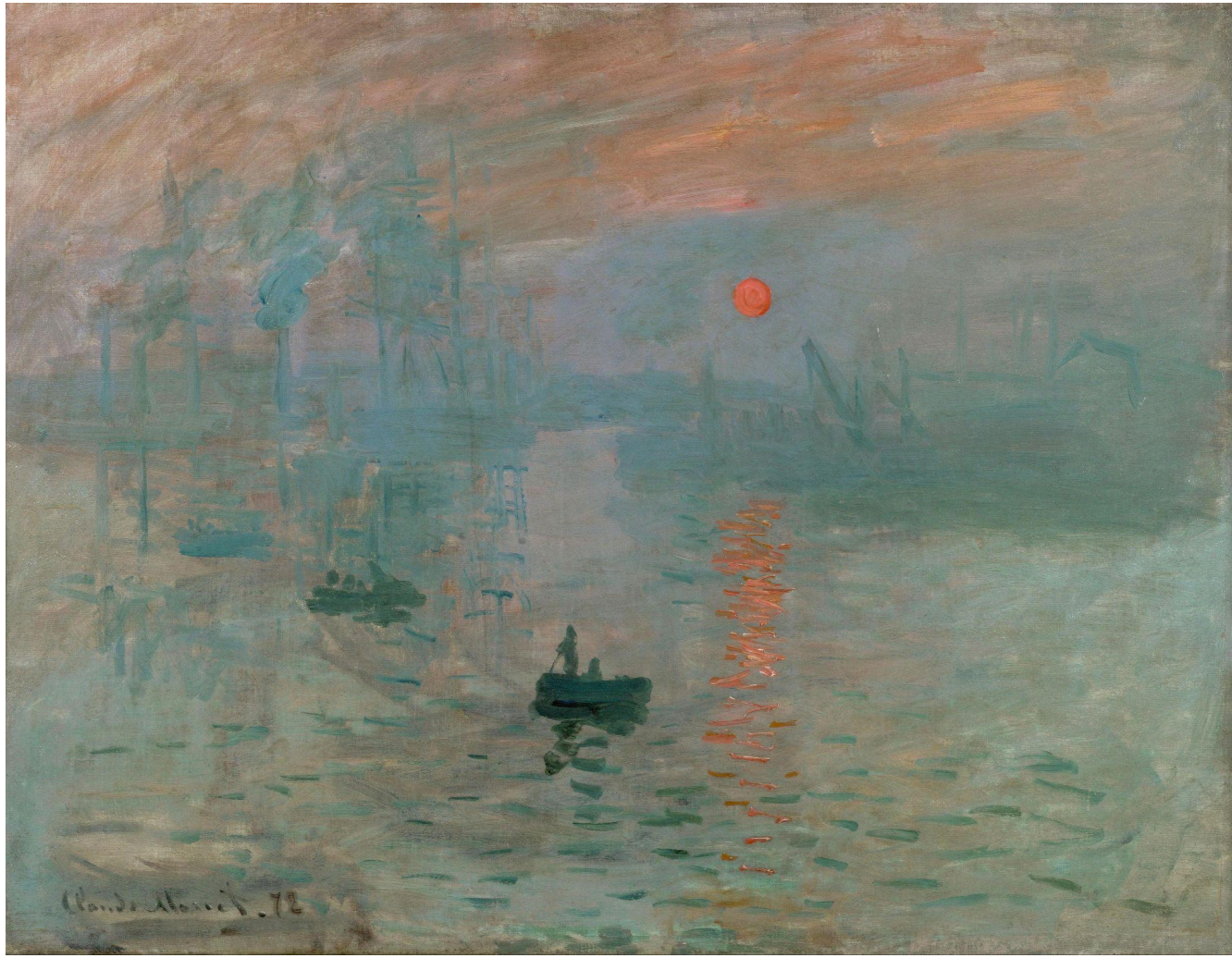
Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class

Ανακοινώσεις / Διευκρινίσεις

- Μόλις ανέβηκε η εργασία #0!
  - Προθεσμία: 12 Μαρτίου 23:59

Την Προηγούμενη Φορά

- Security Fundamentals
  - Adversaries
  - Threat Models
  - Security Properties
  - Trusted Computing Base (TCB)
  - Security Principles



Σήμερα

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

Security in the News

DISA, which provides services like drug and alcohol testing and background checks to more than 55,000 enterprises and a third of Fortune 500 companies, confirmed the data breach in a filing with Maine's attorney general on Monday.

DISA said it discovered it had been the victim of a "cyber incident" that affected a "limited portion" of its network on April 22, 2024. An internal investigation determined that a hacker had infiltrated the company's network on February 9, 2024, where they went unnoticed for over two months.

In a letter sent to those affected by the data breach, which includes individuals who underwent employee screening tests, DISA said the attacker "procured some information" from its systems.

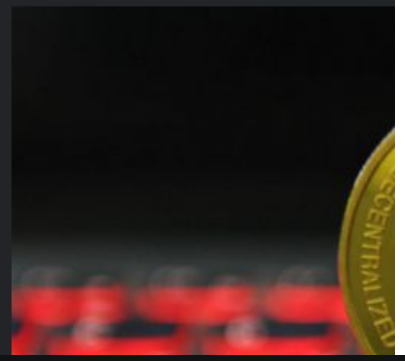
In a separate filing with the Massachusetts attorney general, DISA confirmed the stolen information included individuals' Social Security numbers; financial account information, including credit card numbers; and government-issued identification documents. This filing confirmed that more than 360,000 Massachusetts residents were affected by the breach.

OPSEC FAILURE OF THE YEAR

How North Korea pulled off a $1.5 billion crypto heist—the biggest in history

Attack on Bybit didn't hack infrastructure or exploit sm code. So how did it work?

DAN GOODIN – FEB 24, 2025 6:41 PM | 134



The cryptocurrency industry and those responsible for following Friday's heist, likely by North Korea, that drai exchange Bybit, making the theft by far the biggest eve

- Operate seamlessly across Windows, MacOS, and various wallet interfaces
- Show minimal signs of compromise while maintaining persistence
- Function as backdoors to execute arbitrary commands
- Download and execute additional malicious payloads
- Manipulate what users see in their interfaces

These hackers have also been long known for their relentless social engineering prowess. They often spend weeks or months building online personas that ultimately win the trust of targets. That persistence likely allowed the thieves who hit Bybit to somehow tamper with the UIs of each company employee whose digital imprimatur was required to move the funds out of cold storage—and ultimately into wallets the hackers controlled—all at breakneck speed.

As both Check Point and Trail of Bits point out, the lessons learned here bring cryptocurrency security back to some of the most basic elements, such as segmenting internal networks, adopting defense-in-depth practices that include multiple, overlapping controls for detecting and preventing sophisticated attacks, and preparation for scenarios precisely like this one.

Reminders: Memory, Stack, and Endianness (speedrunning progintro)

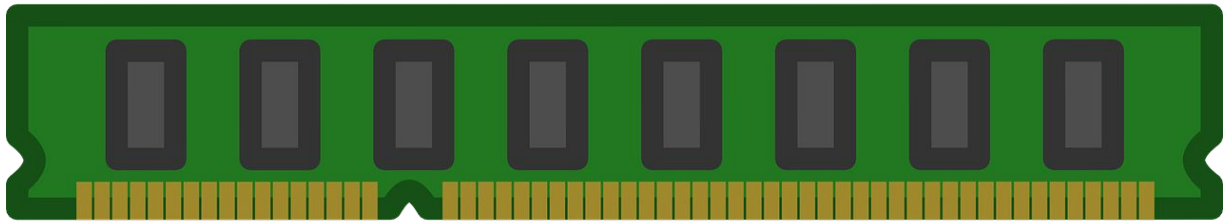
Η Μνήμη Οργανώνεται σε Bytes (Υπενθύμιση)

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
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
…				…				
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0



Κατηγορίες Μνήμης

Υπάρχουν 3 κατηγορίες μνήμης:

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

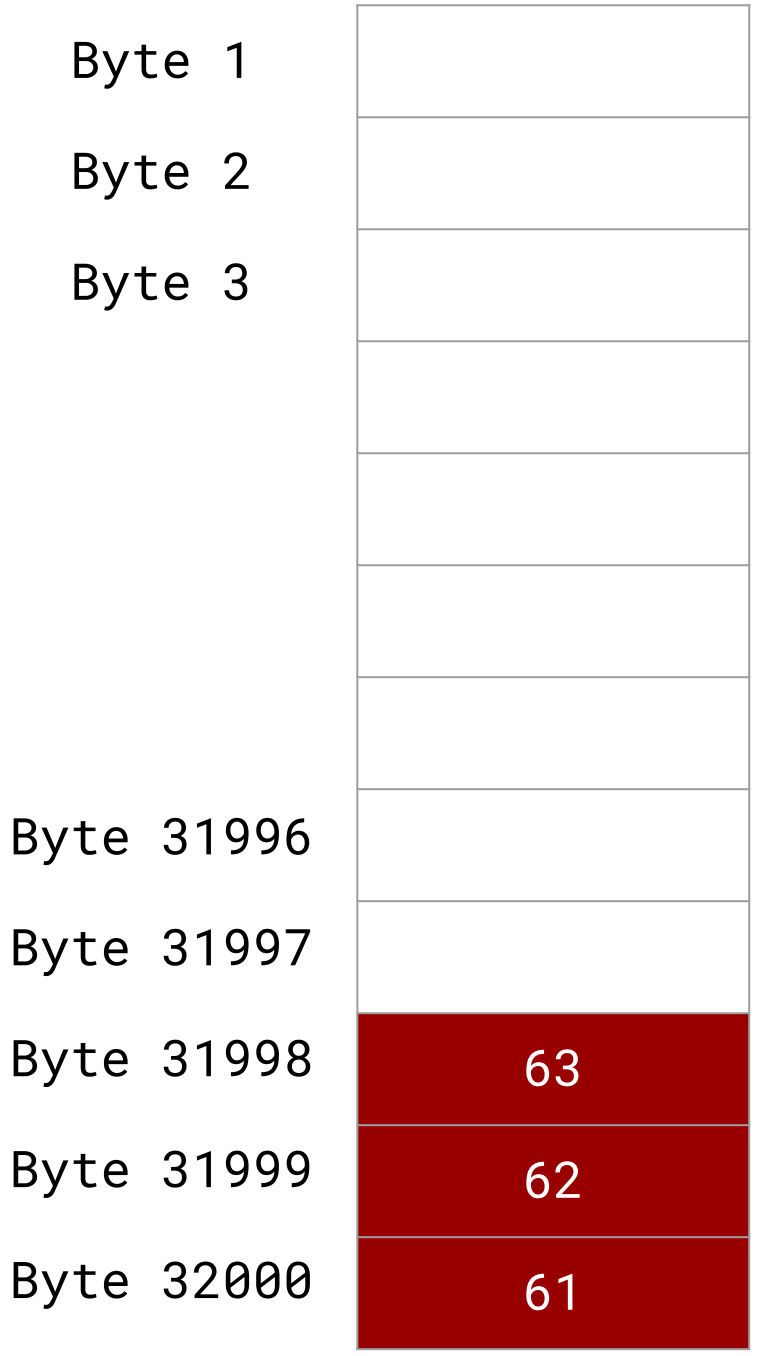
Η στοίβα (Stack)

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.

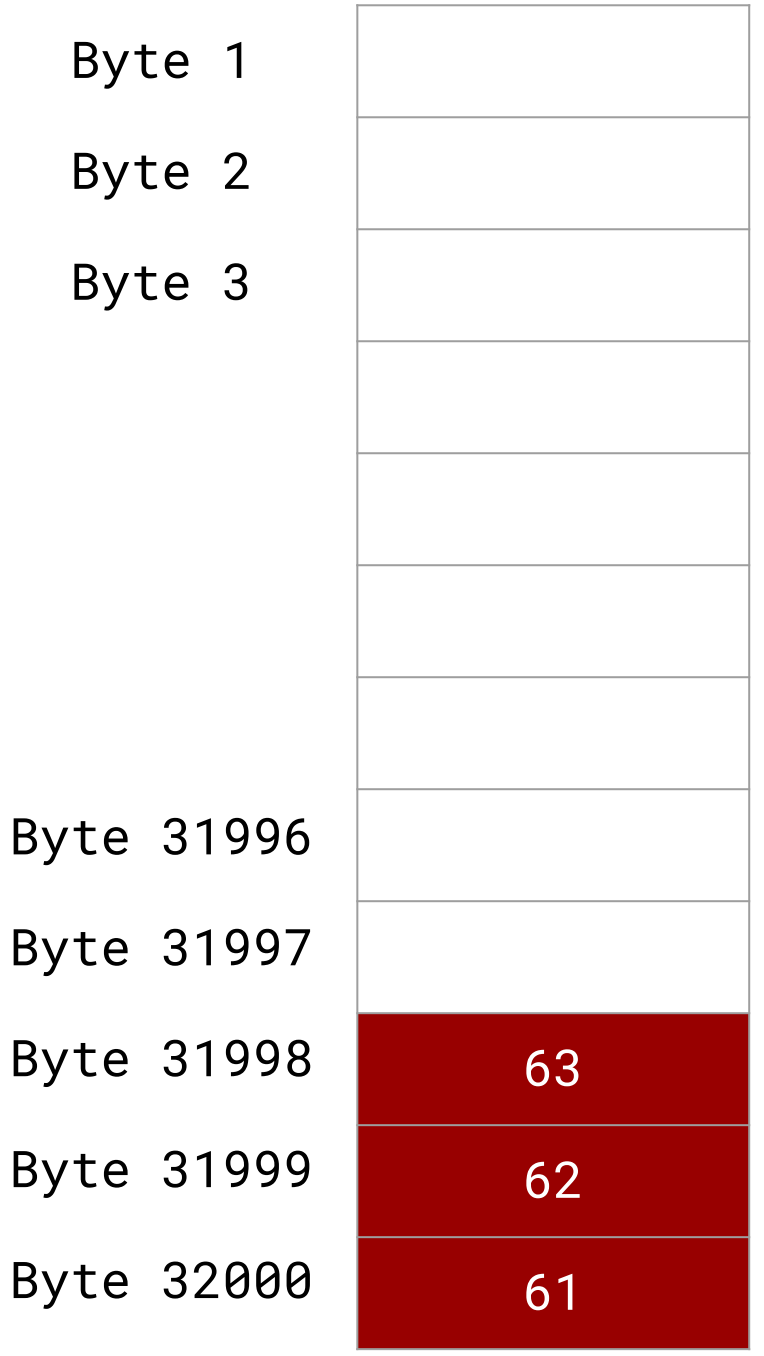
```
char a = 61; char b = 62; char c = 63;
```



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.

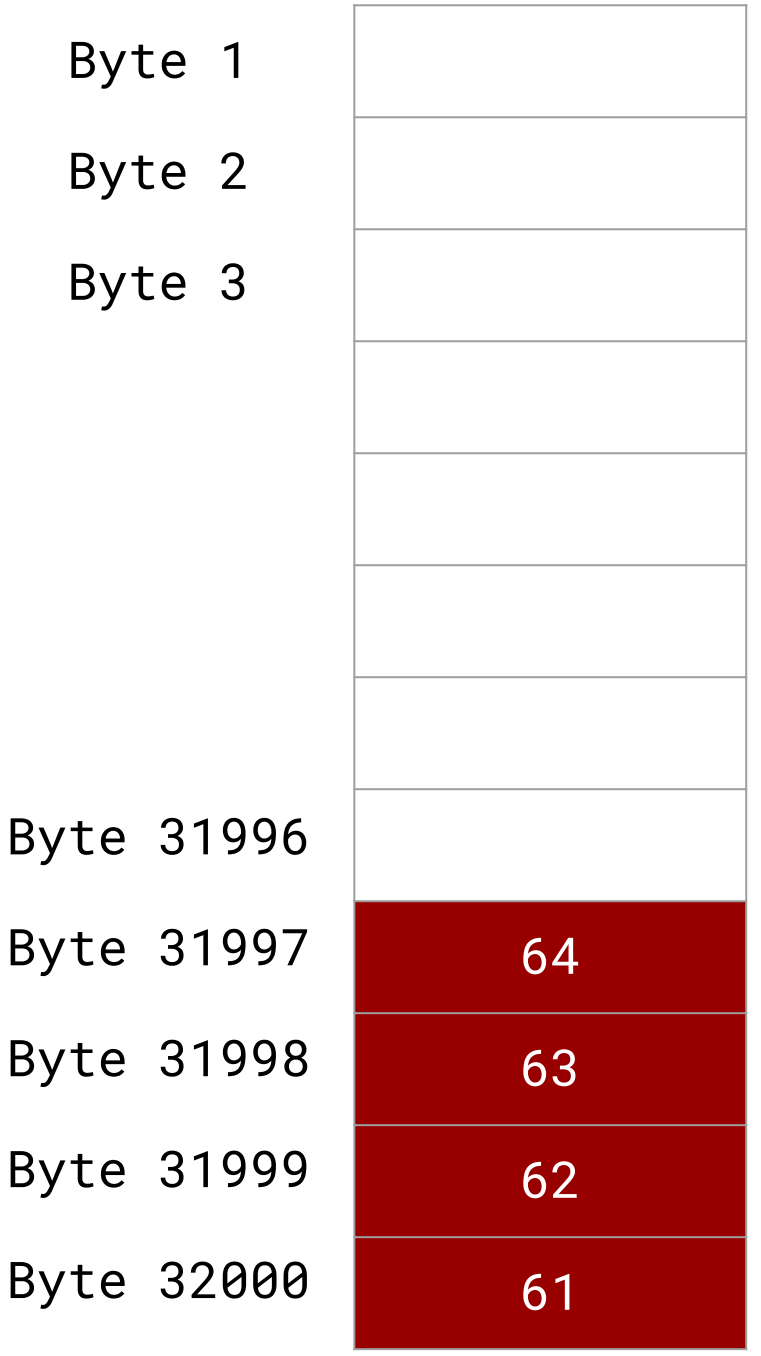
```
char a = 61; char b = 62; char c = 63;

...

 char d = 64;
```



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.

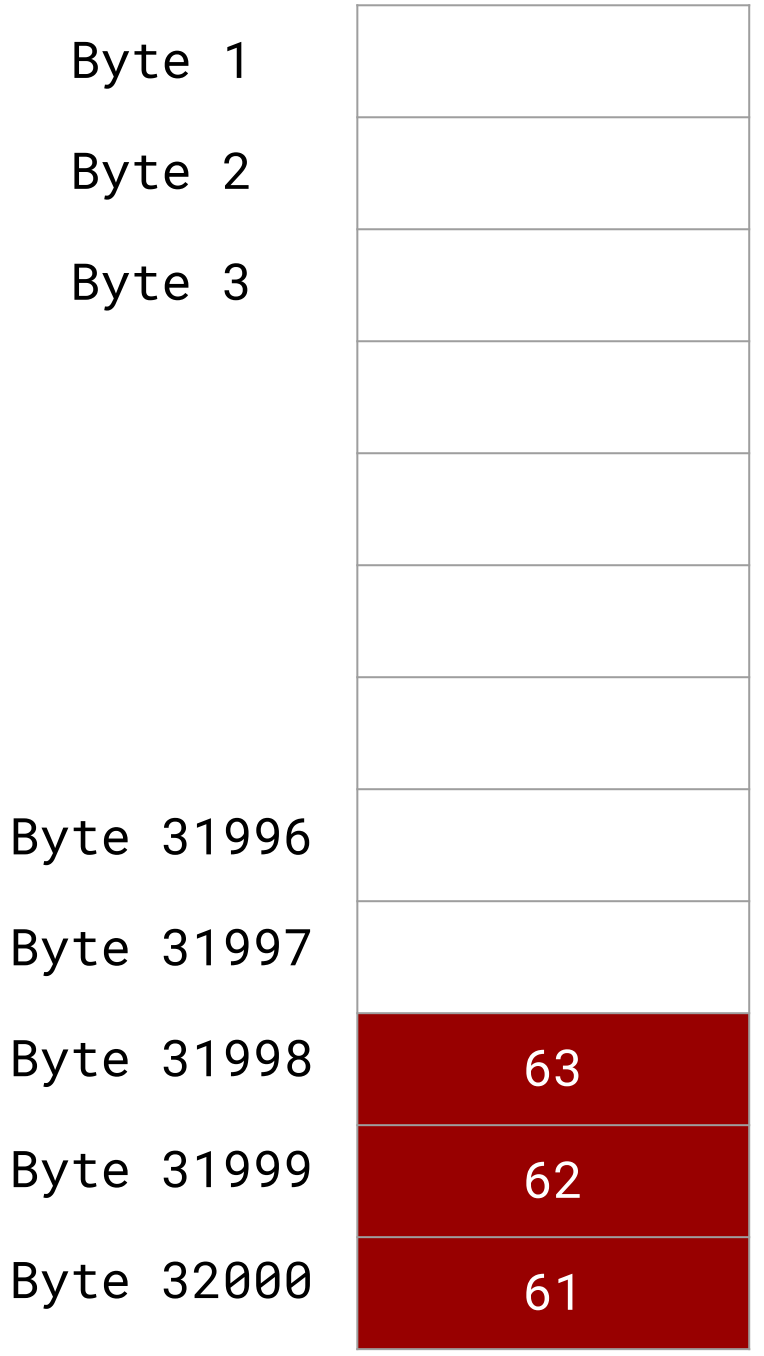
```
char a = 61; char b = 62; char c = 63;

...

 char d = 64;
```



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;
```



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;
```

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

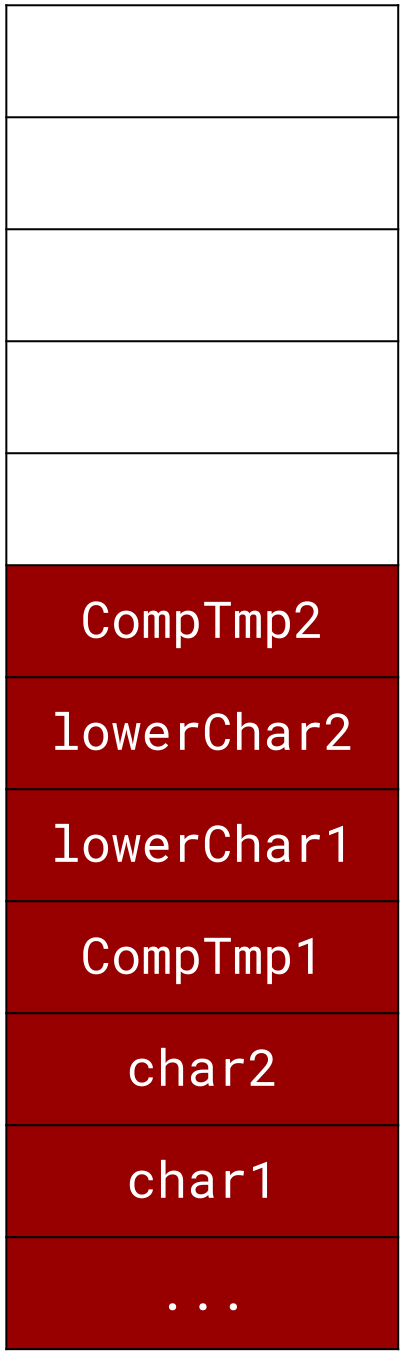
Τι αποθηκεύεται συνήθως στην στοίβα;

2: Ορίσματα που περνάμε στην συνάρτηση

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

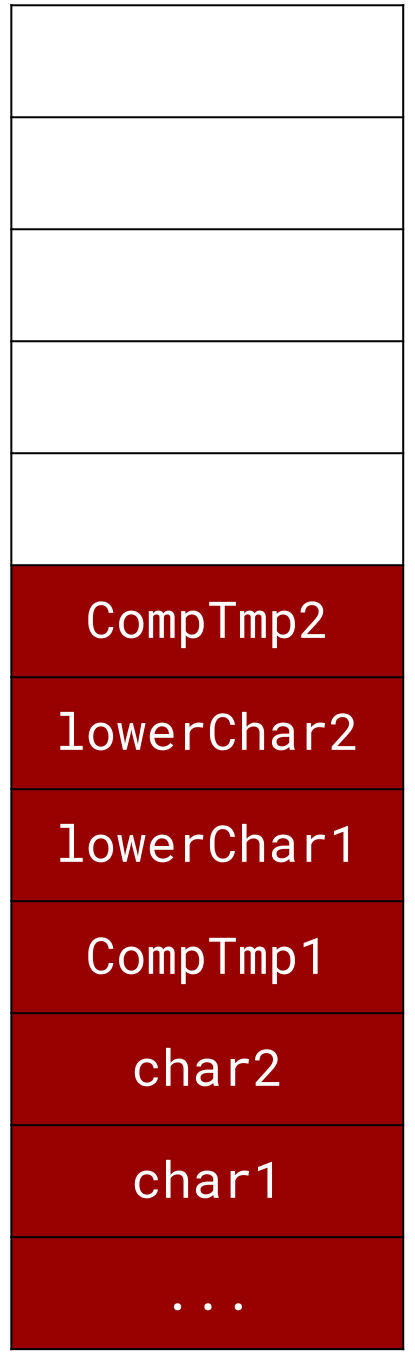
1: Τοπικές Μεταβλητές ορισμένες μέσα στην συνάρτηση

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

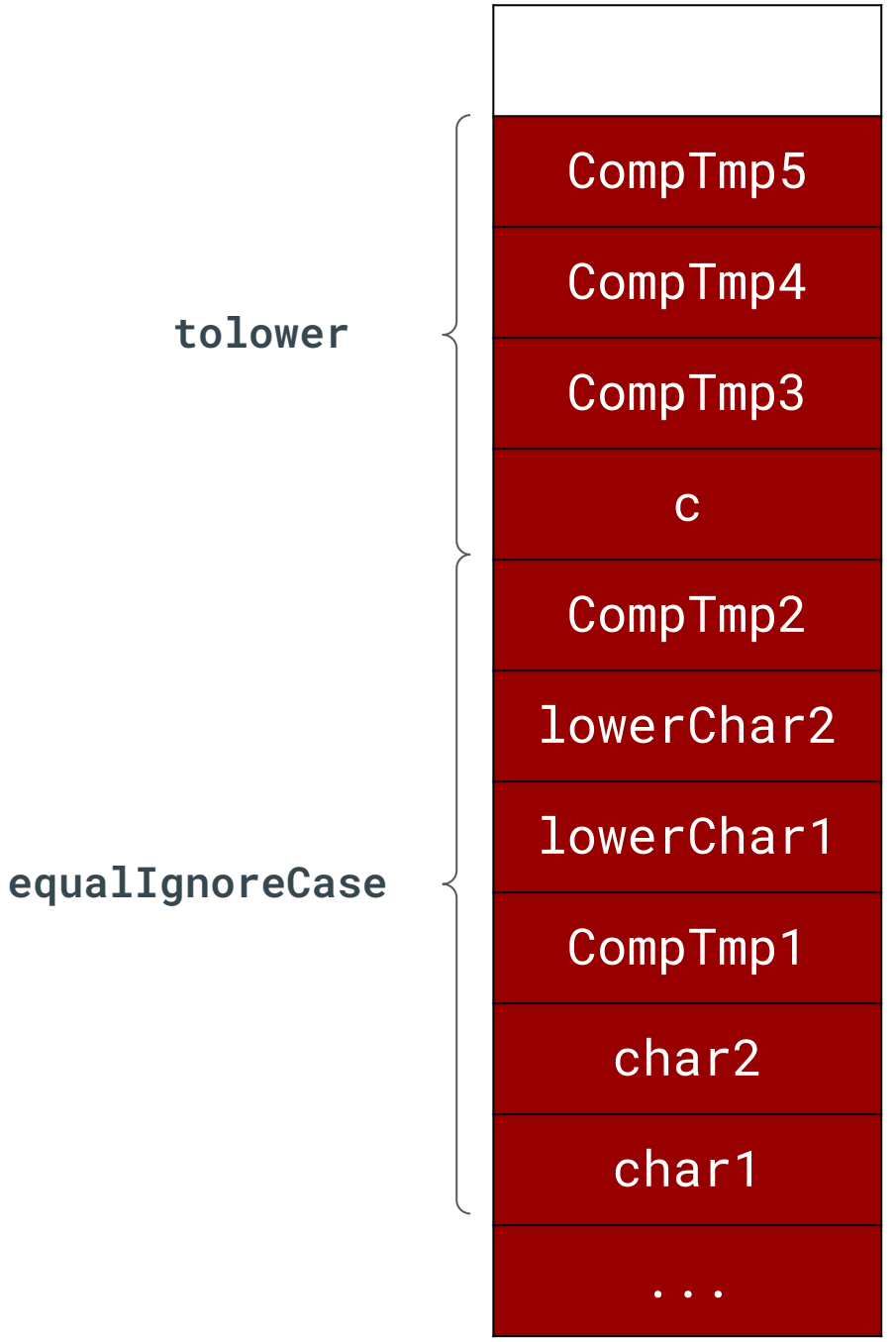
Τι αποθηκεύεται συνήθως στην στοίβα;

```
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;
}
```

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

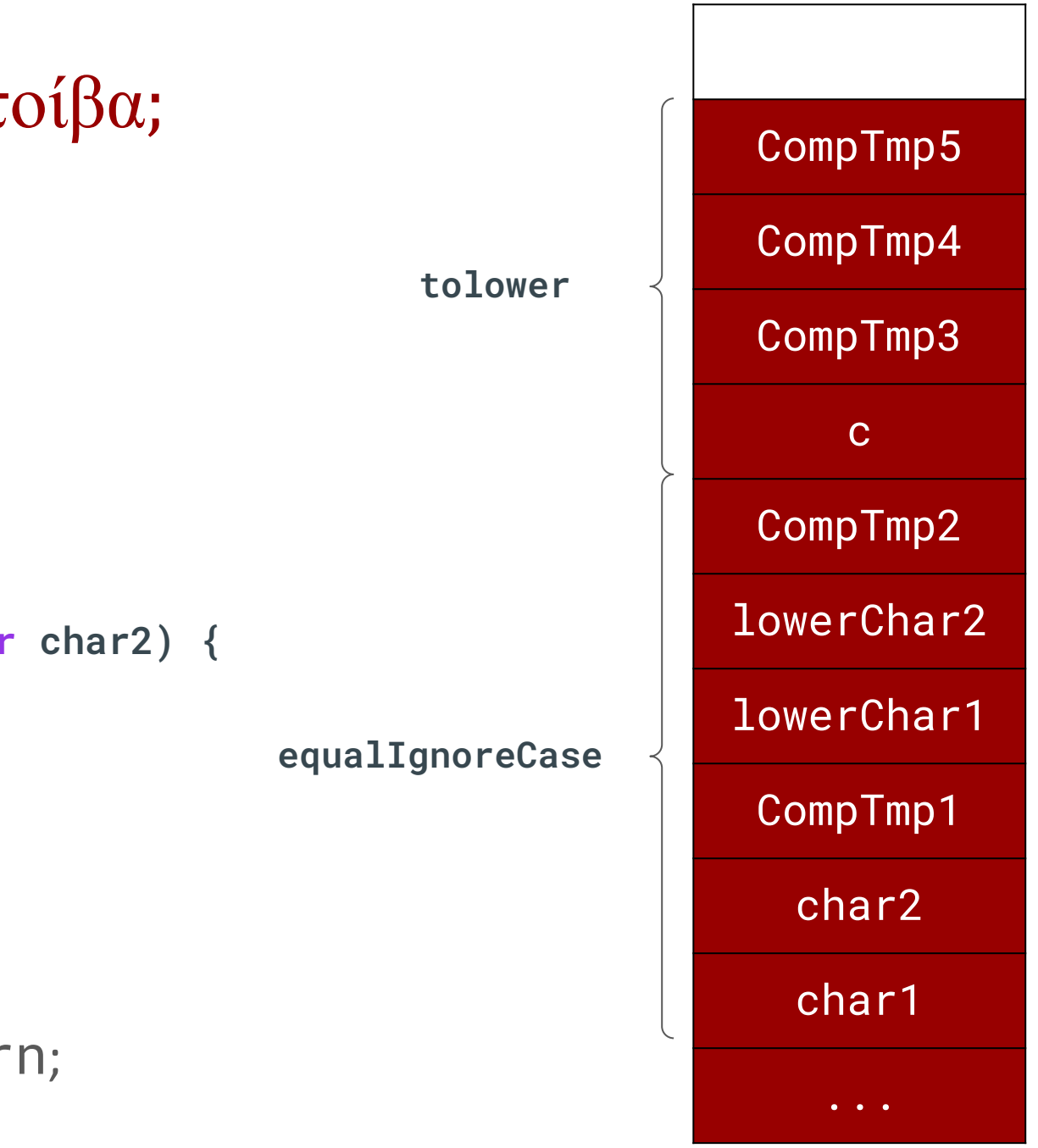
Τι αποθηκεύεται συνήθως στην στοίβα;

```
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;
}
```



Τι αποθηκεύεται συνήθως στην στοίβα;

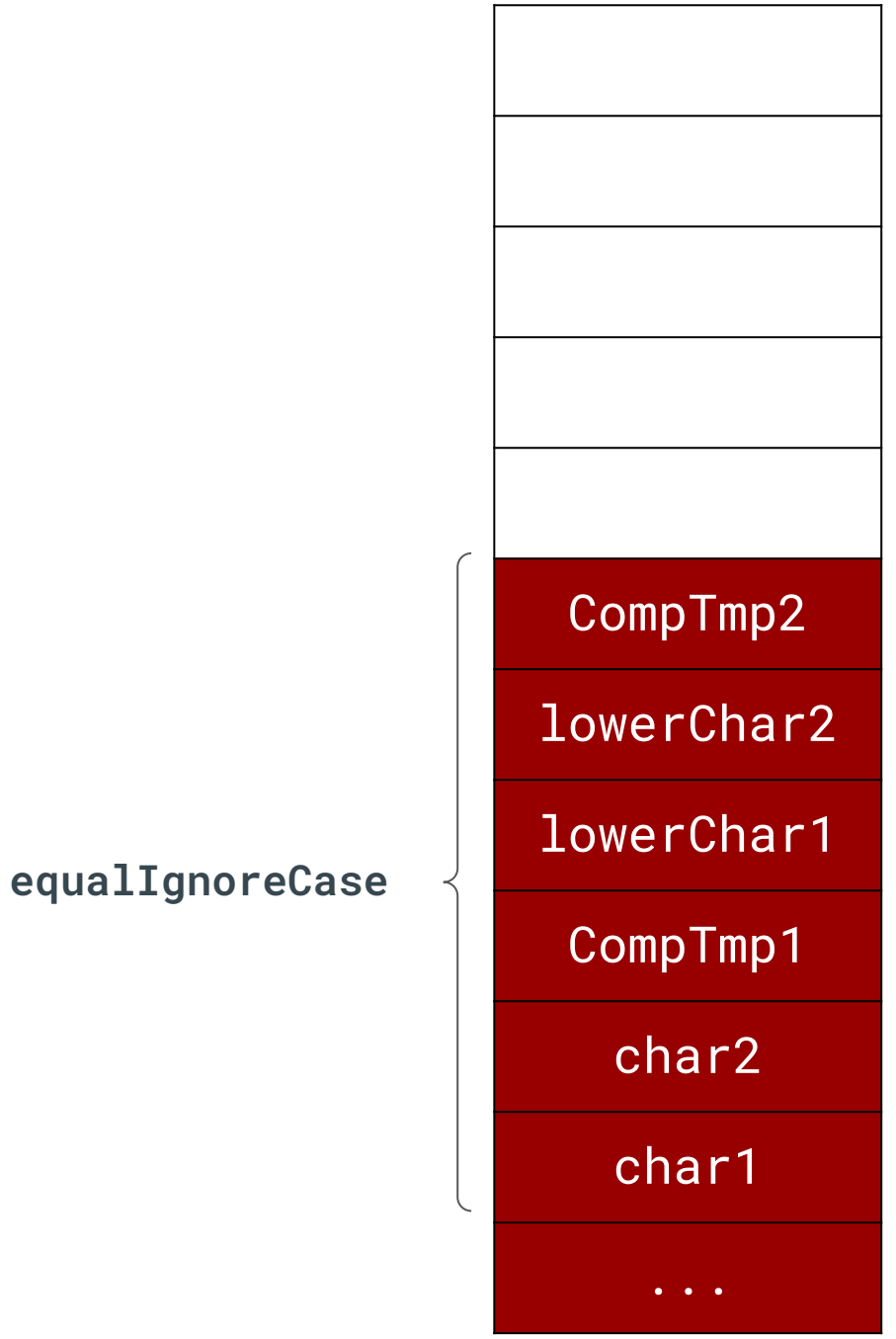
```
char tolower(char c) {
    if (c >= 'A' && c <= 'Z')
        return c + ('a' - 'A');
    return c;
}
int equalIgnoreCase(char char1, char char2) {
    char lowerChar1 = tolower(char1);
    char lowerChar2 = tolower(char2);
    return lowerChar1 == lowerChar2;
}
```



Τι θα συμβεί όταν εκτελεστεί η εντολή `return`;

Τι αποθηκεύεται συνήθως στην στοίβα;

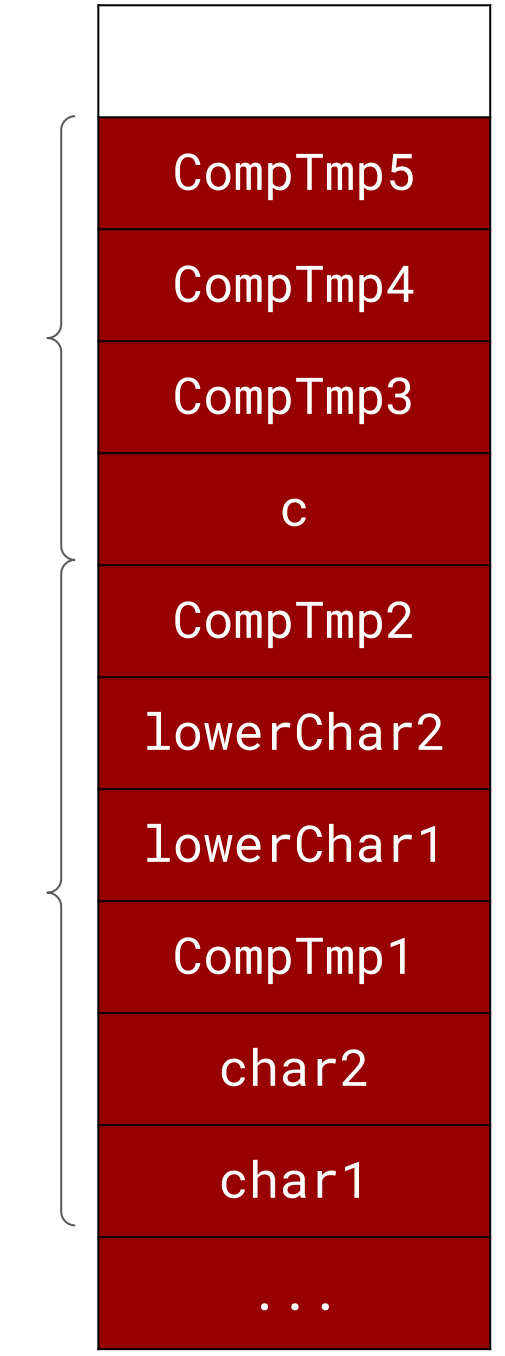
```
char tolower(char c) {
    if (c >= 'A' && c <= 'Z')
        return c + ('a' - 'A');
    return c;
}
int equalIgnoreCase(char char1, char char2) {
 char lowerChar1 = tolower(char1);
 char lowerChar2 = tolower(char2);
 return lowerChar1 == lowerChar2;
}
```



Τι θα συμβεί όταν εκτελεστεί η δεύτερη κλήση tolower;

Τι αποθηκεύεται συνήθως στην στοίβα;

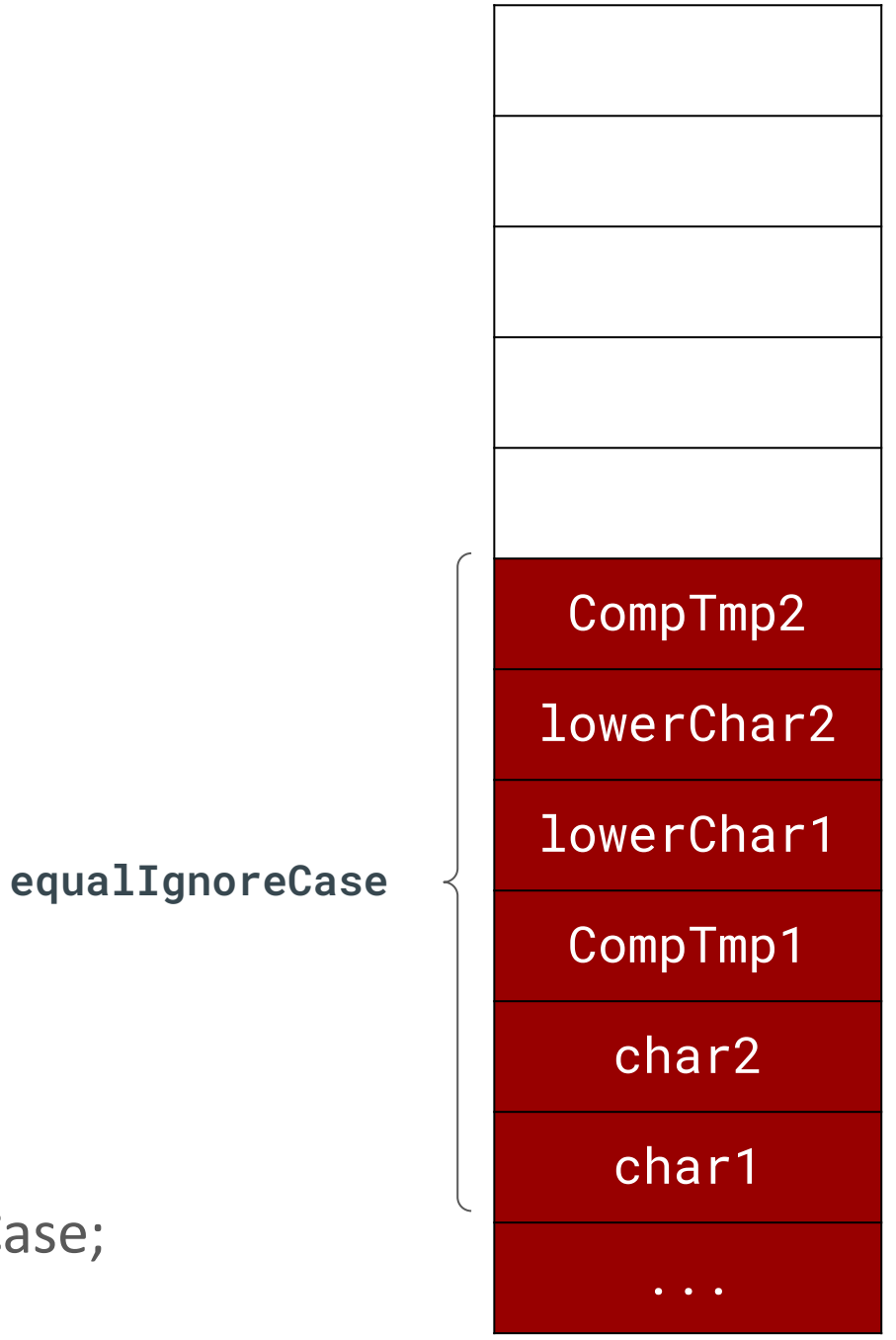
```
char tolower(char c) {
    if (c >= 'A' && c <= 'Z')
        return c + ('a' - 'A');
    return c;
}
int equalIgnoreCase(char char1, char char2) {
 char lowerChar1 = tolower(char1);
 char lowerChar2 = tolower(char2);
 return lowerChar1 == lowerChar2;
}
```



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;
}
```

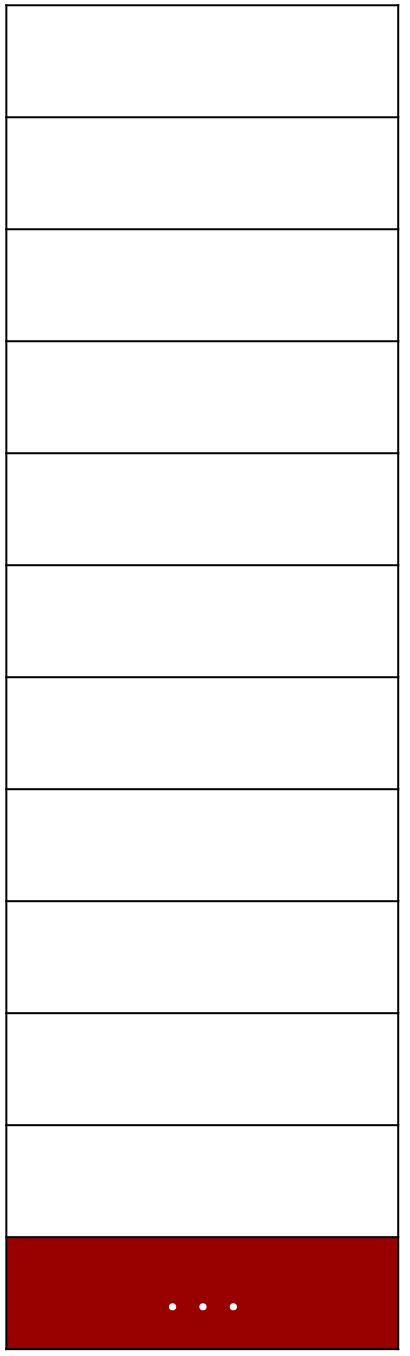


Τι θα συμβεί όταν εκτελεστεί η `return` της equalIgnoreCase;

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;
}
```

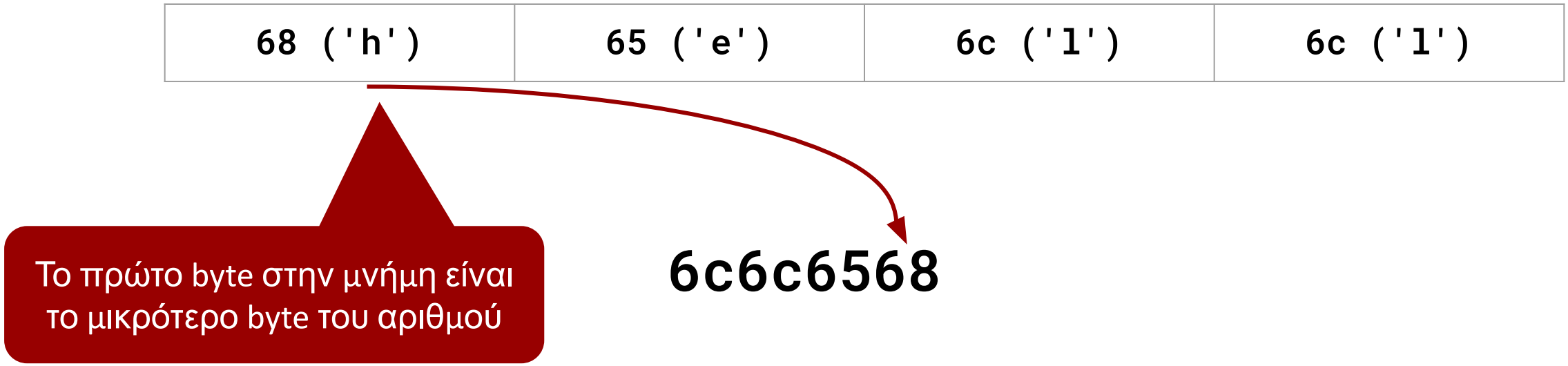
Τι θα συμβεί όταν εκτελεστεί η `return` της equalIgnoreCase;



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).



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;
}
```

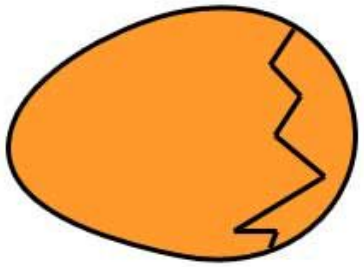
Παράδειγμα Endianness

Τι θα τυπώσει το παρακάτω:

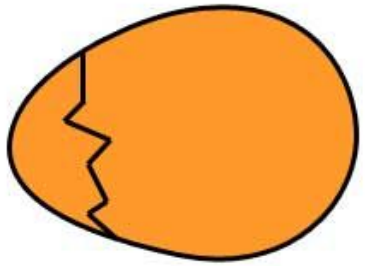
```
#include <stdio.h>
int main() {
  int x = 0x42434445;
  char * bytes = (char*)&x;
  int i;
  for(i = 0; i < sizeof(int) / sizeof(char); i++)
   printf("%02x\n", bytes[i]);
  return 0;
}
```

```
$ ./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/

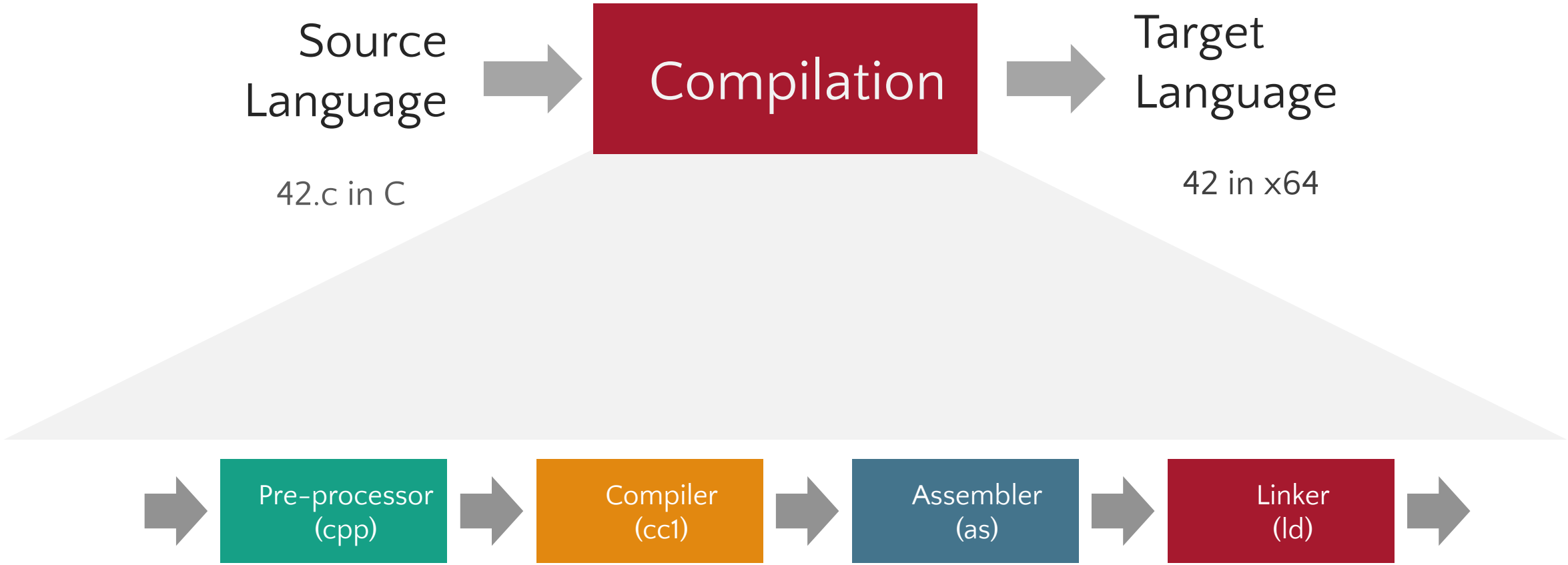
Compilation Workflow & Execution Semantics

To answer

Is this program safe?

We must first know

What will executing it do?


Source Language
42.c in C
Compilation
Target Language
42 in x64
Pre-processor (cpp)
Compiler (cc1)
Assembler (as)
Linker (ld)

- Preprocessor
- Compiler
- Assembler
- Linker

```
$ gcc -E
```

```c
#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
- Compiler
- Assembler
- Linker

```
$ gcc -S
```

```
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)

- Preprocessor
- Compiler
- Assembler
- Linker

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

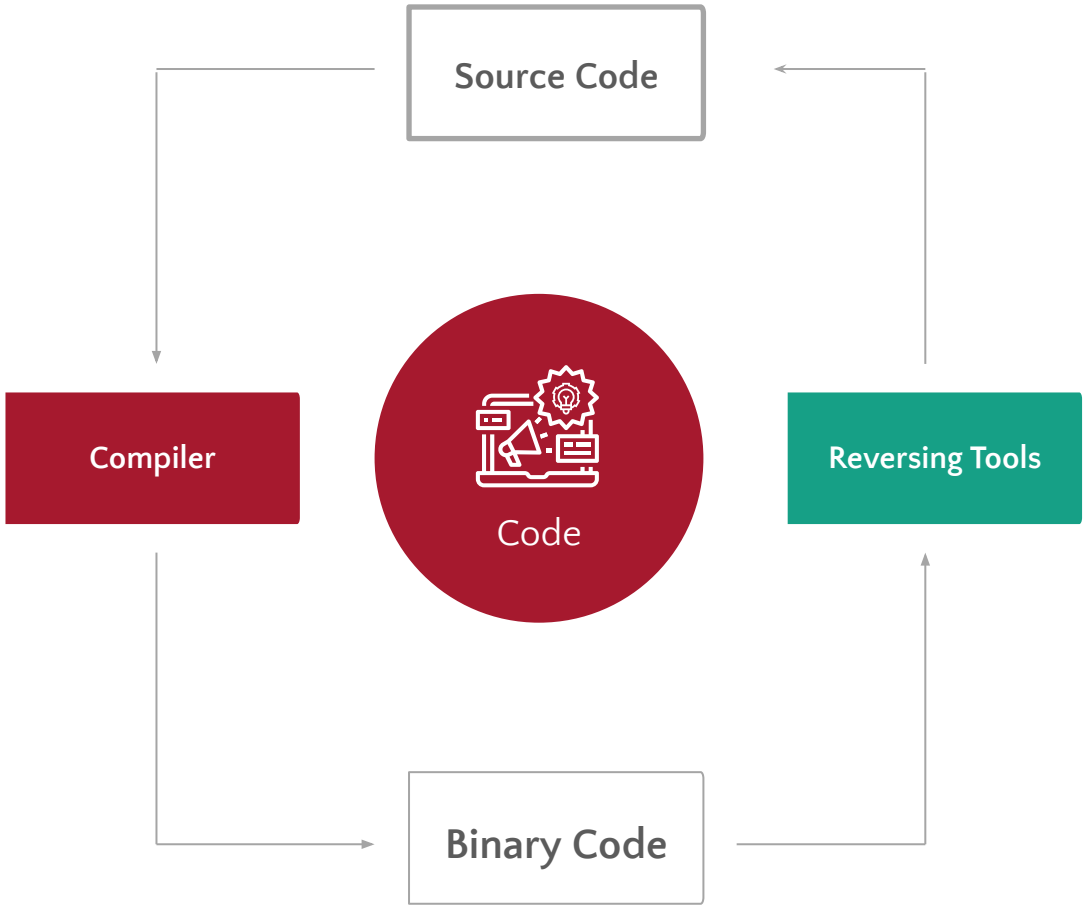
```
gcc -v hello.c
 ...
/usr/lib/gcc/x86_64-linux-gnu/10/collect2 -plugin
/usr/lib/gcc/x86_64-linux-gnu/10/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper
-plugin-opt=-fresolution=/tmp/ccLvq7Mj.res -plugin-opt=-pass-through=-lgcc
-plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc
-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

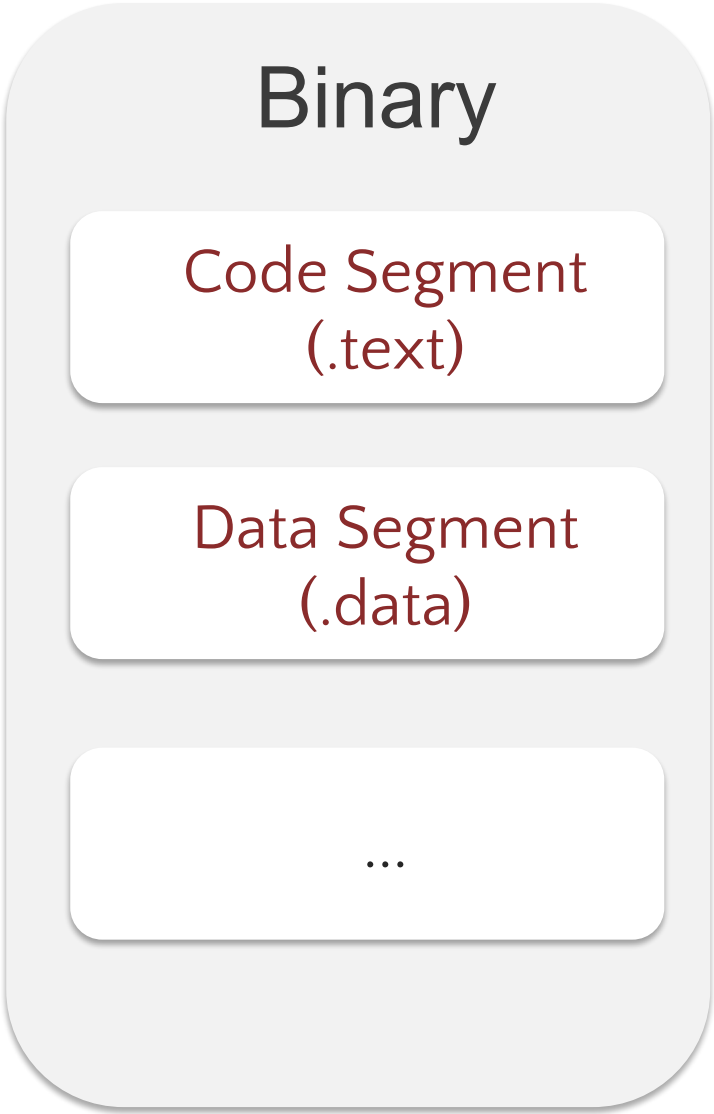
Turn source code into executable code using programmer-defined abstractions as a guide translation.



Reversing

Best case effort to recover source code from binary code.

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

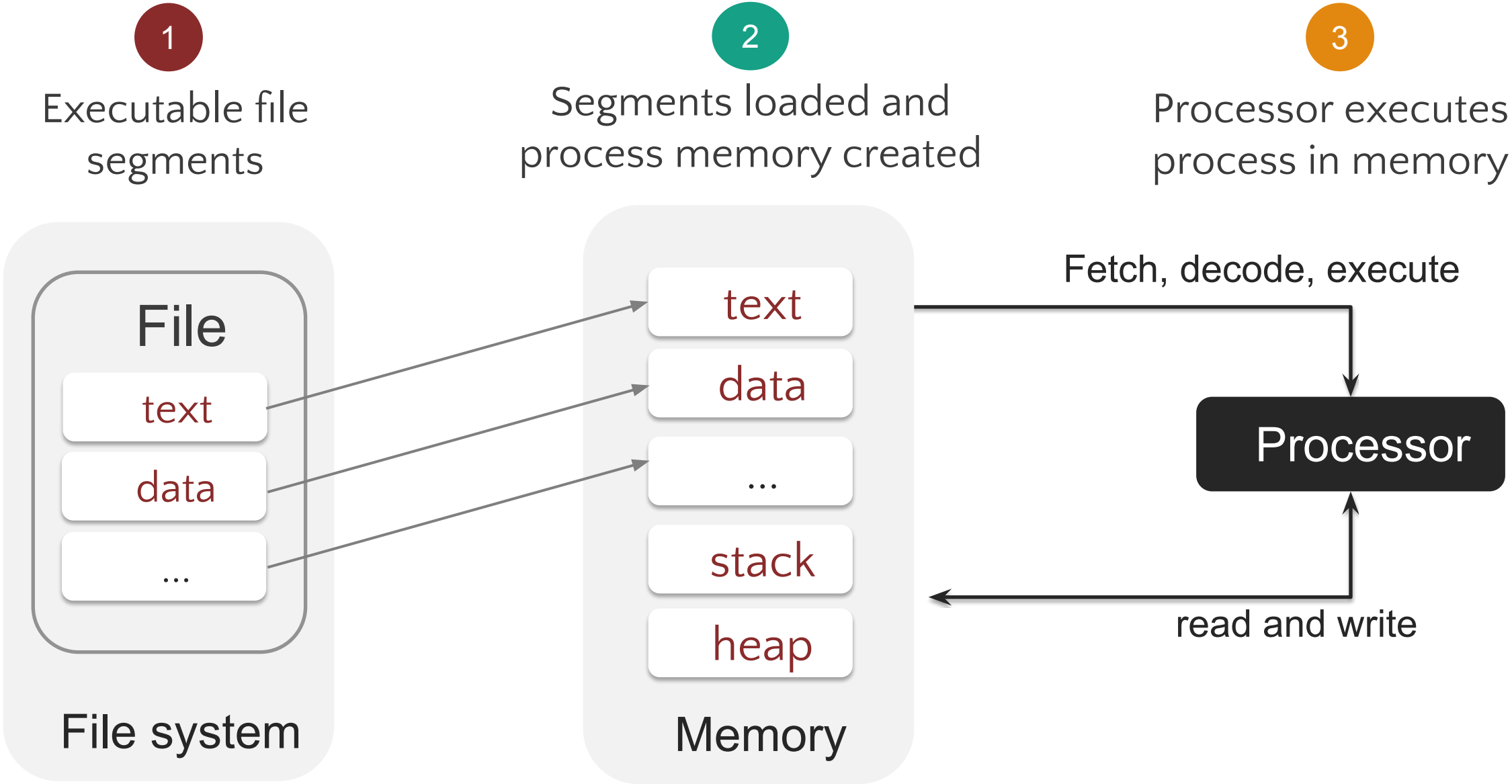


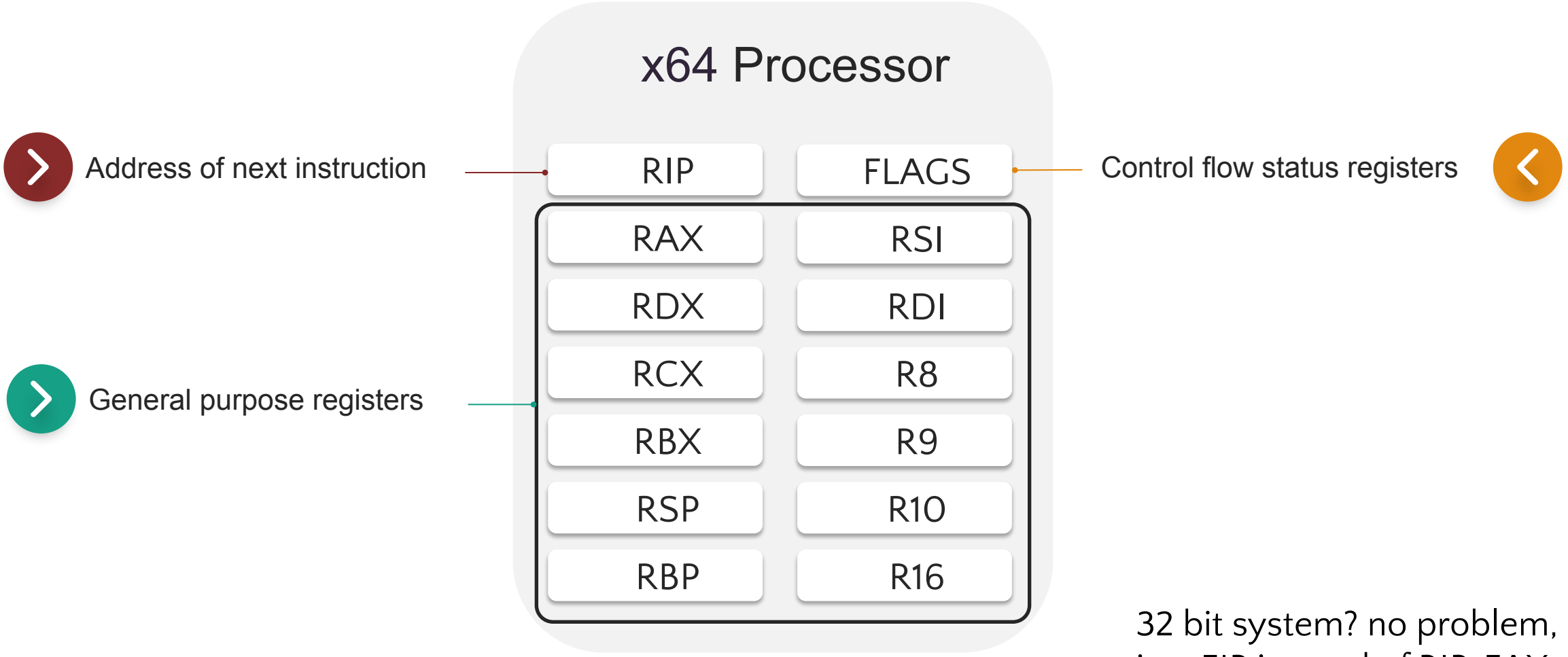
The final **executable binary**

- 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
```

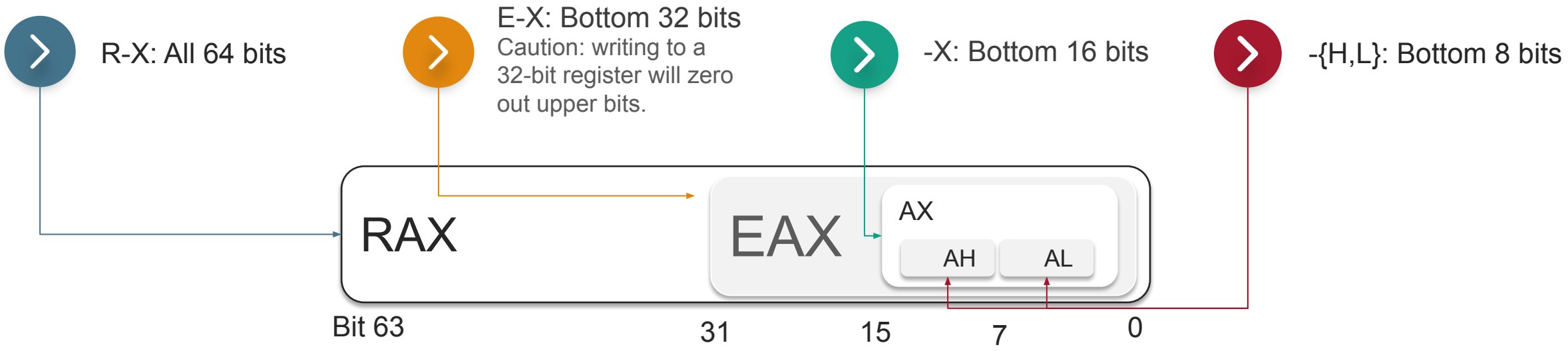
BASIC EXECUTION MODEL


1
Executable file segments
2
Segments loaded and process memory created
3
Processor executes process in memory
File
text
data
...
File system
text
data
...
stack
heap
Memory
Fetch, decode, execute
Processor
read and write



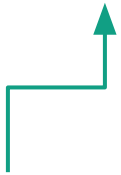
32 bit system? no problem, just EIP instead of RIP, EAX instead of RAX, etc

Parts of a register can be addressed directly



Basic Ops and AT&T Vs. Intel Syntax

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



- 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

Register	Value
rax	0x3
rdx	0x0
rbx	0x5

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

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

Memory Loads

What do these instructions do?

```
1.  movb (%rax), dl

1.  mov (%rax), edx
```

Register	Value
rax	0x3
rdx	0x0
rbx	0x5

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

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

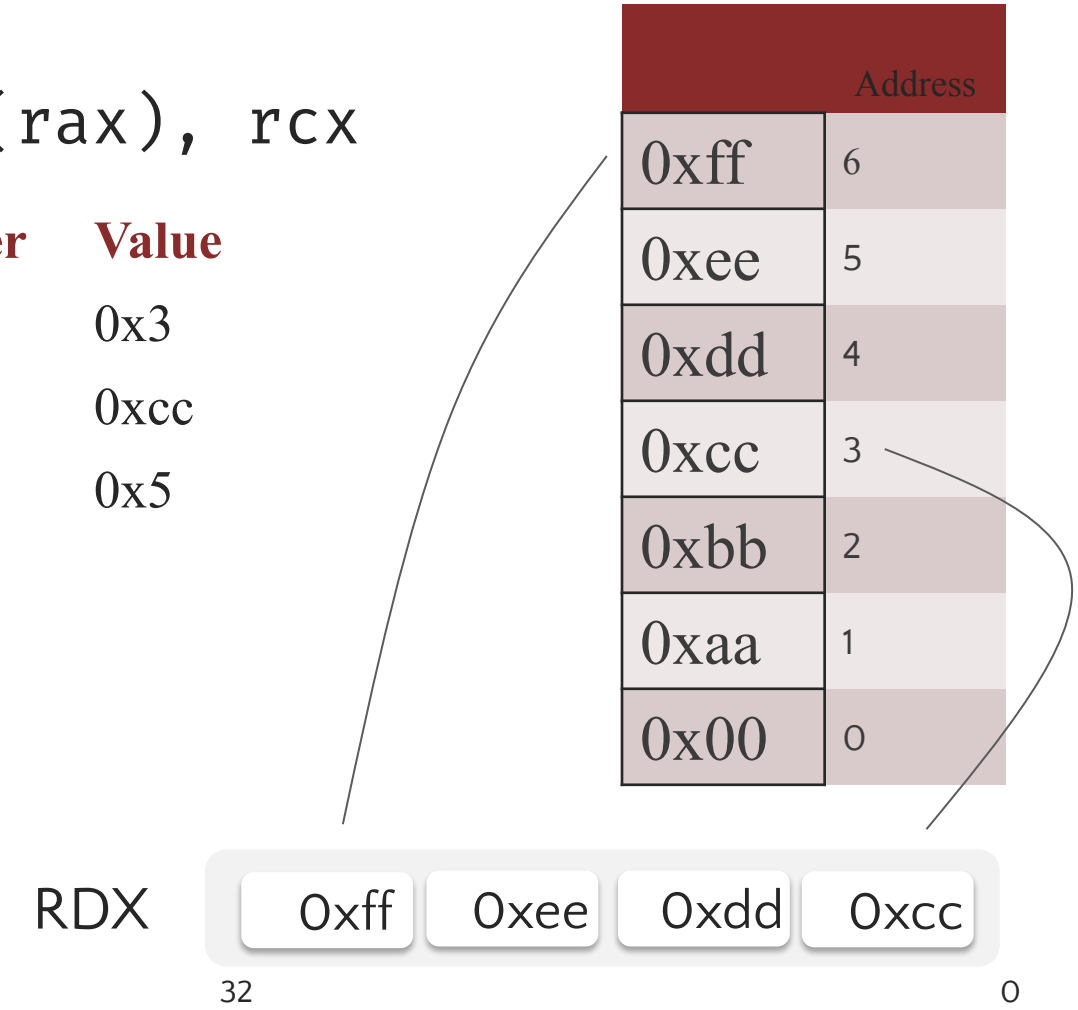
EXAMPLE

We'll assume x86/x86_64 from now on:

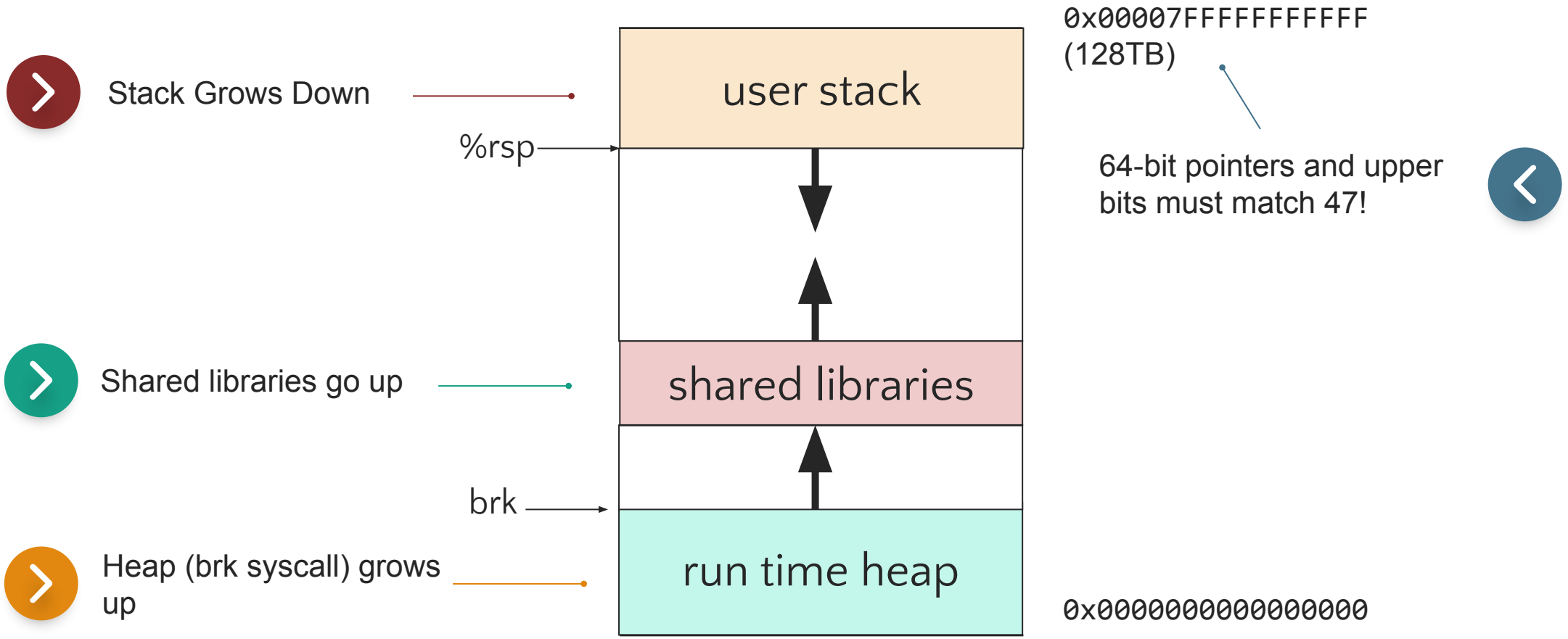
- Address *a* goes into register bits 0–7
- Address *a+1* in the 8–15
- And so on

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

Register	Value
rax	0x3
rdx	0xcc
rbx	0x5



Process Memory Organization

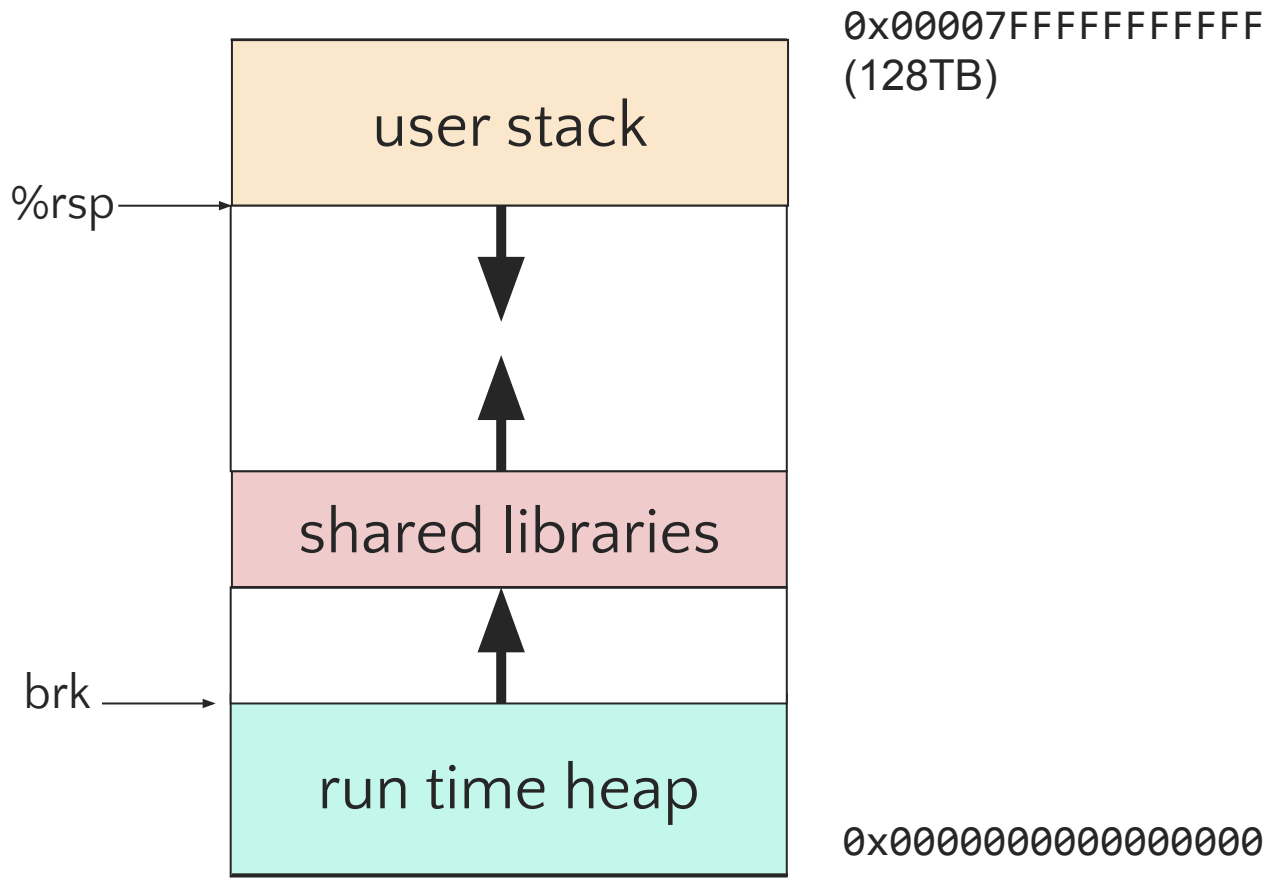

Stack Grows Down
%rsp
user stack
Shared libraries go up
shared libraries
brk
Heap (brk syscall) grows up
run time heap
0x00007FFFFFFFFFFF
(128TB)
64-bit pointers and upper bits must match 47!
0x0000000000000000

On the stack

- Local variables
- Lifetime:
  stack frame

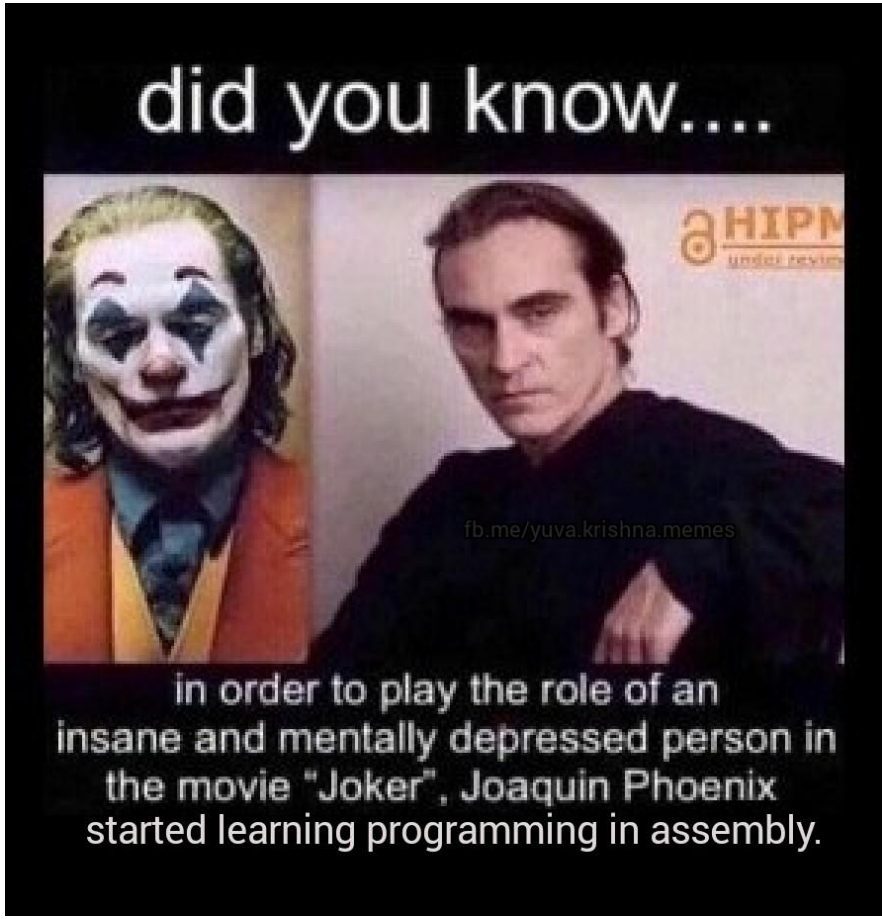
On the heap

- 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

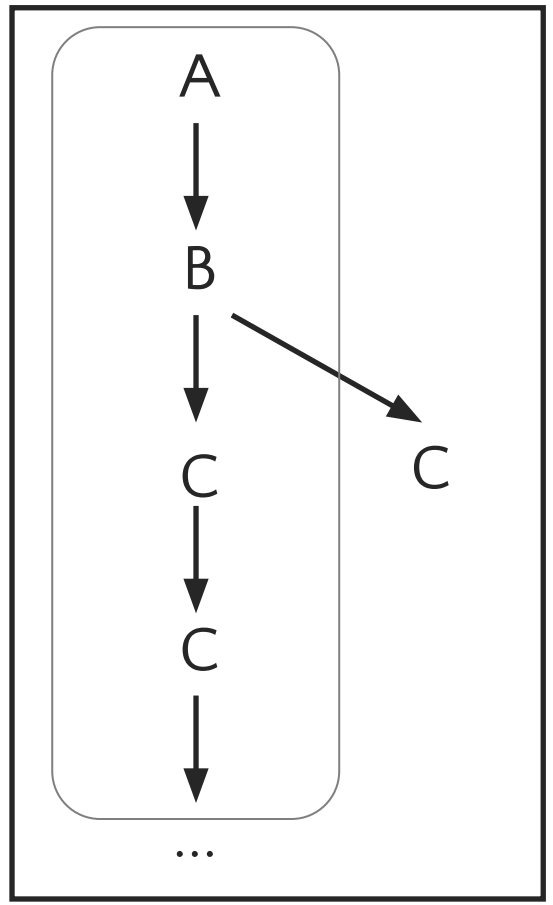
Recall the basic stack semantics of function calls

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

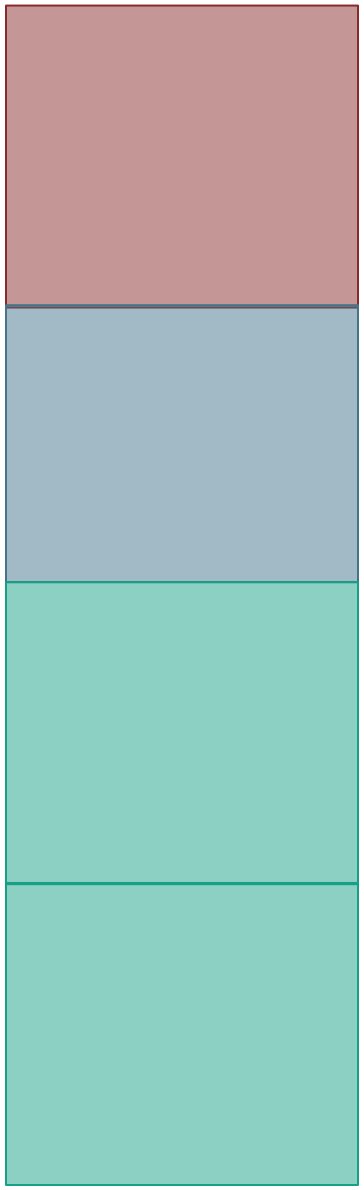
```
B(...)
{
   ...
   C()
   ...
   C()
}
```

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

Function Call Chain



The Stack

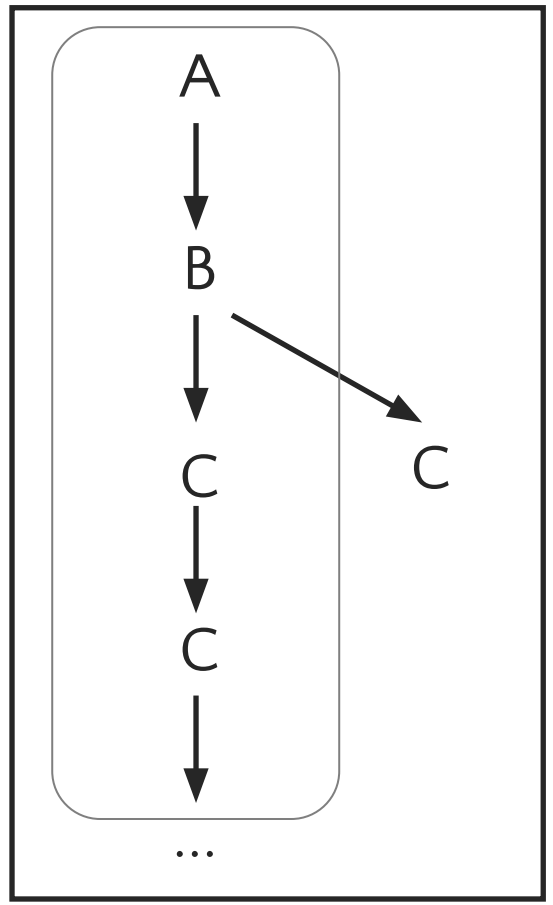


Frame for
- locals
- pushing parameters
- temporary space

Call
"***pushes***"
new frame

When ***callee***
returns it
"***pops***"
its frame

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
```

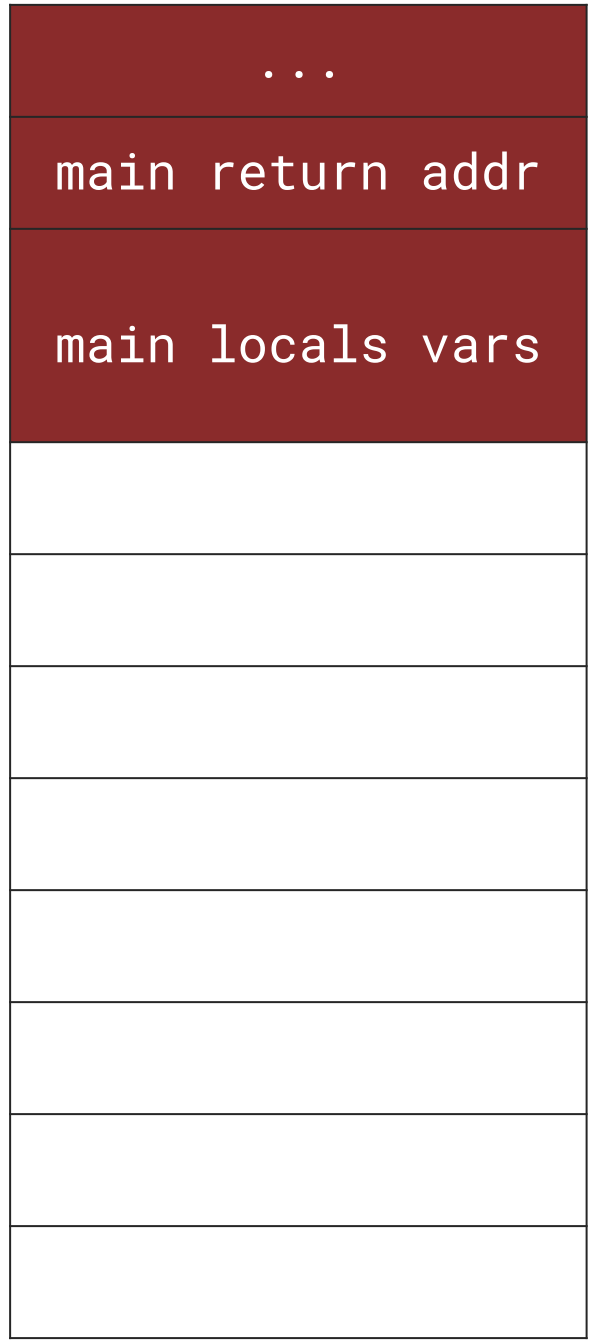
next_addr = mem[sp]; sp += sizeof(next_addr)

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>
```

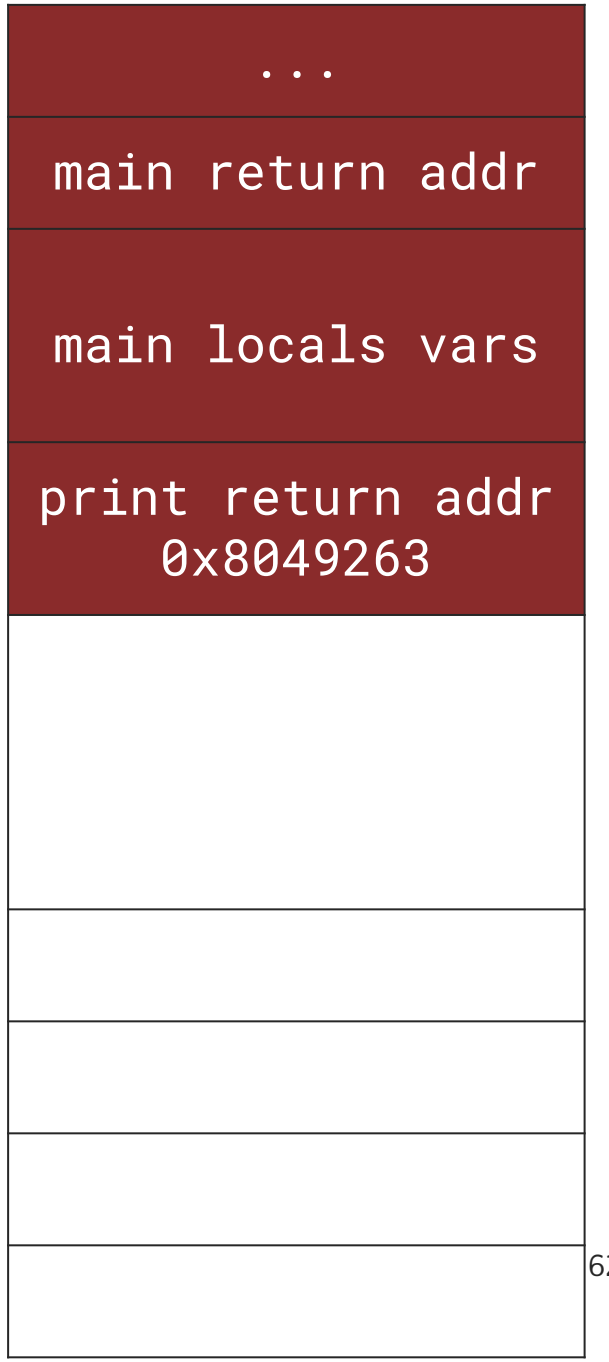
esp = 0xffffd640; eip = 0x804925e



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
```

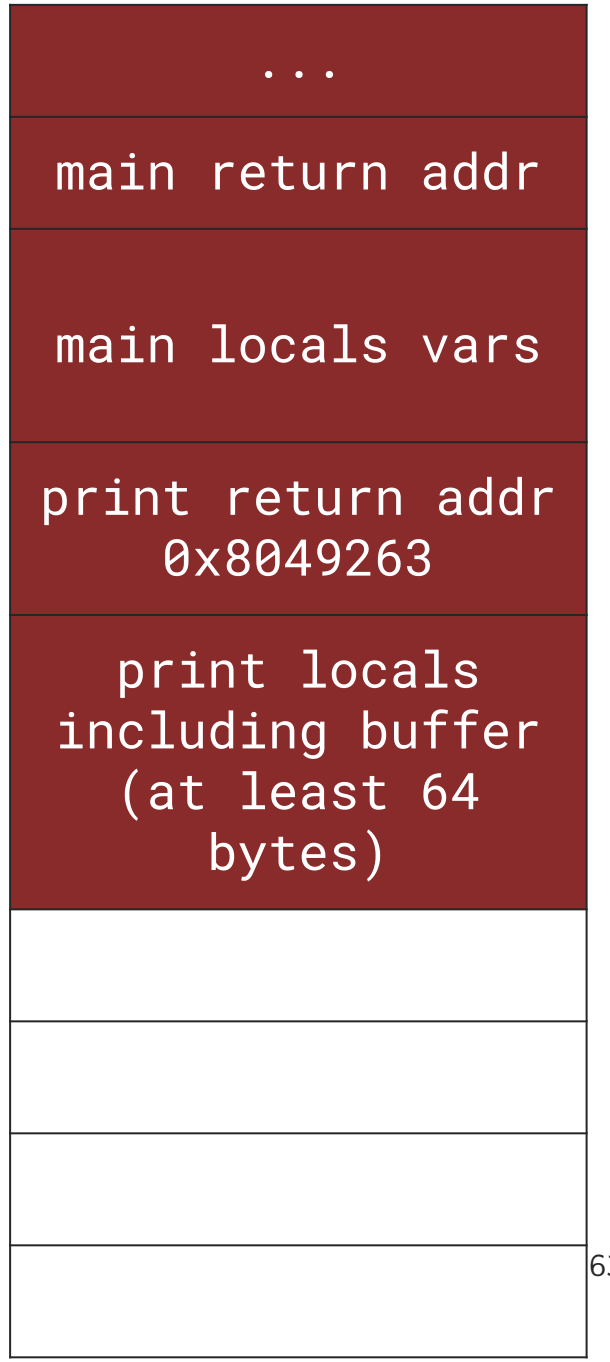


A Toy Example

```
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>
```

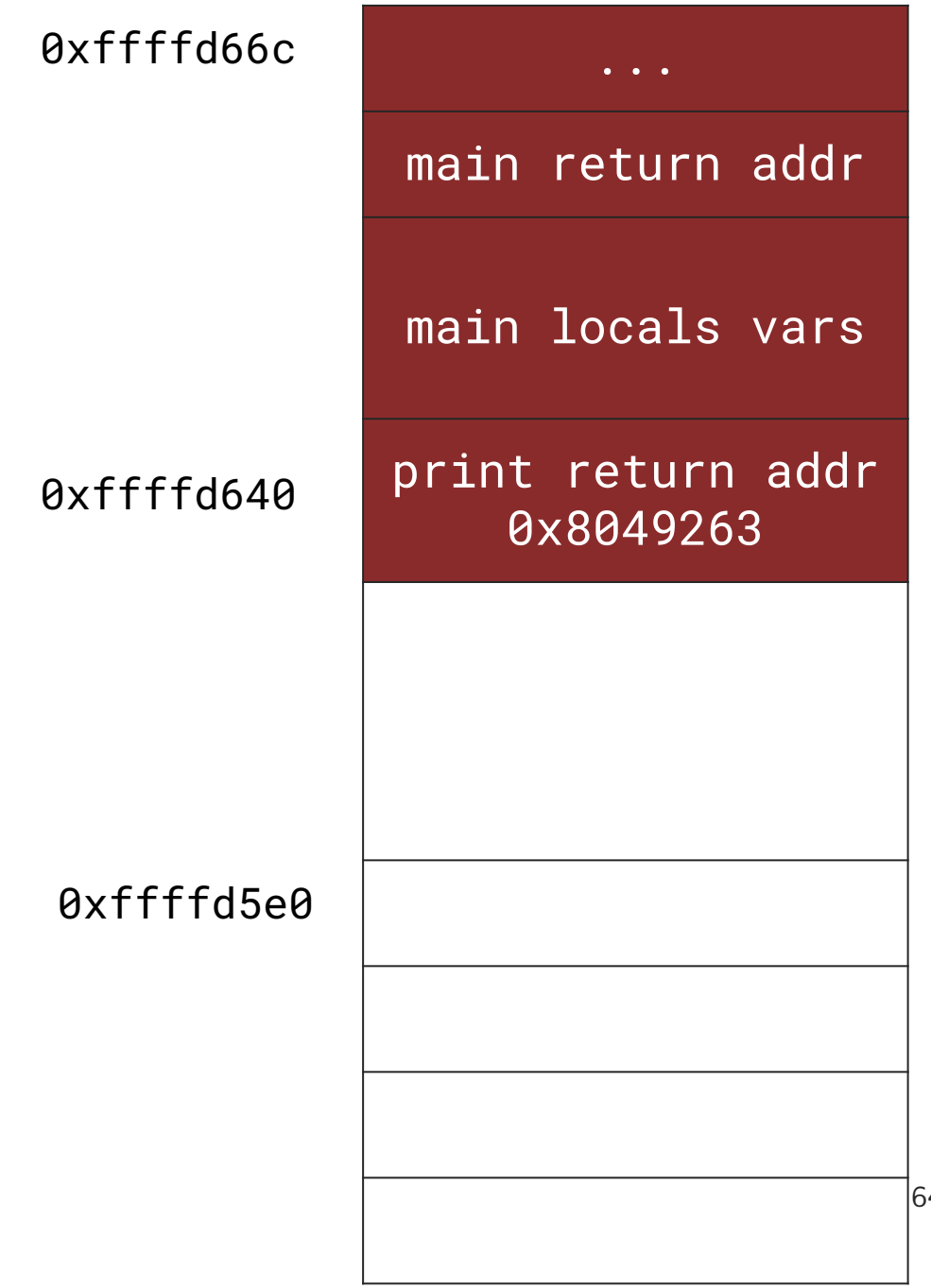
esp = 0xffffd5e0; eip = 0x8049202



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
```

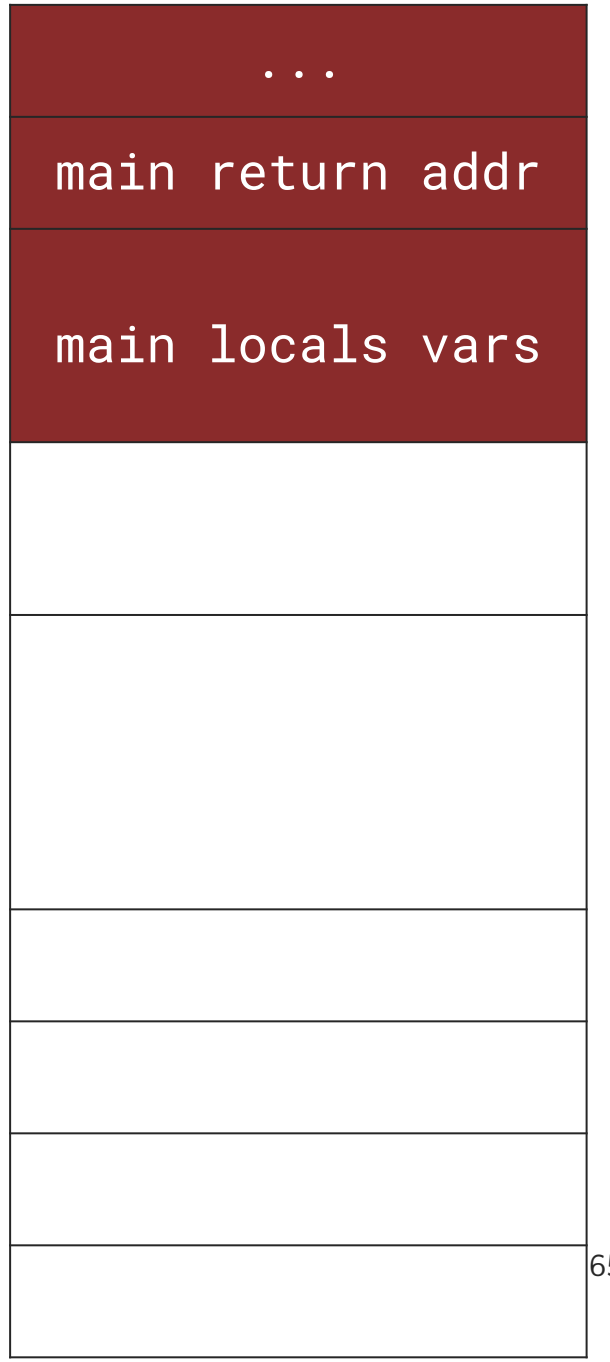


A Toy Example

```
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                add $0x10,%esp
```

esp = 0xffffd640; eip = 0x8049263



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

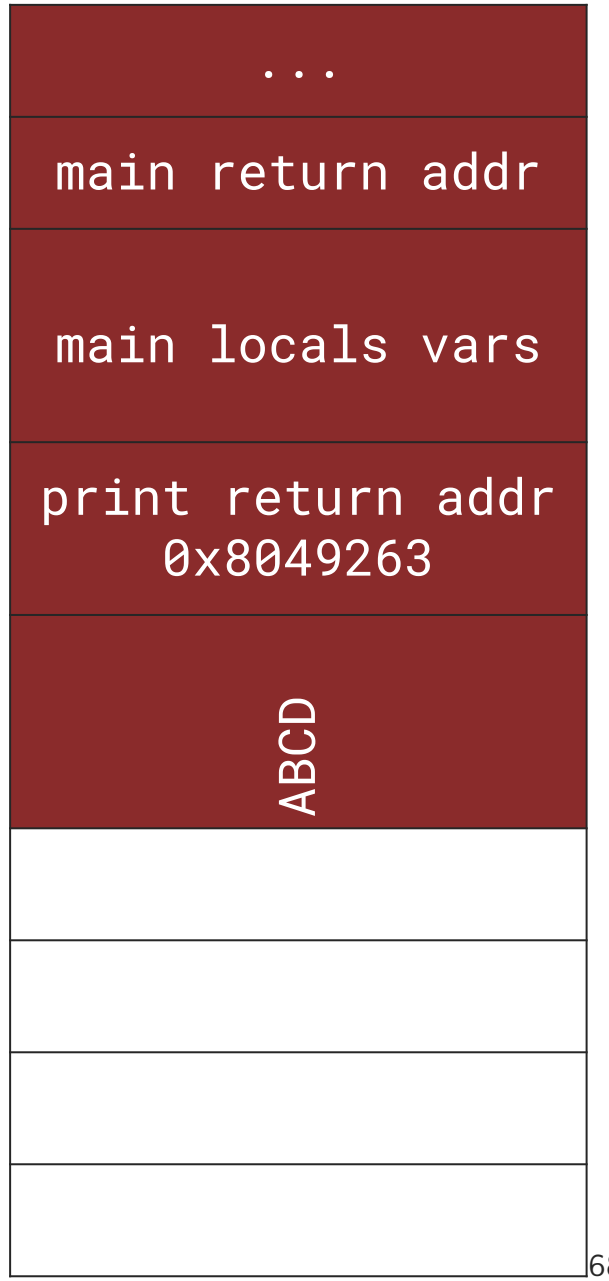
A Toy Example

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;
}
```

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

esp = 0xffffd5e0; eip = 0x8049202



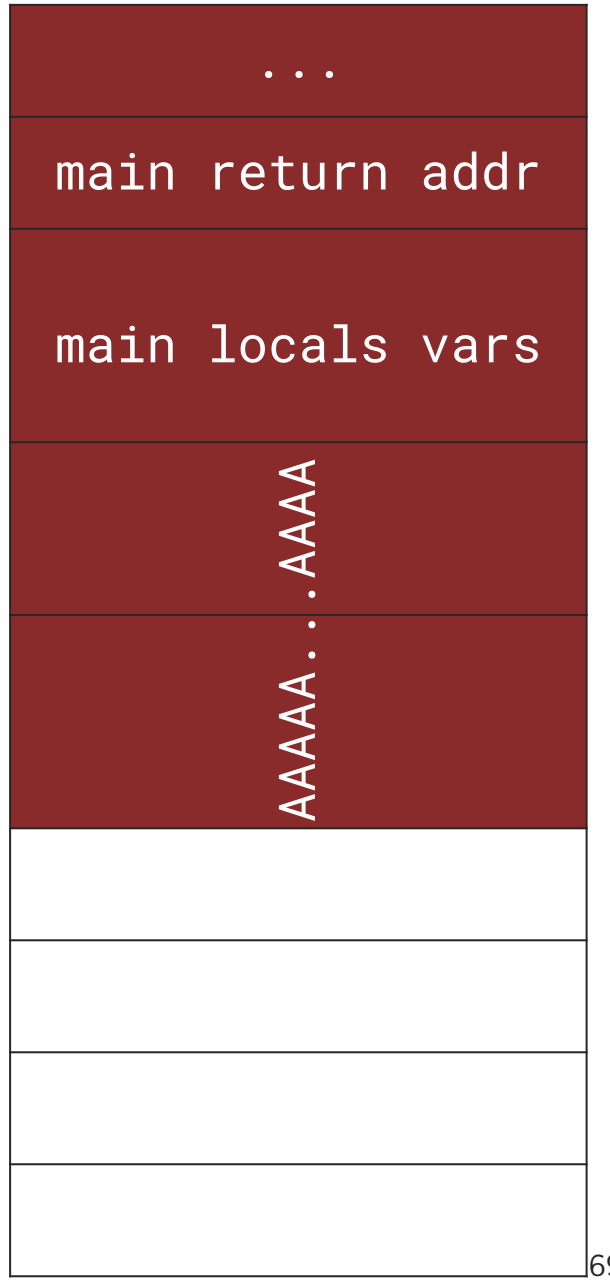
A Toy Example

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;
}
```

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

esp = 0xffffd5e0; eip = 0x8049202



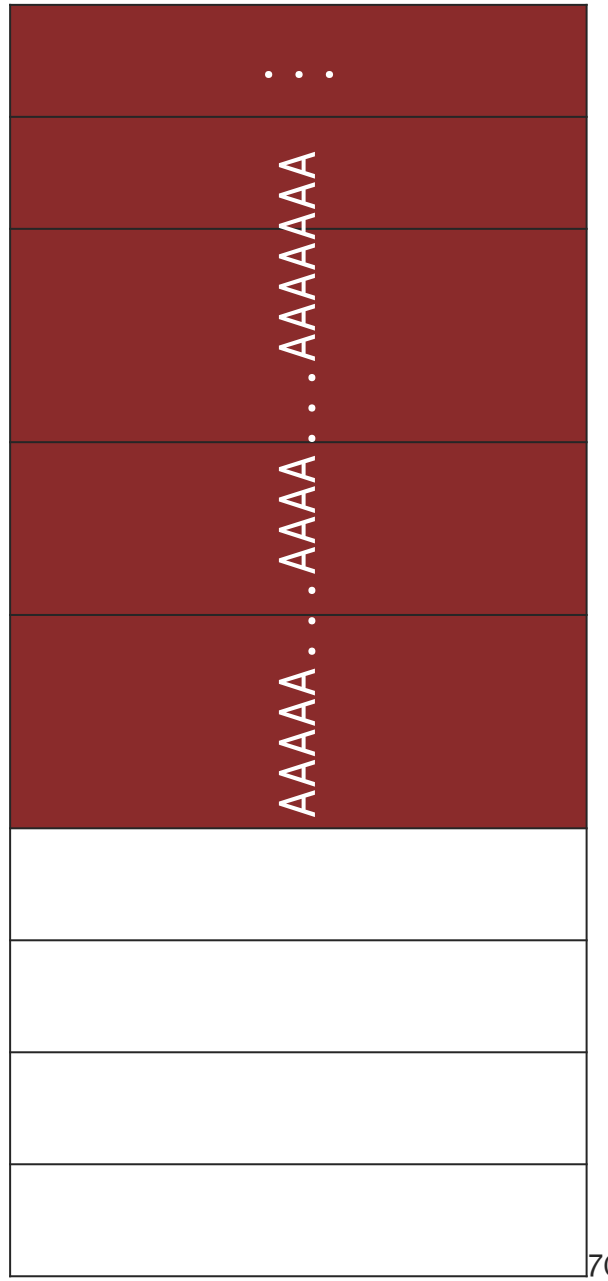
A Toy Example

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;
}
```

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

esp = 0xffffd5e0; eip = 0x8049202



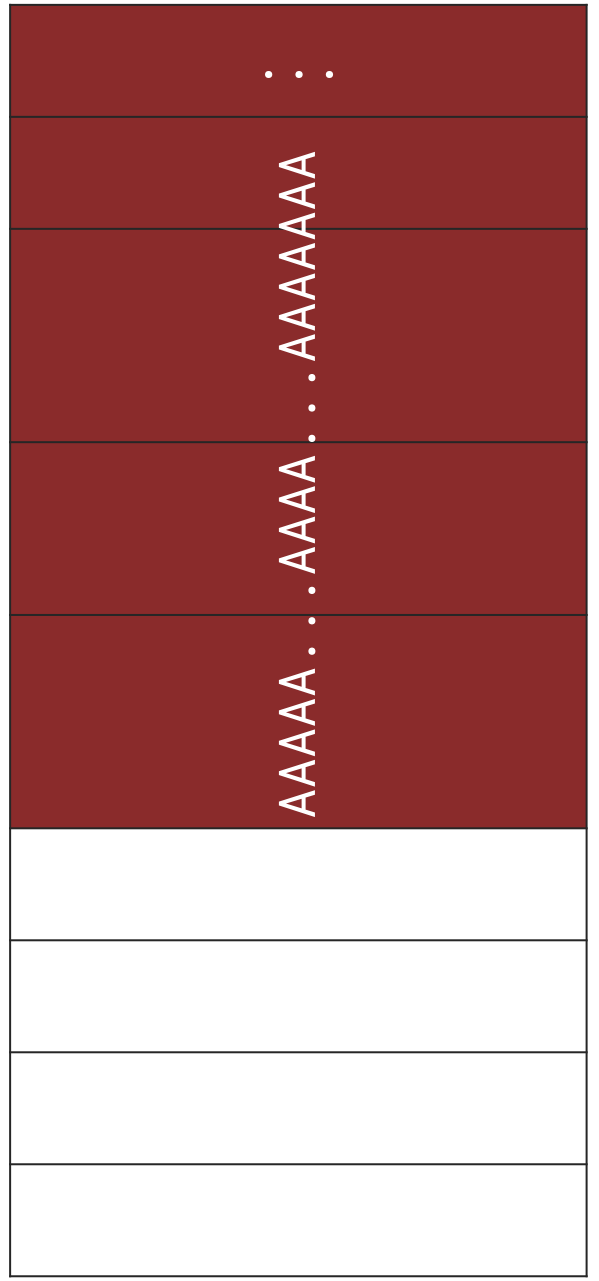
A Toy Example

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
```

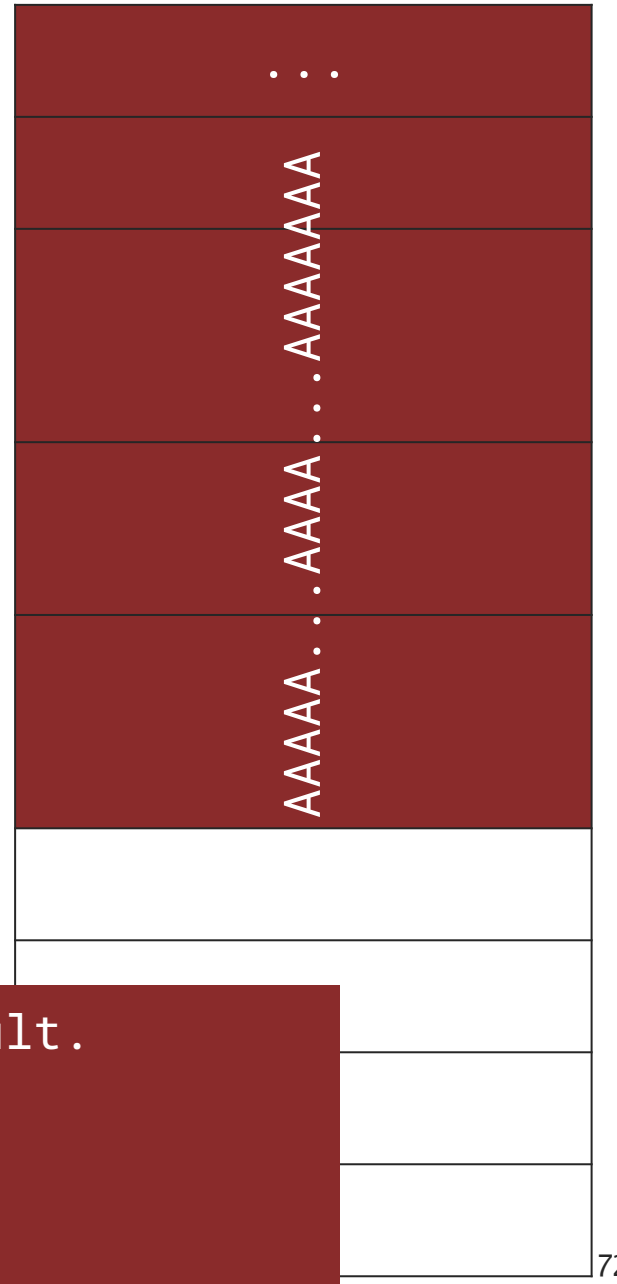
esp = 0xffffd63c; eip = 0x804920f



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;
}
```

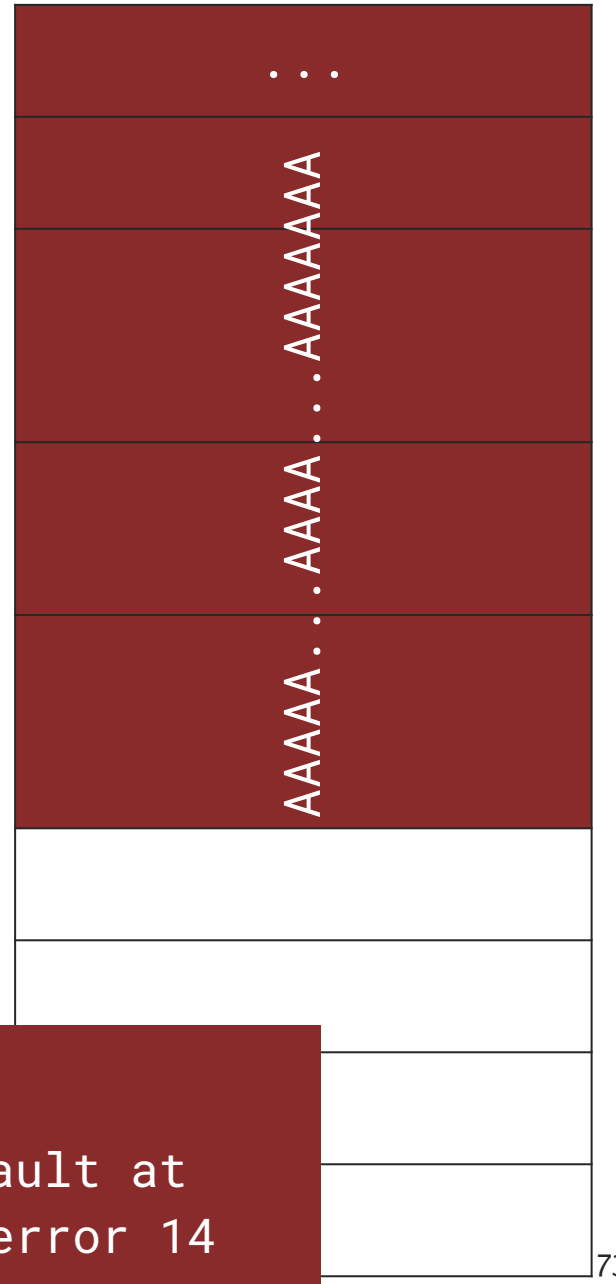



```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r eip
eip            0x41414141       0x41414141
```

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;
}
```




```
$ 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!

Shellcode Injection

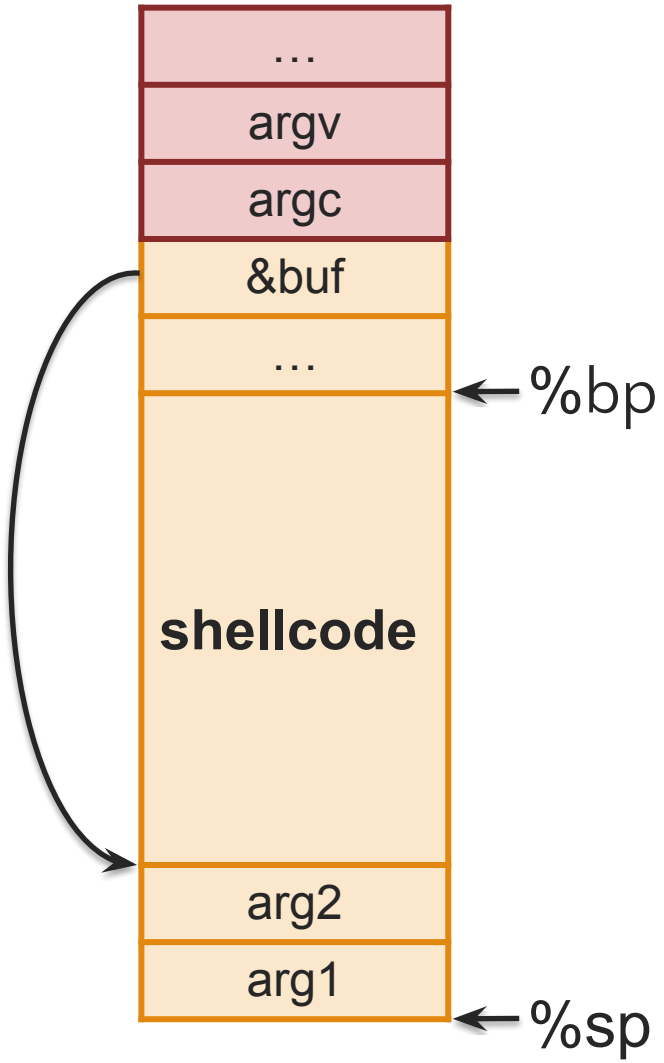
Shellcode

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*!

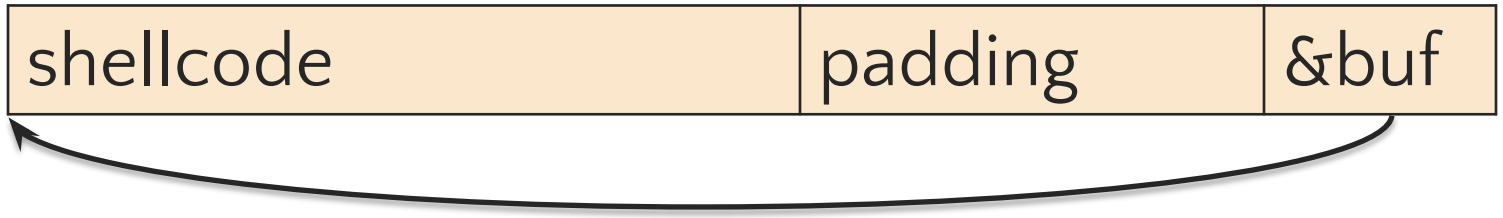


Recap

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***

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!