CS 165 – Computer Security

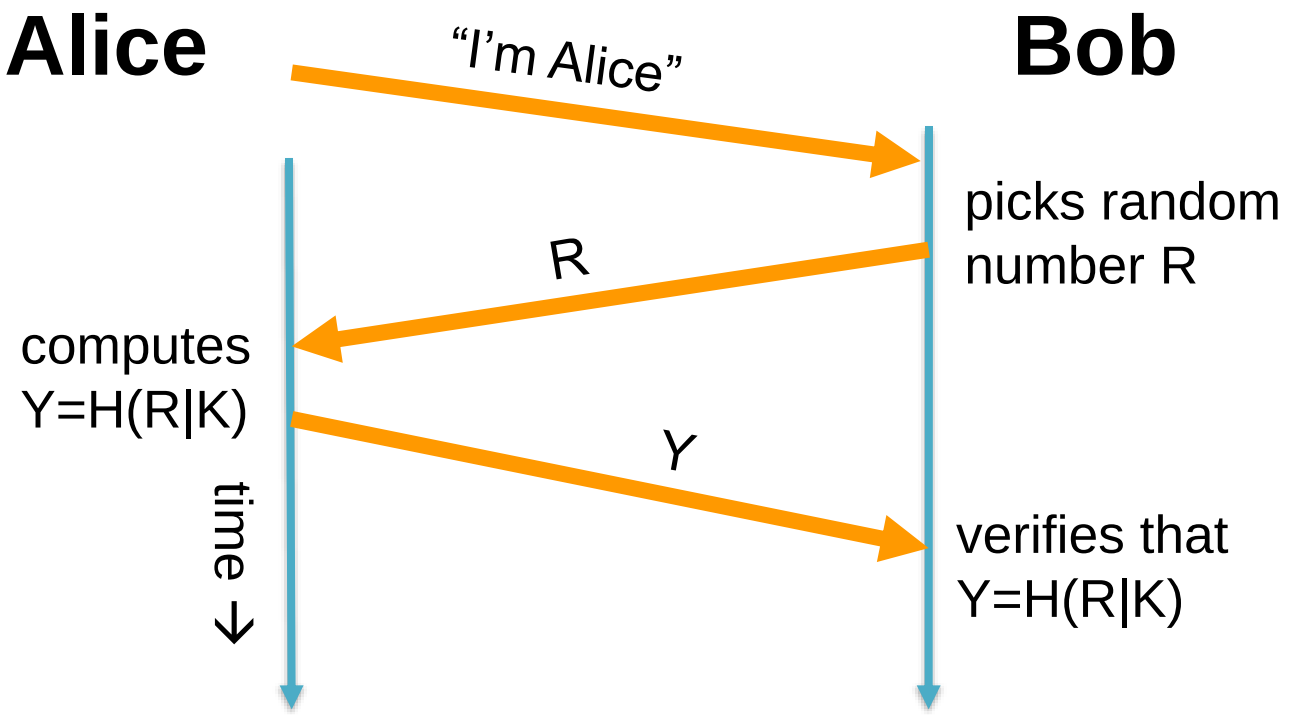
Password, hashes & low-level program execution

Sep 28, 2021

PASSWORD

Authentication

- To prove who you are



Authentication

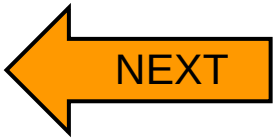
- There are four general means of authenticating a user's identity
  - Something the user knows
    - **Password**, personal idnetification number (PIN)
  - Something the user possesses
    - Smart cards, physical keys, tokens
  - Something the user is (static biometrics)
    - Recognition by fingerprint, face, retina, iris
  - Something the user does (dynamic biometrics)
    - Recognition by voice pattern, hadwriting style, typing rhythm
- Can be used in combination
  - Two-factor, multi-factor authentication

Password Authentication

- Most widely used authentication method
- Key question: How to store the password in hard drive?

Agenda

- How to Store Password
- UNIX Password System Design



Store in plaintext

Username: password
Alice:123
Bob: 123456
…

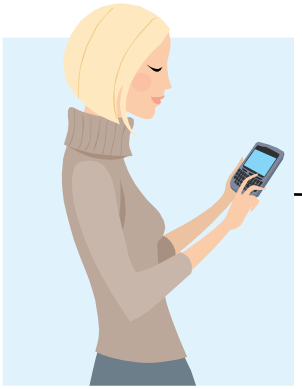
What's the problem of this approach?

RockYou hack compromises 32 million passwords

A hacker was able to break into the database of RockYou and obtain 32 million clear-text passwords through an SQL vulnerability.

http://www.scmagazine.com/rockyou-hack-compromises-32-million-passwords/article/159676/

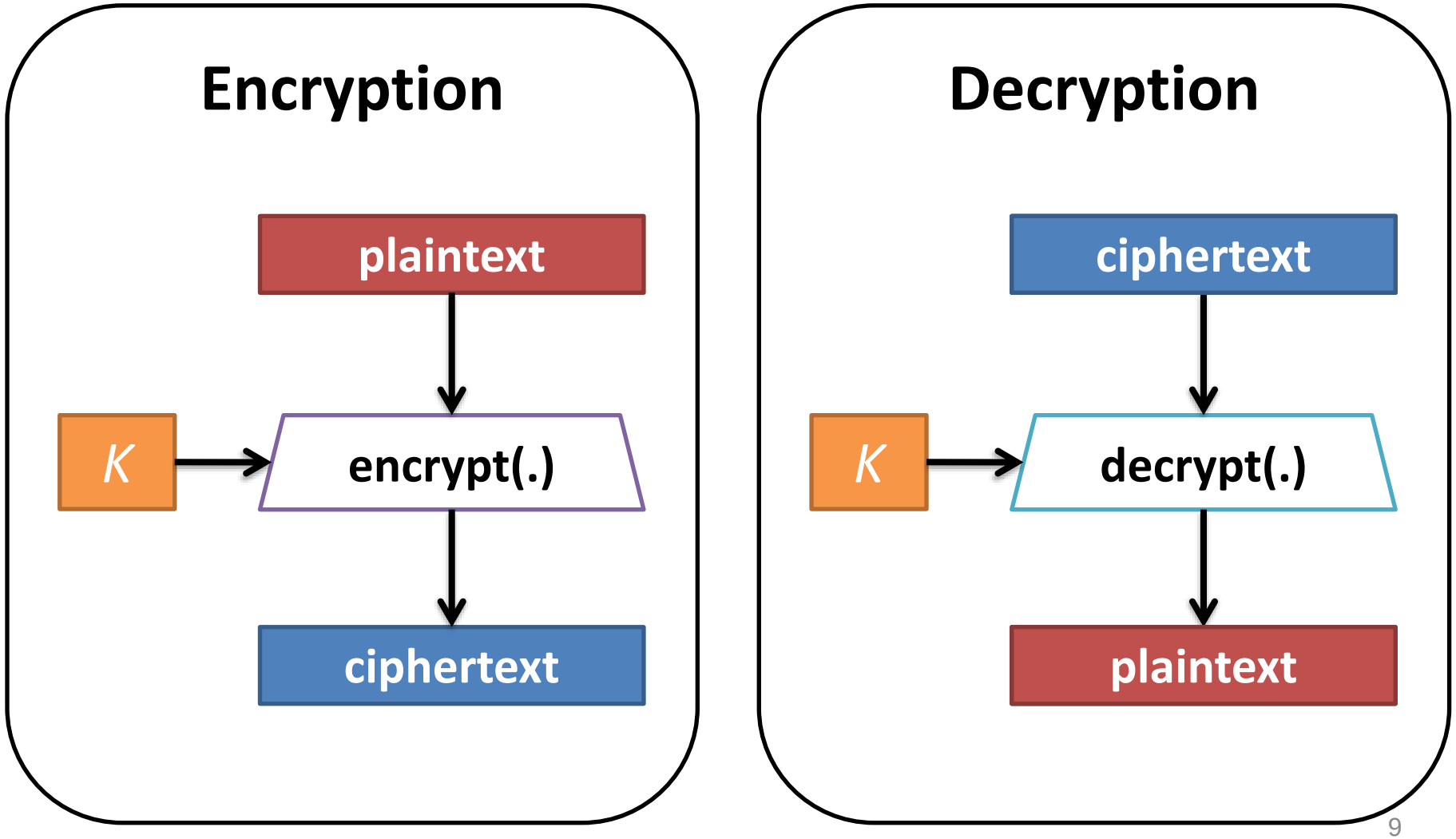
Basic confidentiality requirement



Message

Passive Eavesdropper

Confidentiality - Symmetric Key Encryption



Store E(k, password)

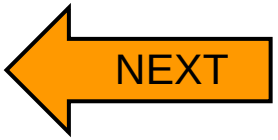
Username: E(k, password)
Alice: E(k, ’123’)
Bob: E(k, ‘123456’)
…

What’s the problem of this approach?

(1) If k gets compromised, all leaked
(2) It reveals two users have the same password if they choose the same one, which is a bad idea

Agenda

- How to Store Password
- UNIX Password System Design



Store in plaintext

Username: password
Alice:123
Bob: 123456
…

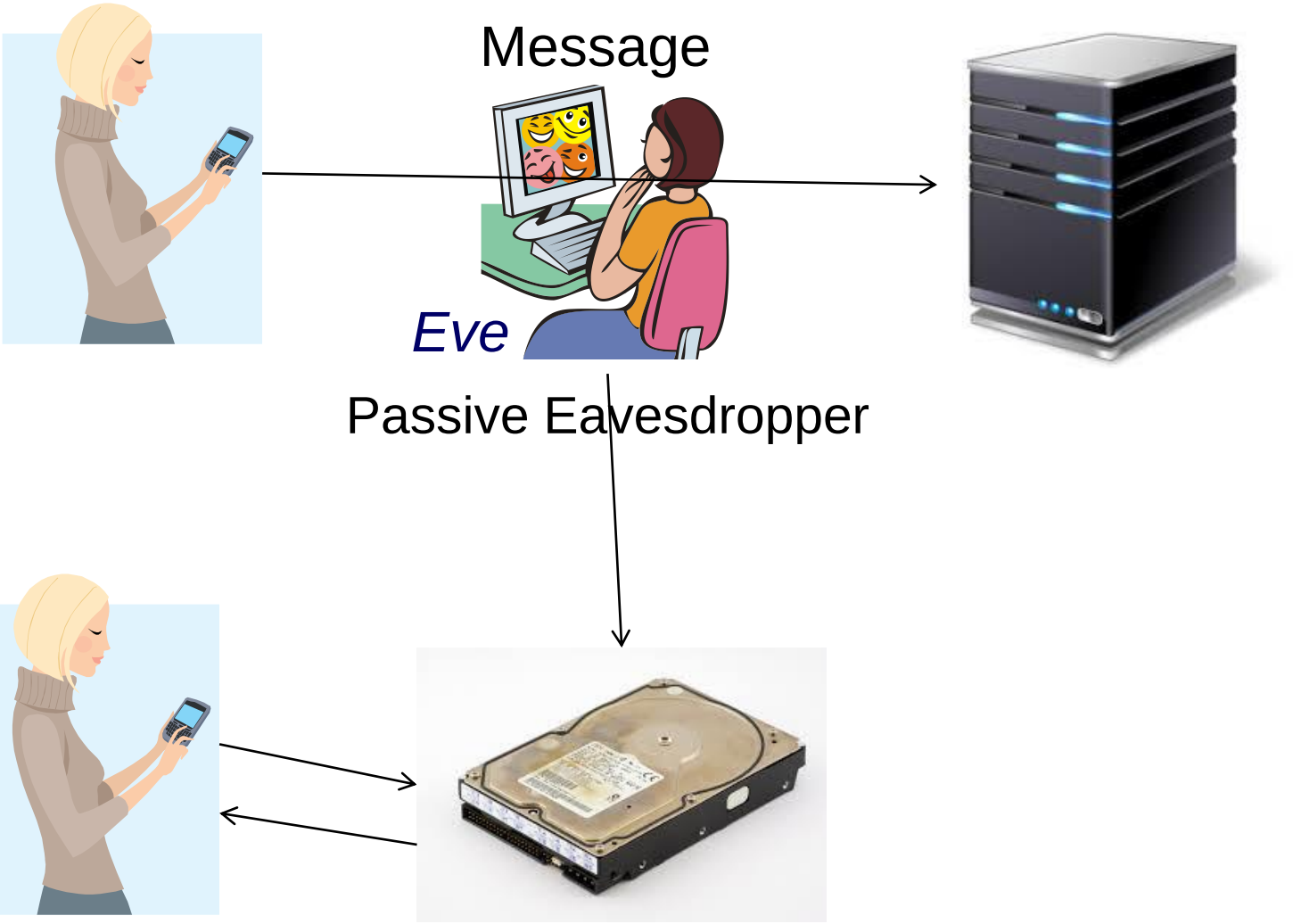
What's the problem of this approach?

RockYou hack compromises 32 million passwords

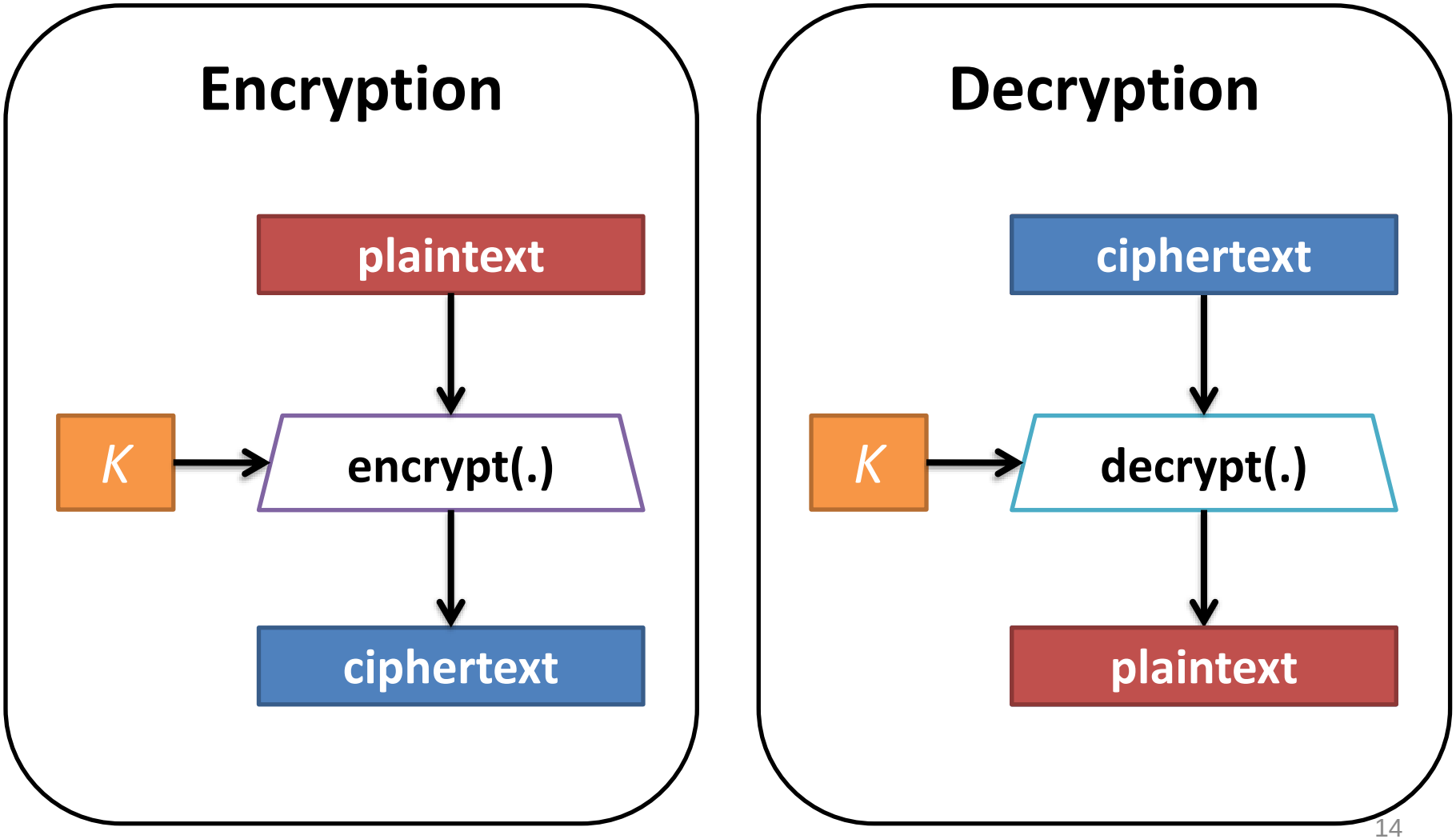
A hacker was able to break into the database of RockYou and obtain 32 million clear-text passwords through an SQL vulnerability.

http://www.scmagazine.com/rockyou-hack-compromises-32-million-passwords/article/159676/

Basic confidentiality requirement



Confidentiality - Symmetric Key Encryption



Store E(k, password)

Username: E(k, password)
Alice: E(k, ’123’)
Bob: E(k, ‘123456’)
…

What’s the problem of this approach?

(1) If k gets compromised, all leaked
(2) It reveals two users have the same password if they choose the same one, which is a bad idea

Store H(password)

Username: H(password)
Alice: H('123')
Bob: H('123456')
…

Good idea?

- do not reveal passwords if file stolen
- exactly how OS (e.g., Linux) stores passwords

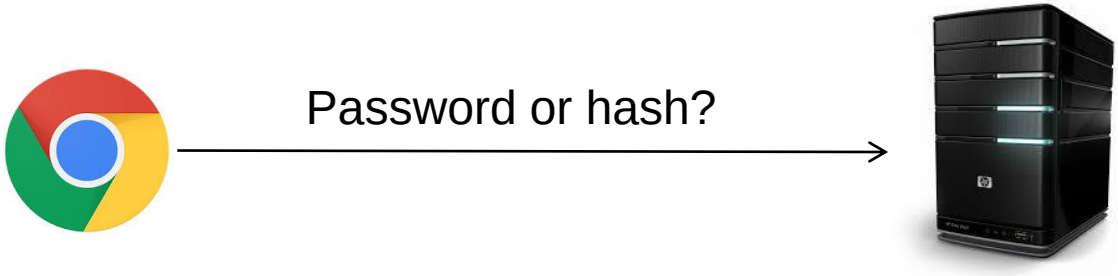
Server programs can also store their own users' password hashes in a file

- For Apache, it is /usr/local/apache/passwd

Browser's remembered passwords

Username: H(password)
Alice: H('123')
Bob: H('123456')
…

Why don't browsers store hashes?



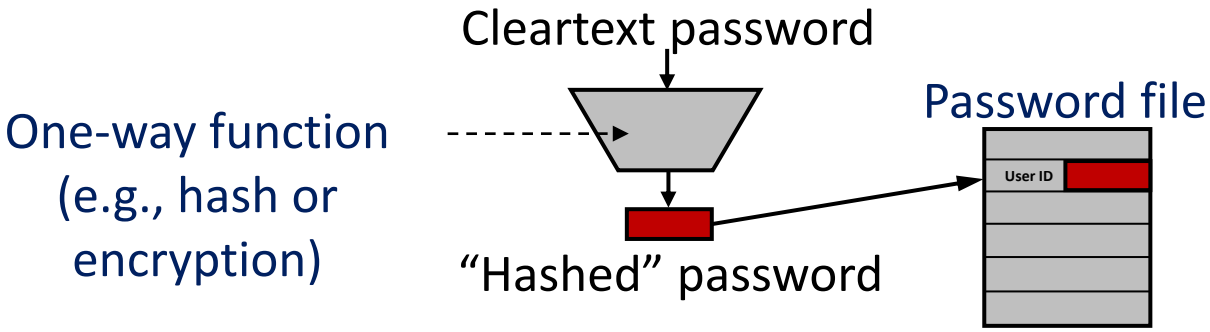
Store H(password)

Username: H(password)
Alice: H('123')
Bob: H('123456')
…

Any problem with this approach?

- It reveals two users have the same password if they choose the same one, which still leaks some information

Store H(password|salt)



Username: H(password|salt)
Alice: H('123456'|salt1)
Bob: H('123456'|salt2)
…

Is there **any** way to find out the password given a hash?

RANKING	PASSWORD USED	# OF USER WITH THIS PASSWORD
1	123456	290,731
2	12345	79,078
3	123456789	76,790
4	Password	61,958
5	Iloveyou	51,622
6	Princess	35,231
7	Rockyou	22,588
8	1234567	21,726
9	12345678	20,553
10	abc123	17,542
11	Nicole	17,168
12	Daniel	16,409
13	babygirl	16,094
14	monkey	15,294
15	Jessica	15,162
16	Lovely	14,950
17	michael	14,898
18	Ashley	14,329
19	654321	13,984
20	Qwerty	13,856

http://splashdata.com/press/WorstPasswords-2013.jpg

http://www.cbsnews.com/news/the-25-most-common-passwords-of-2013/

Not much different today

Brute Force – password cracking

Password Guessing (dictionary) Attack:

input: *hp* = hash(password) to crack

for each i in dictionary file

if(h(i) == hp)

output success;

Time Space Tradeoff Attack (rainbow table):

precompute: h(i) for each i in dict file in hash tbl

input: *hp* = hash(password)

check if *hp* is in hash tbl

Brute Force – password cracking

How hard is it to crack passwords?

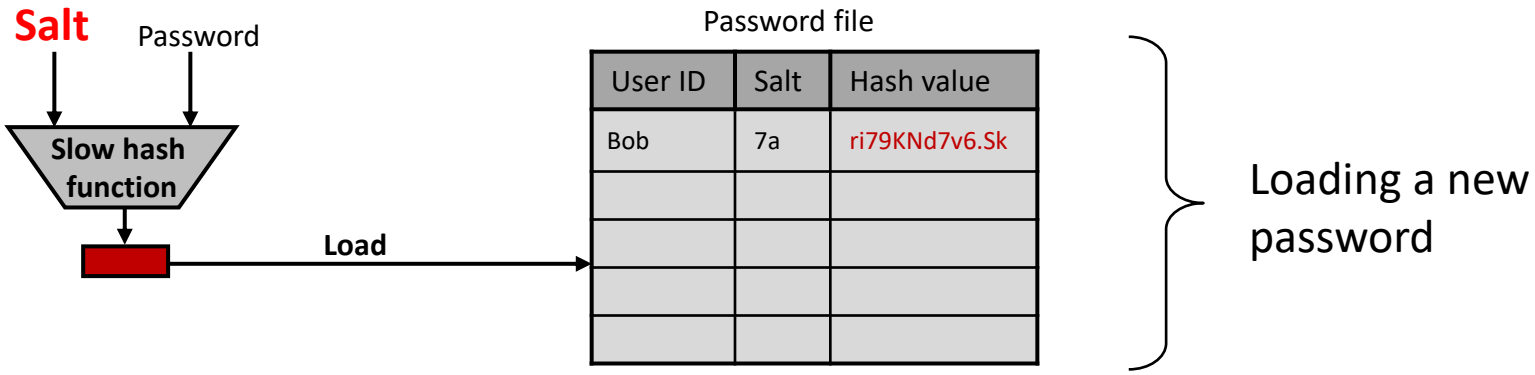
How many 8-character passwords assuming that 52 characters (upper and lower case) can be used?

52^8 = 53 trillion

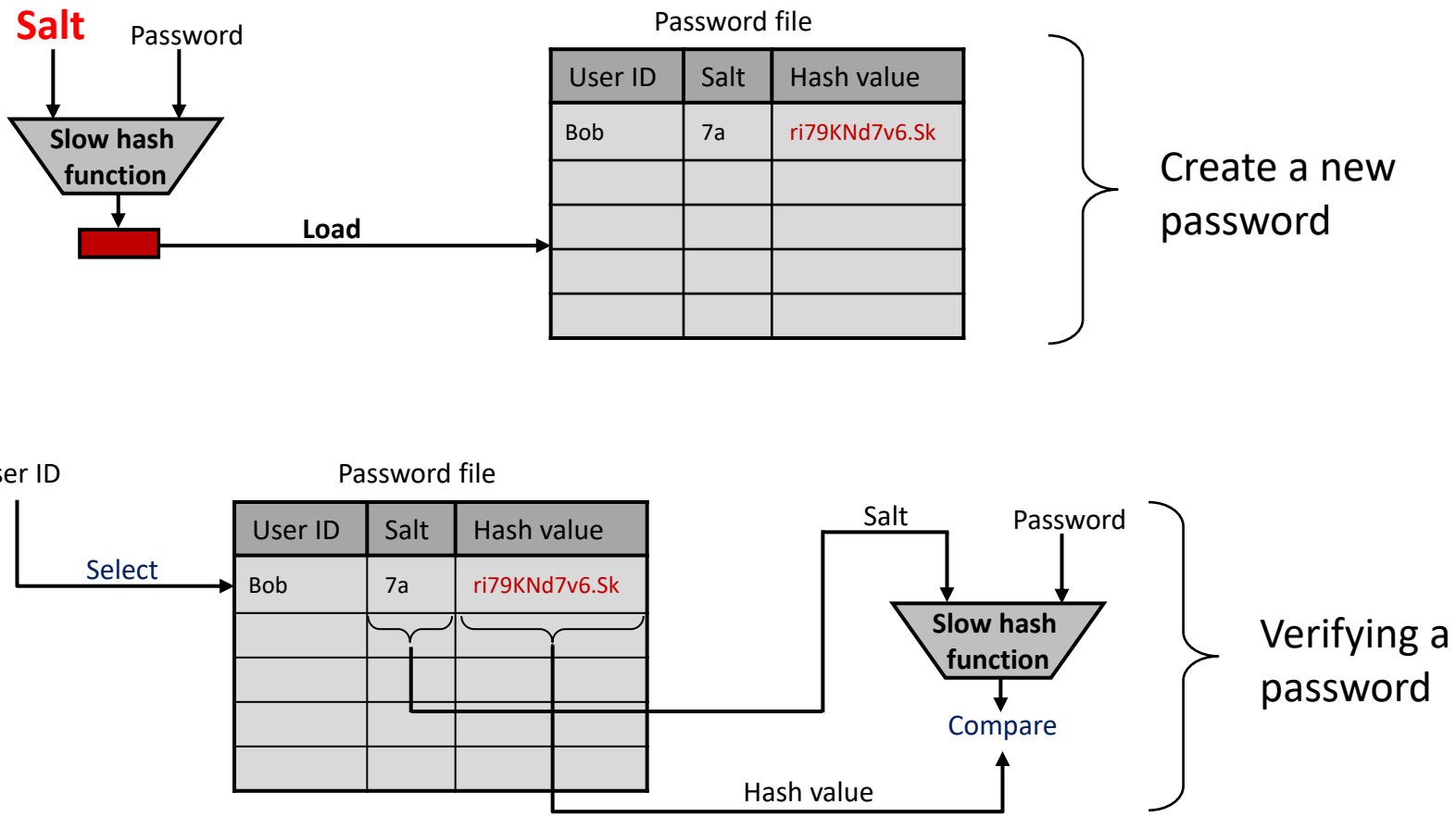
Agenda

- How to Store Password
- UNIX Password System Design

Modern Unix Password Scheme

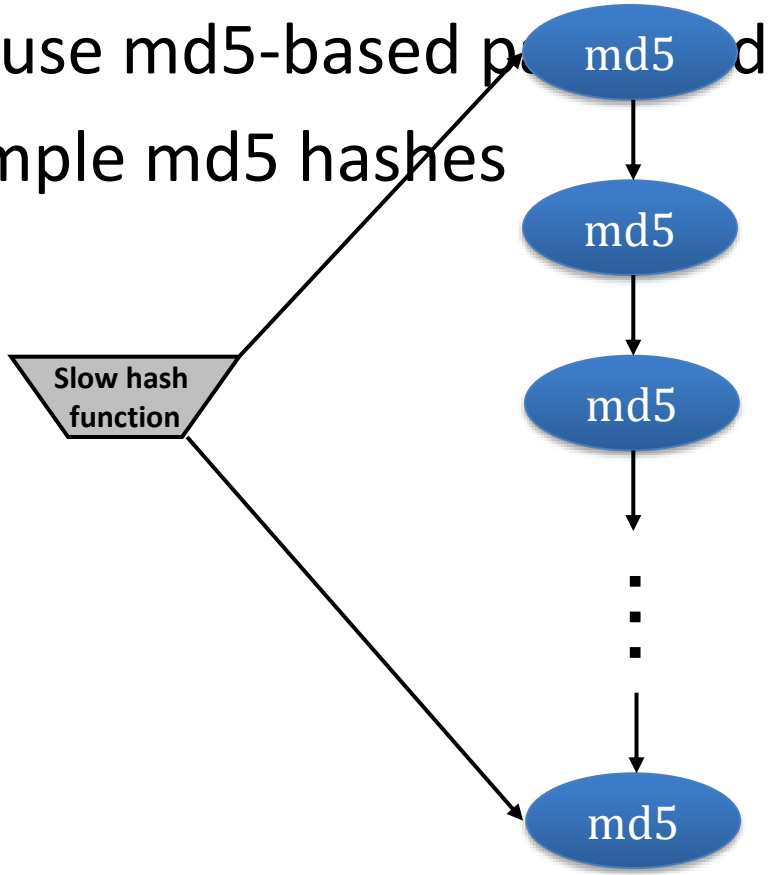


Modern Unix Password Scheme



Slow hash function

- Why slow hash? To slow down password cracking!
  - In project 1, we use md5-based p d hashes
  - 1000 times of simple md5 hashes



How difficult is it to crack passwords?

How many 8-character passwords given that 52 characters (upper and lower case) are available?

52^8 = 53 trillion

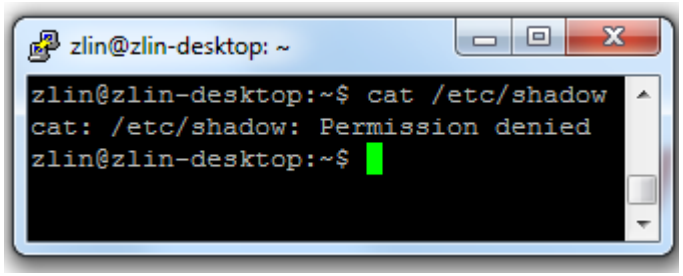
CPUs can do millions of primitive hashes per second = thousands (at least) of password hashes

-> ~100000 days to bruteforce

- Project 1 asks you to crack 6-character passwords with lowercase only. Much easier!

Password File Access

- Old method: names and hashes are stored in /etc/passwd
  - Free for anybody to read
  - Opens up for easy offline dictionary attack
- Safer method: the hashes stored in separate file /etc/shadow
  - Only root can access to this file



Password File Access

- Old method: names and hashes are stored in /etc/passwd
  - Free for anybody to read
  - Opens up for easy offline dictionary attack
- Safer method: the hashes stored in separate file /etc/shadow
  - Only root can access to this file

Password File Access

- Old method: names and hashes are stored in /etc/passwd
  - Free for anybody to read
  - Opens up for easy offline dictionary attack
- Safer method: the hashes stored in separate file /etc/shadow
  - Only root can access to this file

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
zlin:x:1000:1000:zlin,,,:/home/zlin:/bin/bash
```

```
root:$6$OpBsSYf2$2N7.hAERKFhxFg
HGHLOIz4ngC0wlZATZK.yCZ7capUpkc
Hjusp1nmQFATZD
anMt/kTpsHKuZYYTYskillxnE/1:15549
:0:99999:7:::
```

- Theft of Unix Hashes
  - Goal: gain access to /etc/shadow
  - Take away the hard drive
    - Physical access
  - Obtain root privileges (e.g., by using an exploit)
    - Can be remotely done

Understanding low-level program execution

Source code vs. binary/assembly

```
.text:004014E9
.text:004014E9 ; Attributes: bp-based frame
.text:004014E9
.text:004014E9 ; _IMAGE_SECTION_HEADER *__cdecl find_pe_section(char *const image_base, const unsigned int rva)
.text:004014E9 find_pe_section proc near               ; CODE XREF: ___scrt_is_nonwritable_in_current_image+49↓p
.text:004014E9
.text:004014E9 image_base      = dword ptr  8
.text:004014E9 rva             = dword ptr  0Ch
.text:004014E9
.text:004014E9                 push    ebp
.text:004014EA                 mov     ebp, esp
.text:004014EC                 mov     eax, [ebp+image_base]
.text:004014EF                 push    esi
.text:004014F0                 mov     ecx, [eax+3Ch]
.text:004014F3                 add     ecx, eax
.text:004014F5                 movzx   eax, word ptr [ecx+14h]
.text:004014F9                 lea     edx, [ecx+18h]
.text:004014FC                 add     edx, eax
.text:004014FE                 movzx   eax, word ptr [ecx+6]
.text:00401502                 imul    esi, eax, 28h
.text:00401505                 add     esi, edx
.text:00401507                 cmp     edx, esi
.text:00401509                 jz      short loc_401524
.text:0040150B                 mov     ecx, [ebp+rva]
.text:0040150E
.text:0040150E loc_40150E:                             ; CODE XREF: find_pe_section+39↓j
.text:0040150E                 cmp     ecx, [edx+0Ch]
.text:00401511                 jb      short loc_40151D
.text:00401513                 mov     eax, [edx+8]
.text:00401516                 add     eax, [edx+0Ch]
.text:00401519                 cmp     ecx, eax
.text:0040151B                 jb      short loc_401529
.text:0040151D
.text:0040151D loc_40151D:                             ; CODE XREF: find_pe_section+28↑j
.text:0040151D                 add     edx, 28h
.text:00401520                 cmp     edx, esi
.text:00401522                 jnz     short loc_40150E
.text:00401524
.text:00401524 loc_401524:                             ; CODE XREF: find_pe_section+20↑j
.text:00401524                 xor     eax, eax
.text:00401526
.text:00401526 loc_401526:                             ; CODE XREF: find_pe_section+42↓j
.text:00401526                 pop     esi
.text:00401527                 pop     ebp
.text:00401528                 retn
```

Source code vs. binary/assembly

- Source code
  - Readable
  - Allows development
  - Open source trend
- Binary
  - Difficult to understand
  - Protection of intellectual property

What can you do if you understand binaries

- Cracked software, game trainer, bots, patch
  - Through Reverse Engineering (RE)
  - Project 2
- Discover security vulnerabilities
  - Project 3
  - Will try to give some demo next week

What will executing this program do?

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
          name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```

To answer the question

"Is this program safe/secure/vulnerable?"

We need to know

"What will executing this program do?"

To answer the question

"Is this program safe/secure/vulnerable?"

We need to know

"What will executing this program do?"

Understanding the **compiler** and **machine semantics** are key.

Agenda

- Compilation Workflow
- x86 Execution Model
  - Basic Execution
  - Memory Operation
  - Control Flow
  - Memory Organization

Agenda

- Compilation Workflow
- x86 Execution Model
  - Basic Execution
  - Memory Operation
  - Control Flow
  - Memory Organization

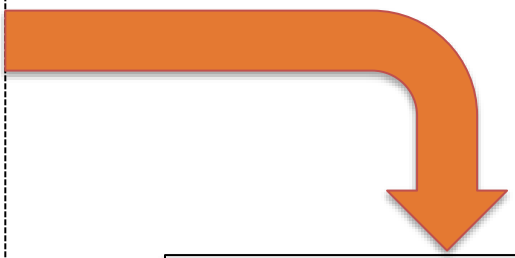
```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```

```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```

Compilation

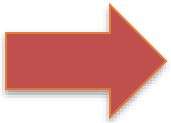
00110101
10101010
00101

```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```



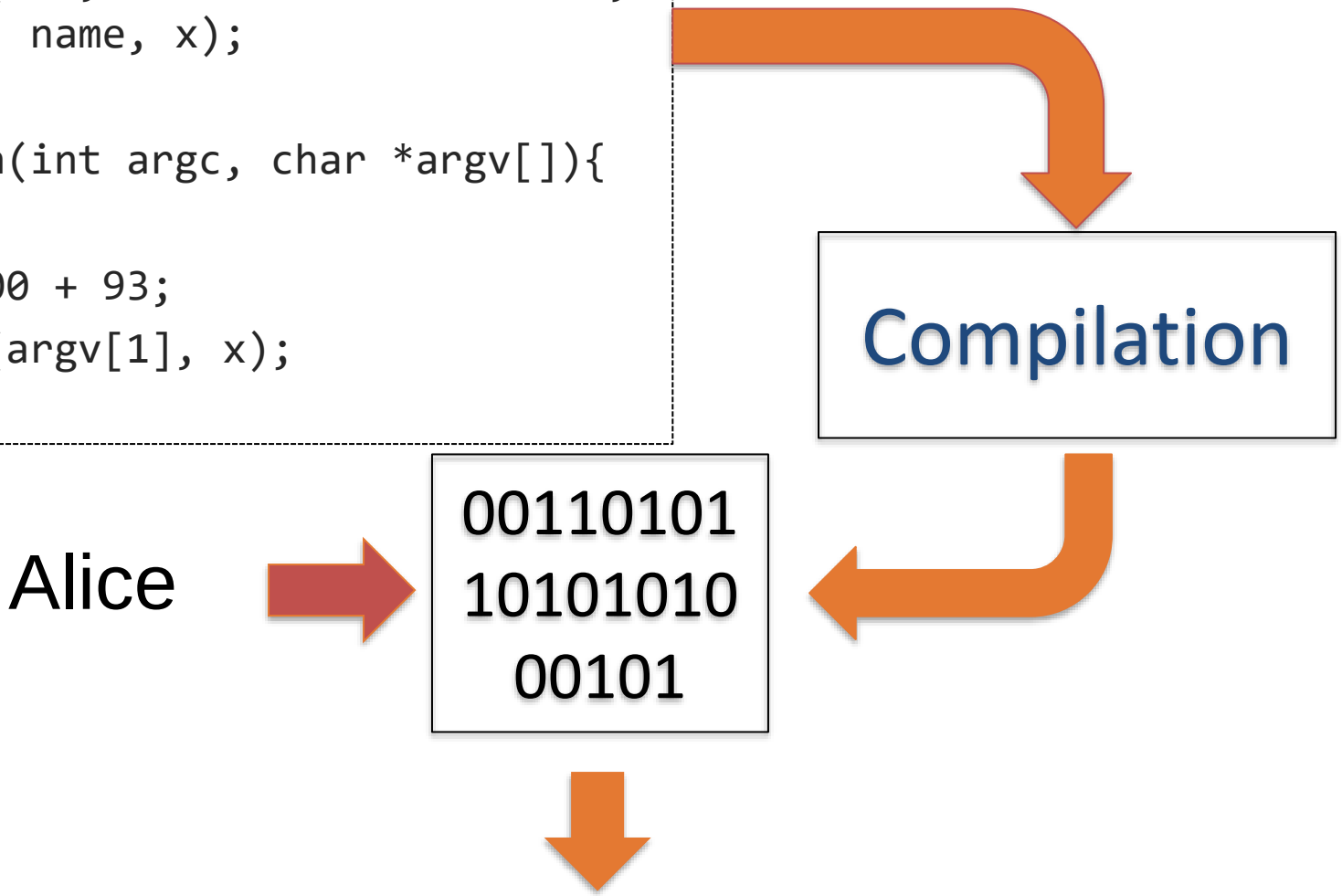
Compilation

00110101
10101010
00101



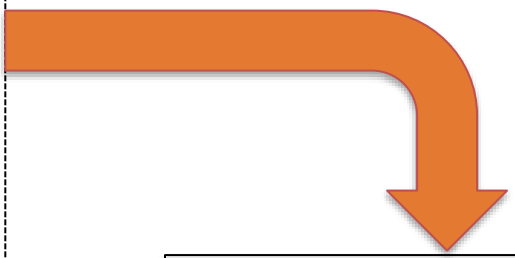
Alice

```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```



Alice, the answer is 4393

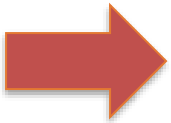
```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 4300 + 93;
  answer(argv[1], x);
}
```



Compilation

00110101
10101010
00101

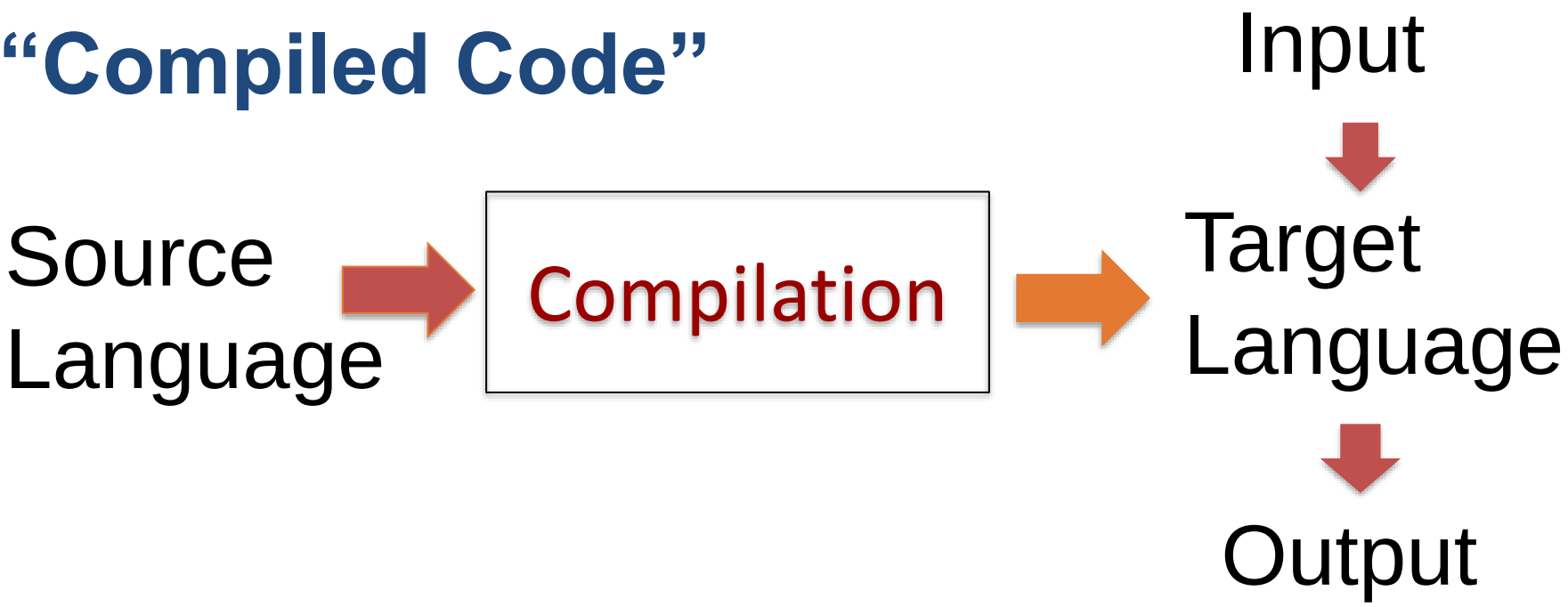
Alice



The ***compiler*** and ***machine*** determines the semantics

Alice, the answer is 4393

"Compiled Code"

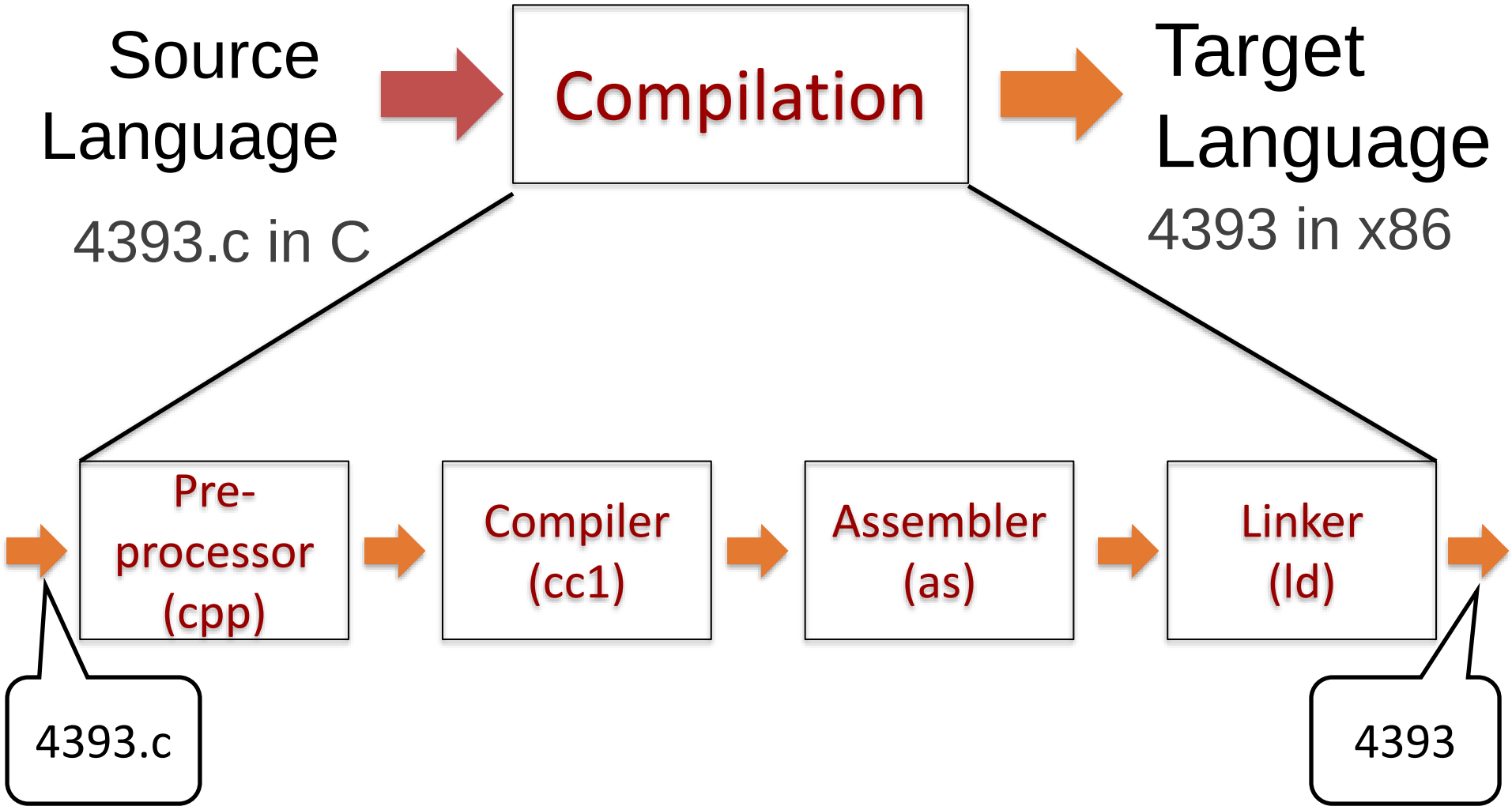


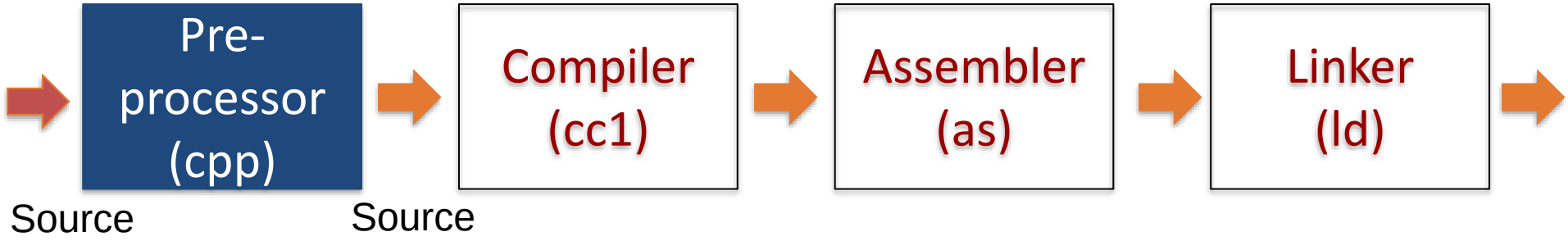
Terminology: Target language = machine-readable code = binary instructions

“Interpreted Code”


Source
Language
Compilation
Target
Language


Source Language
4393.c in C
Compilation
Target Language
4393 in x86

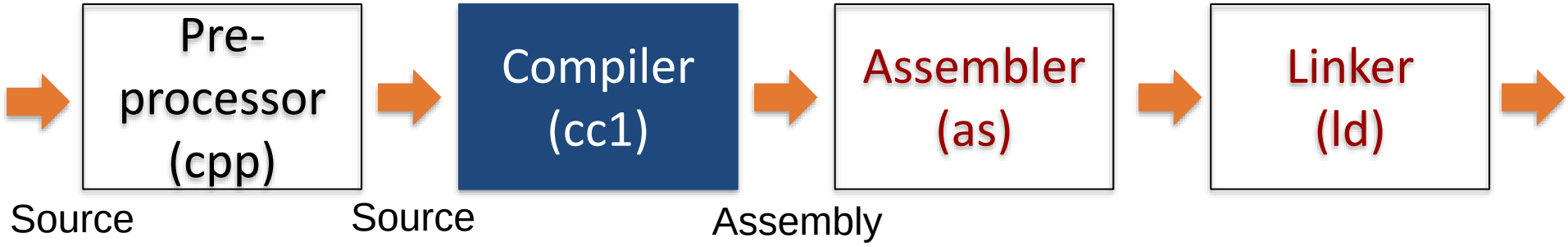

Source Language
4393.c in C
Compilation
Target Language
4393 in x86
Pre-processor (cpp)
Compiler (cc1)
Assembler (as)
Linker (ld)
4393.c
4393



$ cpp

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
...
```

#include expansion
#define substitution



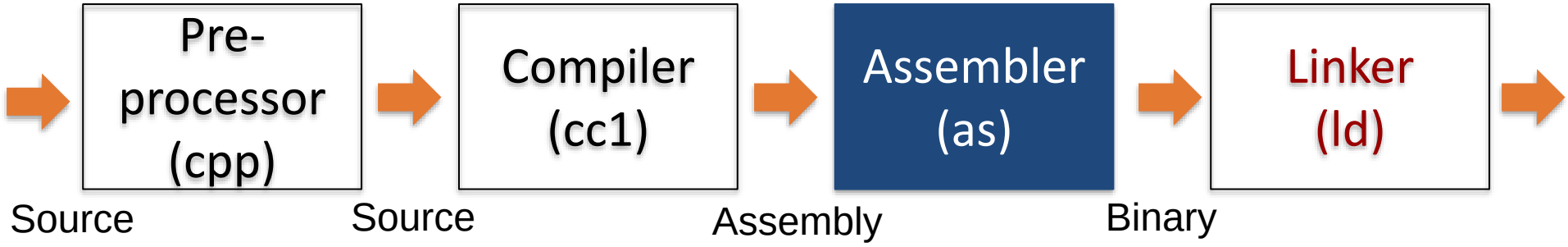
$ gcc -S

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
          name, x);
}
...
```

Creates Assembly

gcc –S 4393.c outputs 4393.s

```
_answer:
Leh_func_begin1:
        pushq   %rbp
Ltmp0:
        movq    %rsp, %rbp
Ltmp1:
        subq    $16, %rsp
Ltmp2:
        movl    %esi, %eax
        movq    %rdi, -8(%rbp)
        movl    %eax, -12(%rbp)
        movq    -8(%rbp), %rax
        ....
```

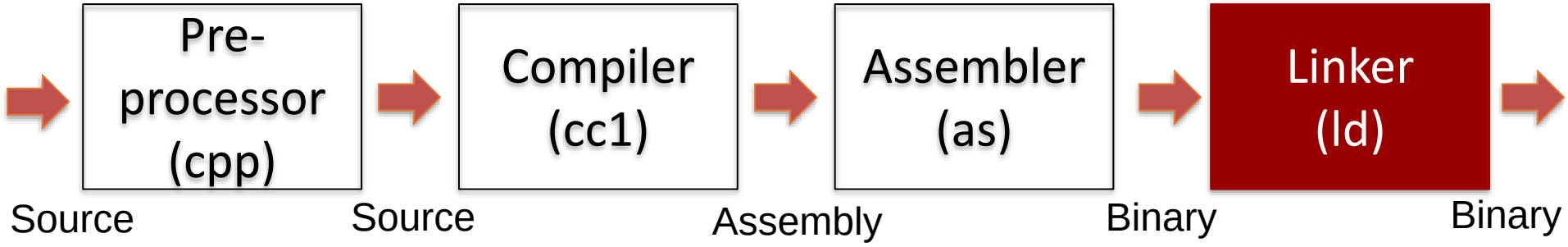


$ as <options>

```
_answer:
Leh_func_begin1:
        pushq   %rbp
Ltmp0:
        movq    %rsp, %rbp
Ltmp1:
        subq    $16, %rsp
Ltmp2:
        movl    %esi, %eax
        movq    %rdi, -8(%rbp)
        movl    %eax, -12(%rbp)
        movq    -8(%rbp), %rax
        ....
```

4393.s

Creates object code



$ ld <options>

```
01011001010101011010101
10101010101010101111111100
00110101011010100101011
01011110101001011000010101
10111101
```

4393.o

Links with other files and libraries to produce an exe