

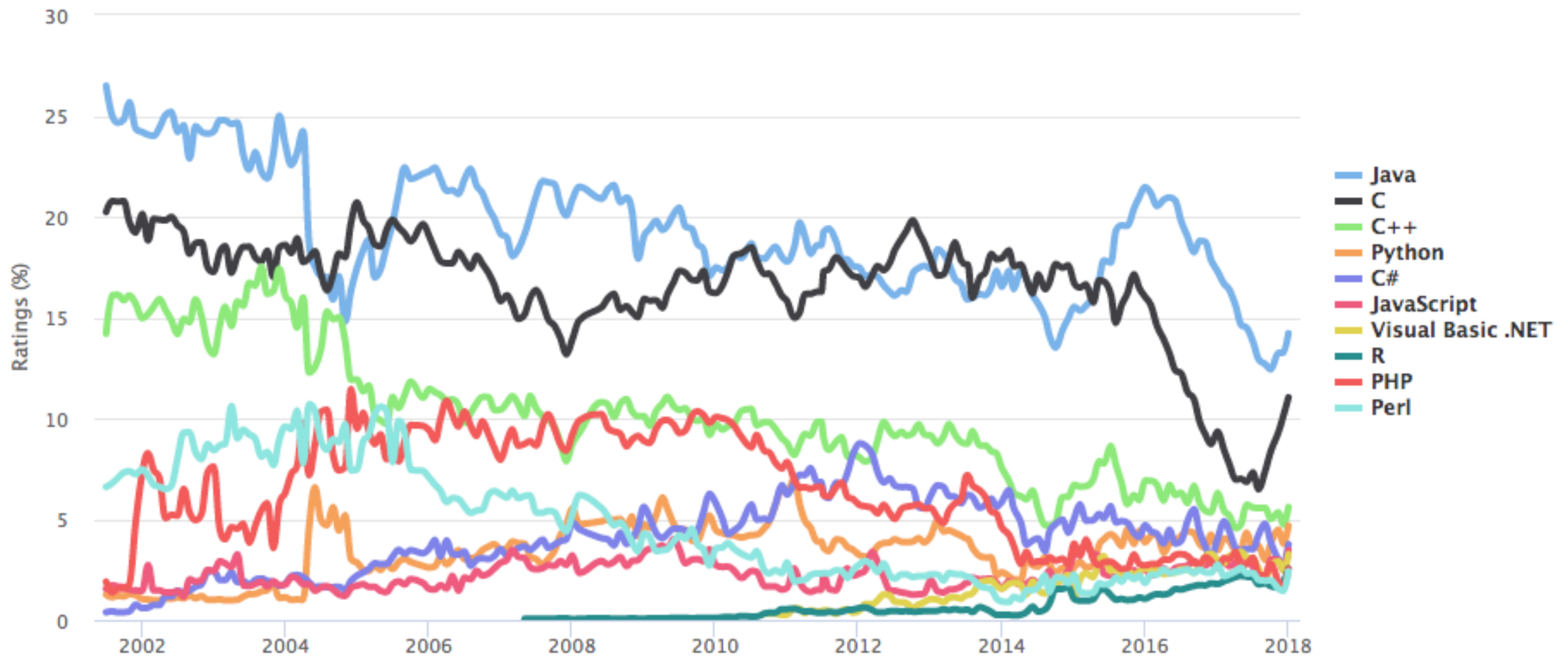
Memory Safety and Protection Mechanisms

51.502 Systems Security
Paweł Szalachowski























C/C++

TIOBE Programming Community Index

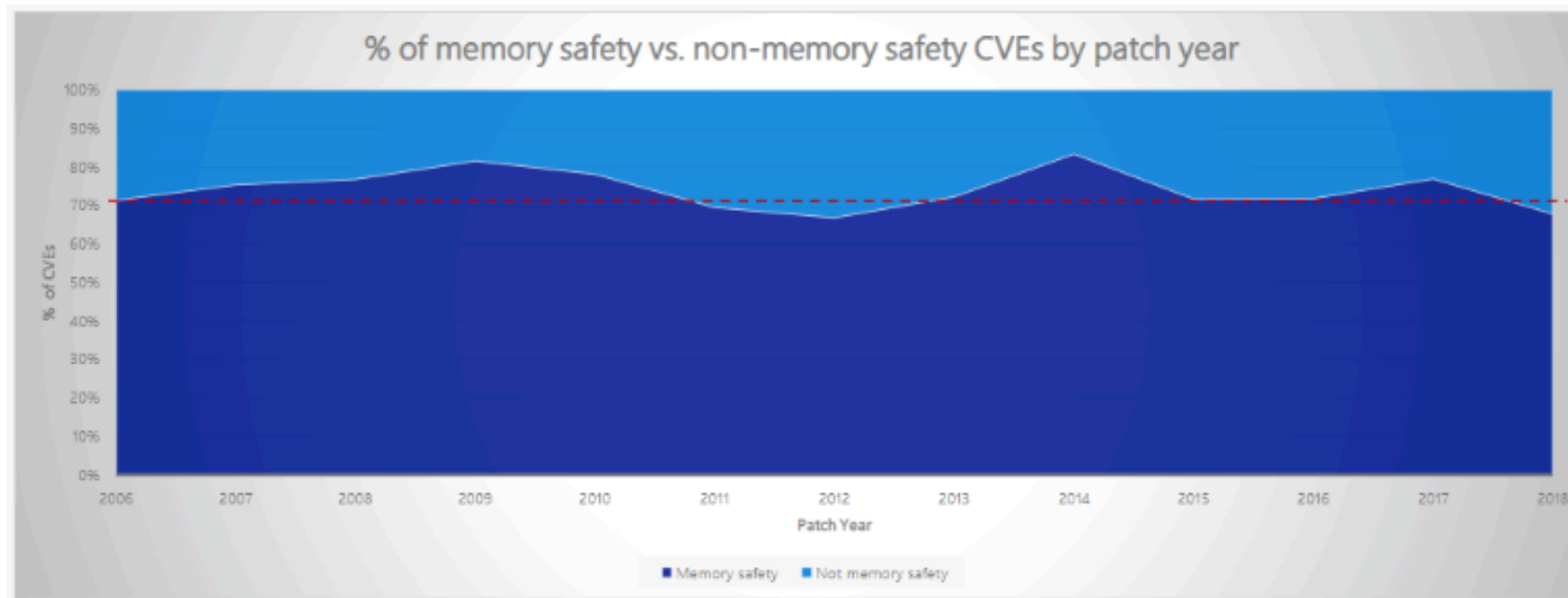
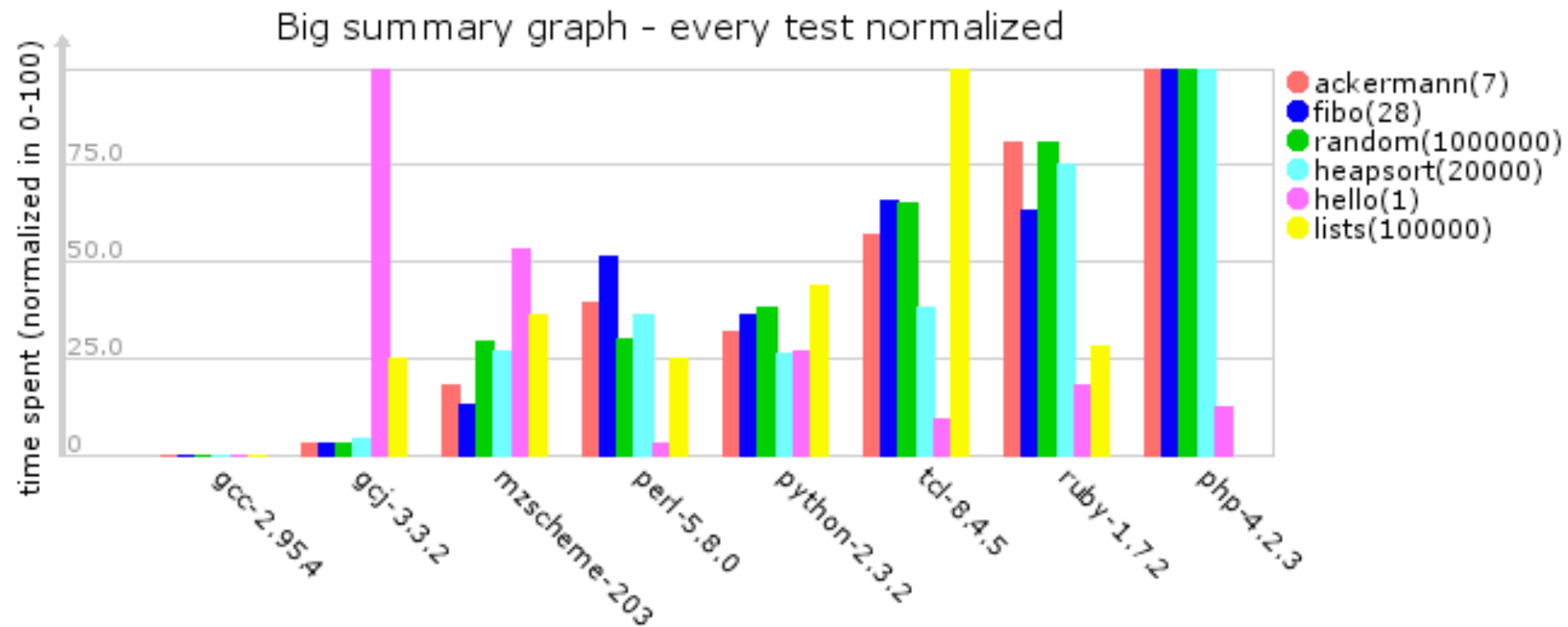
Source: www.tiobe.com



C/C++

Language Rank	Types	Spectrum Ranking
1. Python	 	100.0
2. C	  	99.7
3. Java	  	99.5
4. C++	  	97.1
5. C#	  	87.7
6. R		87.7
7. JavaScript	 	85.6
8. PHP		81.2
9. Go	 	75.1
10. Swift	 	73.7

C/C++

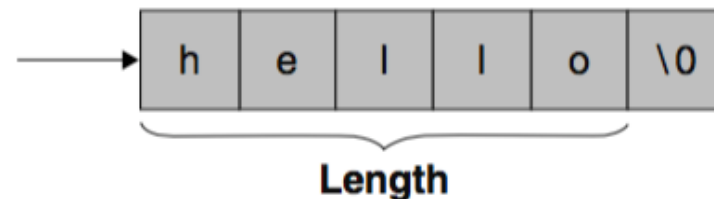


Memory Management

- C is a “simple” abstraction
 - Everything is a pointer
- “Manual” memory management
 - Static & dynamic allocation
 - Programmer responsible to allocate and free memory
 - Compiler does not check it
 - No garbage collector

Strings/Arrays in C

- No native support for strings
 - Contiguous sequence of characters terminated by null characters
 - “hello” is represented as



- String length vs array length
- `char msg[20]; int ints[20];`
- Arrays start at ???

Arrays start at 0.





"Our team needs a programmer to win the championship! You're our only hope!"

"Arrays start at 1!"



"I guess 2nd place is OK, too."

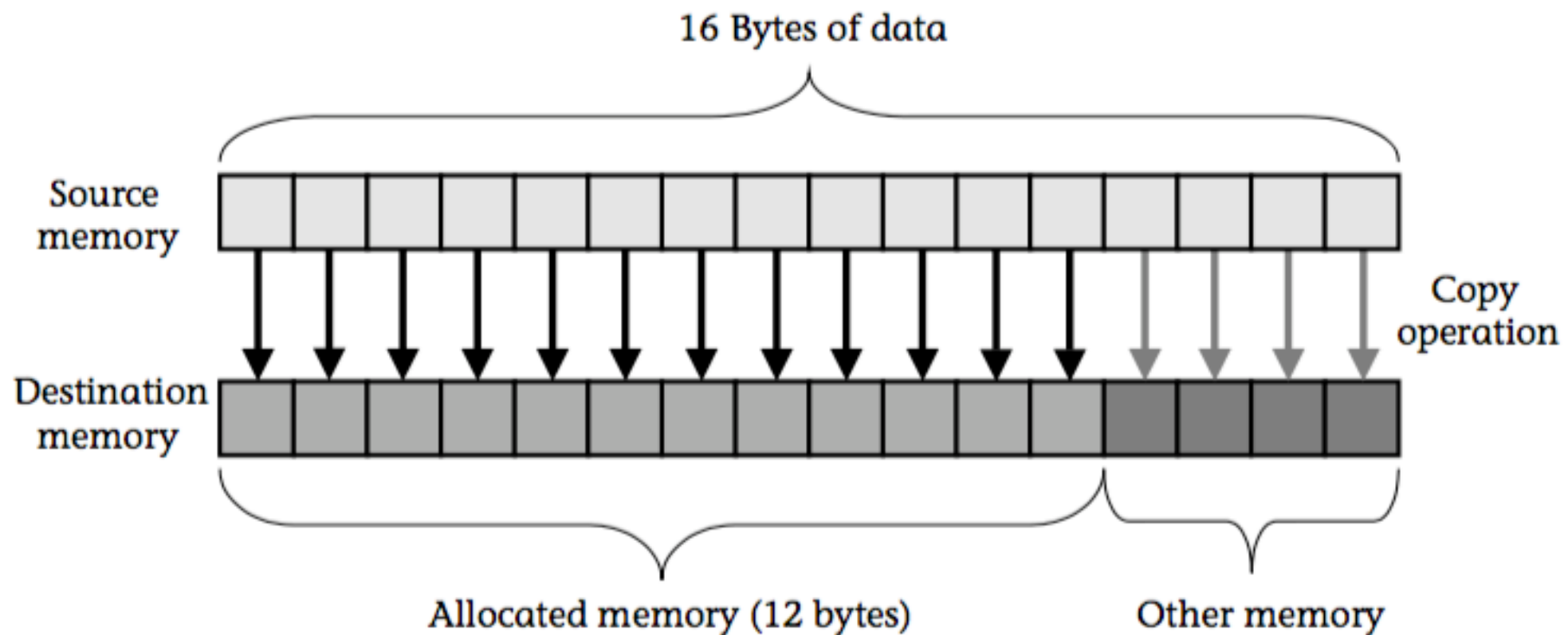
Microsoft Learning

Example

```
01  bool IsPasswordOK(void) {
02      char Password[12];
03
04      gets(Password);
05      return 0 == strcmp(Password, "goodpass");
06  }
07
08  int main(void) {
09      bool PwStatus;
10
11      puts("Enter password:");
12      PwStatus = IsPasswordOK();
13      if (PwStatus == false) {
14          puts("Access denied");
15          exit(-1);
16      }
17  }
```

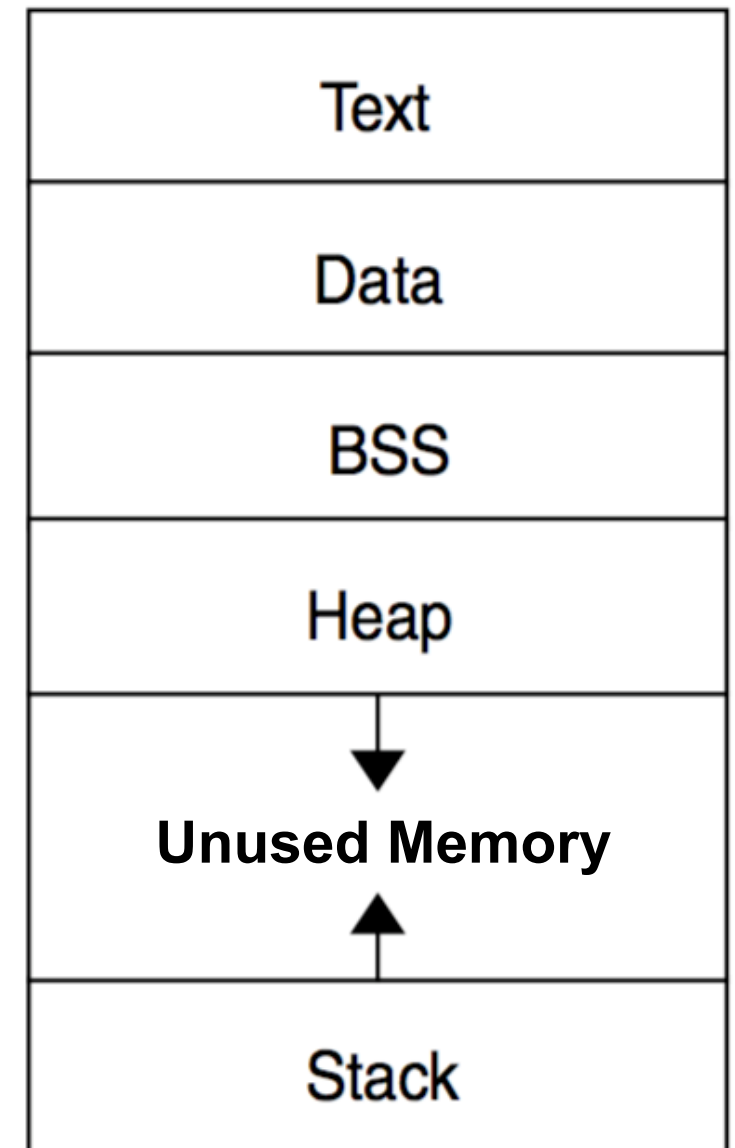
Bug ?

```
02 char Password[12];  
03  
04 gets(Password);
```

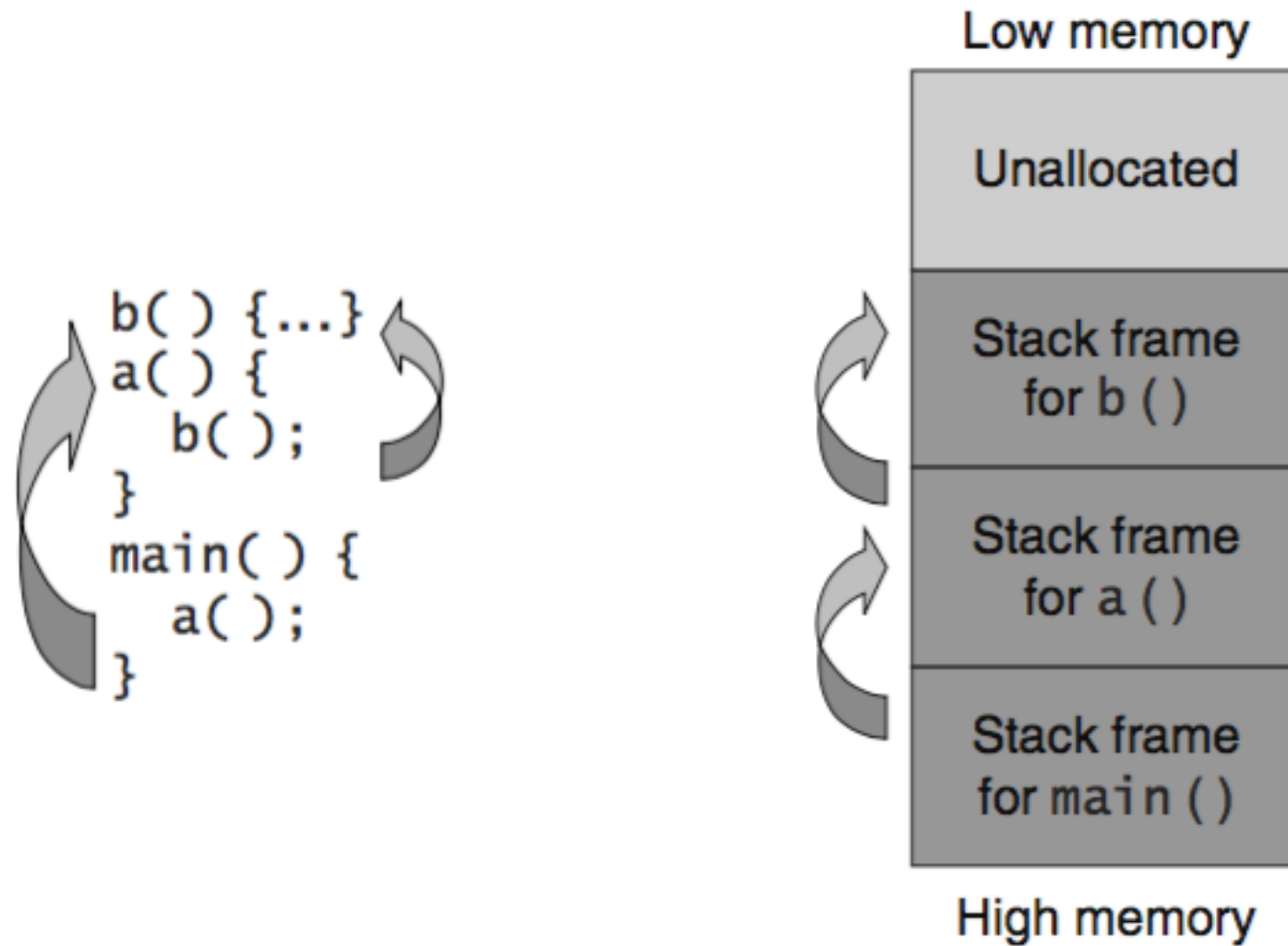


Process Memory Layout

- Each process has own dedicated memory
- In Unix, this memory is segmented as follows
 - **Text**: contains program instructions. It is readable and executable.
 - **Data**: contains initialized global variables. It is readable and writable.
 - **BSS**: contains uninitialized global variables. It is readable and writable.
 - **Heap**: contains dynamically allocated memory. Heap grows to higher-memory addresses.
 - **Stack**: contains memory associated to function calls (including statically allocated memory). Stack grows to lower-memory addresses.



Function Calls



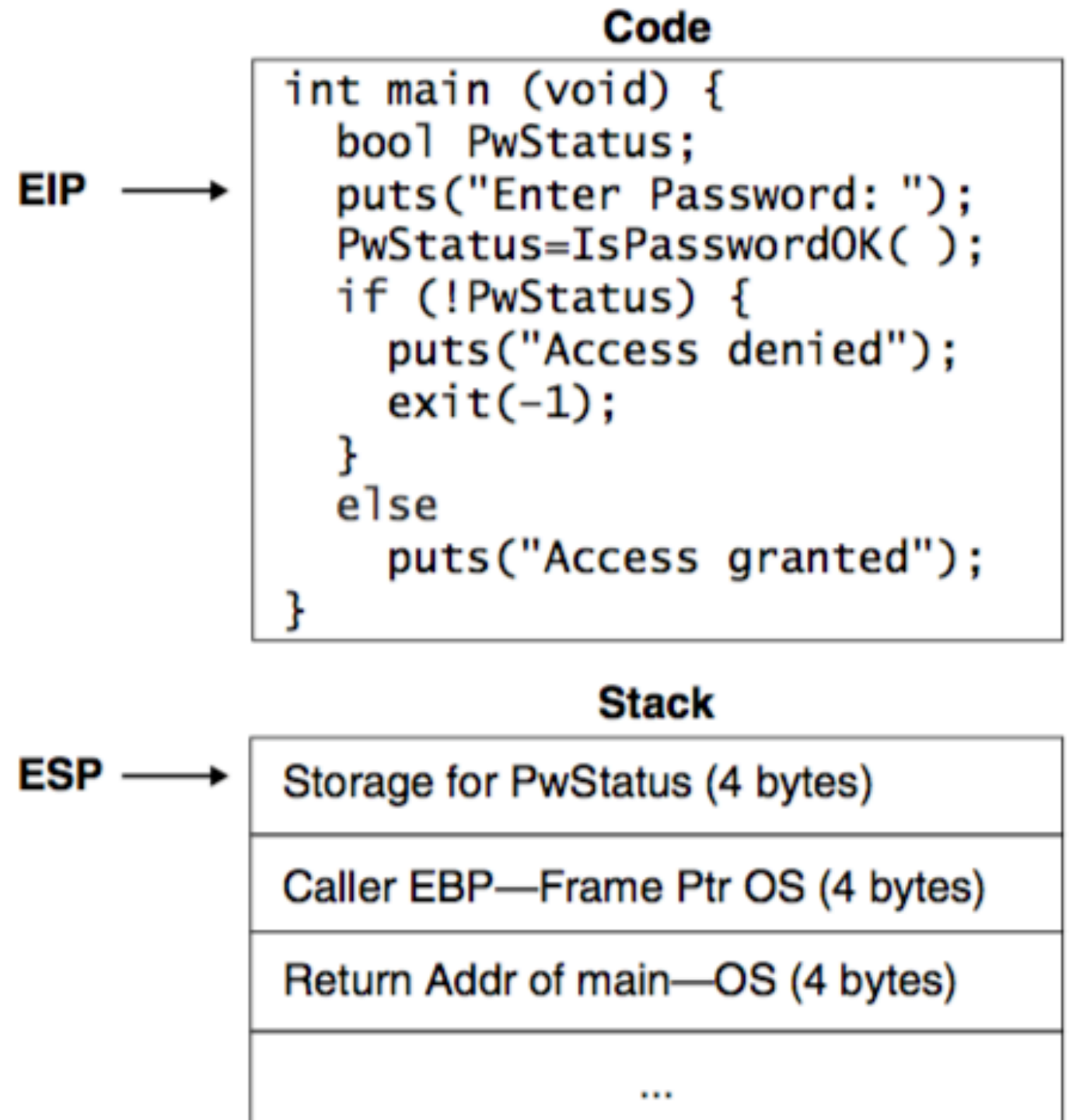
Stack Frame

- Stack stores stack frames:
 - Stack frames added at top of stack
 - Top stack frame corresponds to the currently executed function and is removed when the function returns
- Registers
 - ESP: mark top of stack
 - EBP: point the base address of stack
 - EIP: next instruction to read on the program
- Stack frame contains (at least):
 - Current function parameters
 - Return address (what to execute when the function finishes its execution)
 - The caller EBP, points to EBP in caller stack frame
 - Local function parameters (including local variables and statically allocated memory)

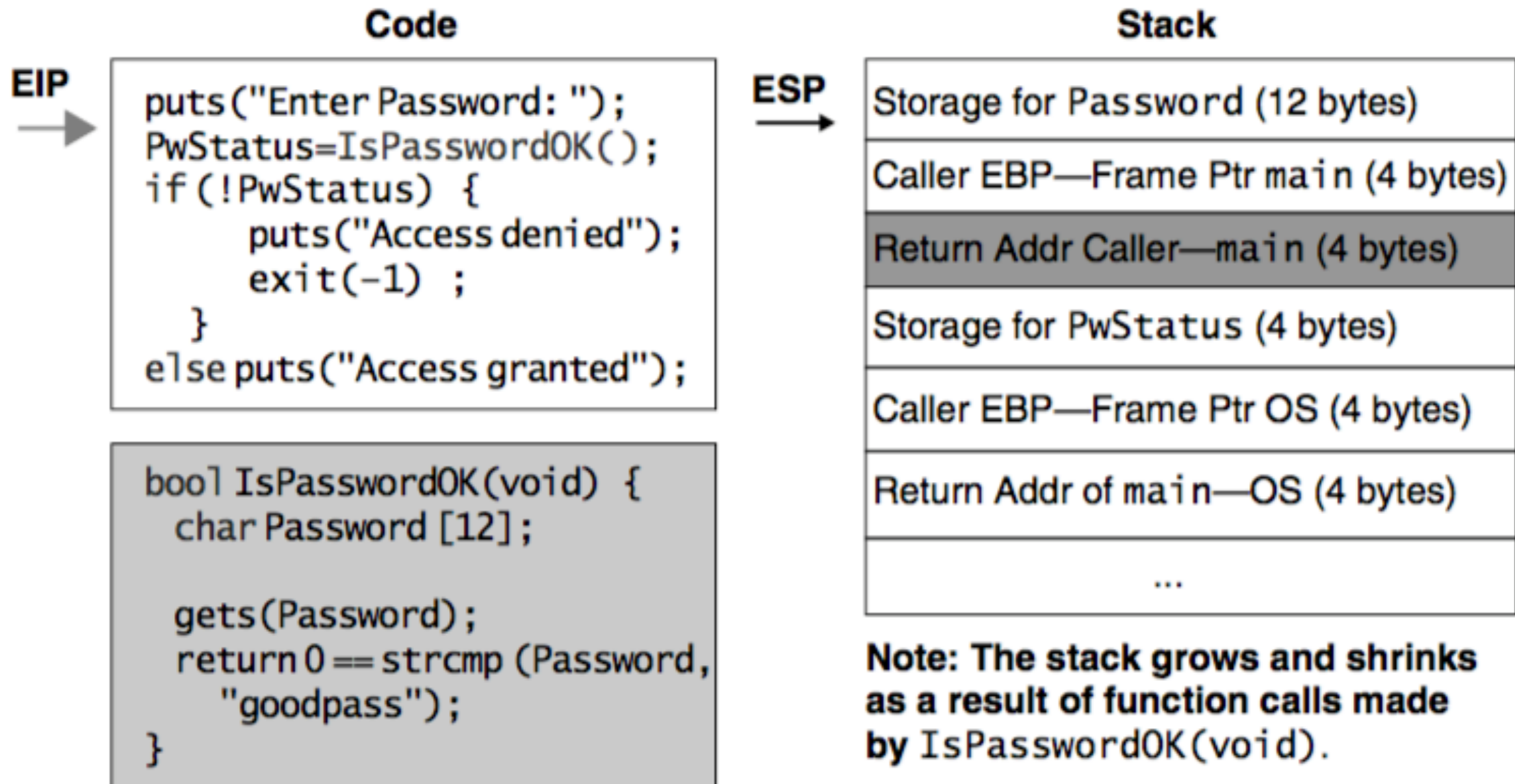
Example (reminder)

```
01  bool IsPasswordOK(void) {
02      char Password[12];
03
04      gets(Password);
05      return 0 == strcmp(Password, "goodpass");
06  }
07
08  int main(void) {
09      bool PwStatus;
10
11      puts("Enter password:");
12      PwStatus = IsPasswordOK();
13      if (PwStatus == false) {
14          puts("Access denied");
15          exit(-1);
16      }
17  }
```


- The stack before IsPasswordOK() is called
 - PwStatus
 - Caller's frame pointer
 - Return address



- The stack when IsPasswordOK() is executed



- The stack restored to initial state

Code

EIP



```
puts("Enter Password: ");  
PwStatus=IsPasswordOK( );  
if (!PwStatus) {  
    puts("Access denied");  
    exit(-1);  
}  
else puts("Access granted");
```

Stack

ESP



Storage for Password (12 bytes)

Caller EBP—Frame Ptr main
(4 bytes)

Return Addr Caller—main (4 bytes)

Storage for PwStatus (4 bytes)

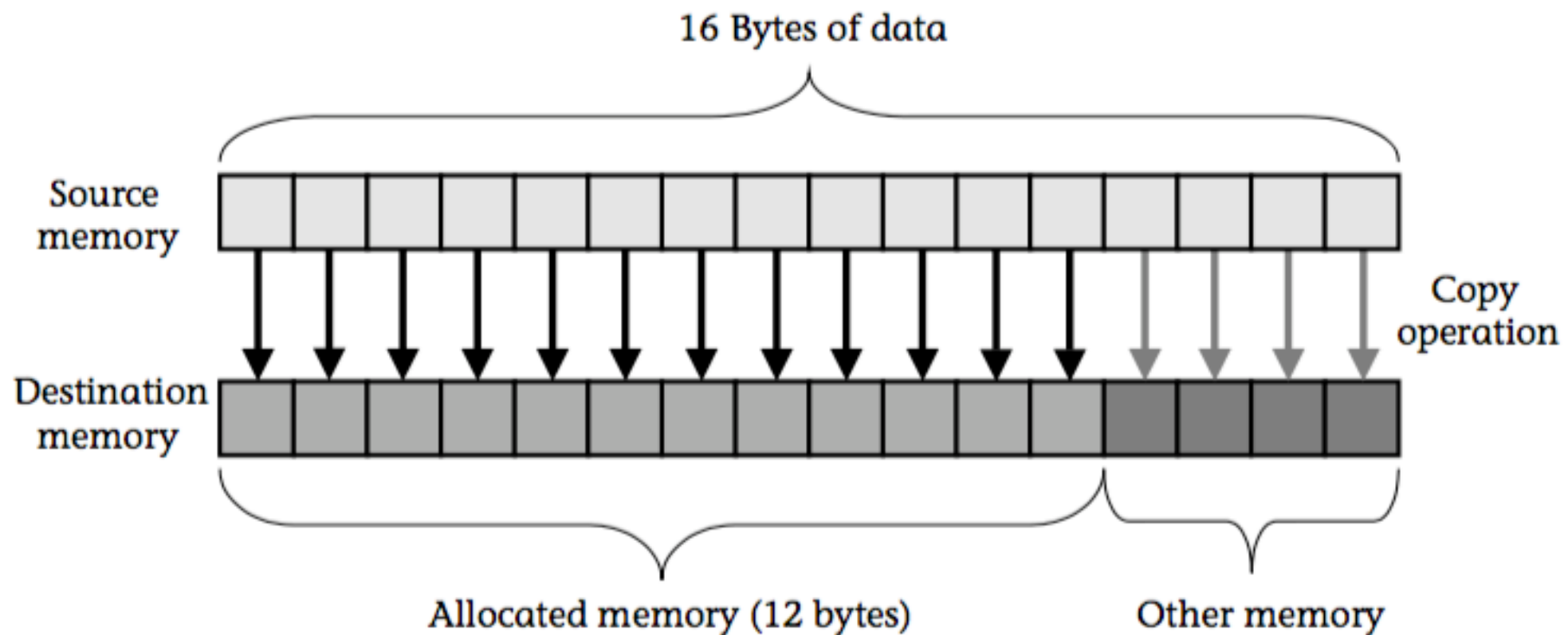
Caller EBP—Frame Ptr OS (4 bytes)

Return Addr of main—OS (4 bytes)

...

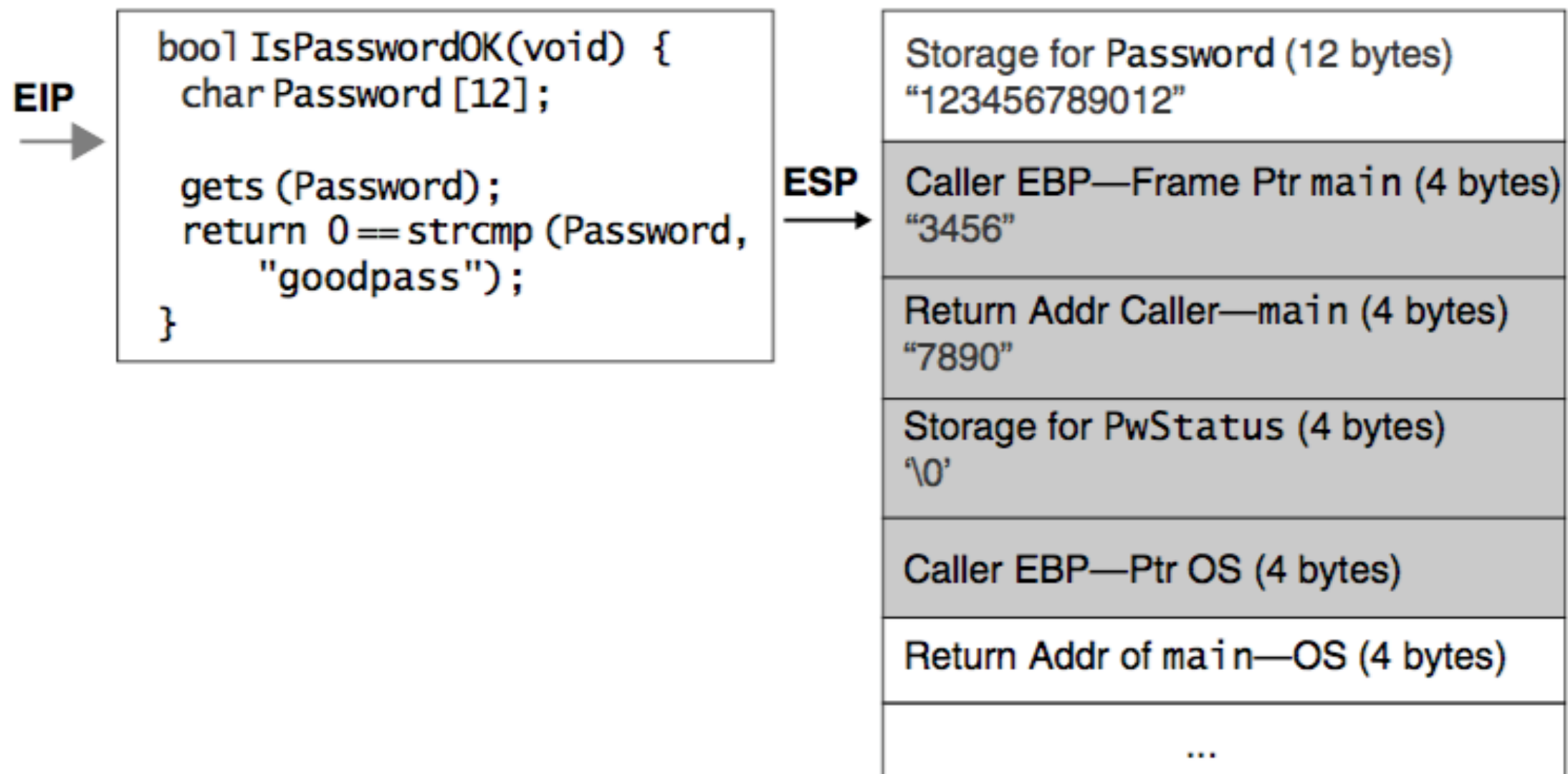
Bug (reminder)

```
02    char Password[12];  
03  
04    gets(Password);
```



Corrupted Stack

- Input 12345678901234567890
 - App will (should) crash



Corrupted Stack

- Input 1234567890123456W▶*!
- The return address is overwritten to “Access granted” branch

Line	Statement
1	puts("Enter Password: ");
2	PwStatus=IsPasswordOK();
3	if (!PwStatus)
4	puts("Access denied");
5	exit(-1);
6	else puts("Access granted");

Stack

Storage for Password (12 bytes) "123456789012"
Caller EBP—Frame Ptr main (4 bytes) "3456"
Return Addr Caller—main (4 bytes) "W▶*!" (return to line 6 was line 3)
Storage for PwStatus (4 bytes) '\0'
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)

How to exploit it?

- Denial-of-Service
 - Application can be crashed
 - Especially bad when it is a service (e.g., web server)
- Return address points to the next call
 - So what if we inject code (to stack) and modify the return address such that it points to that code?

Code Execution: ./app < exploit.bin

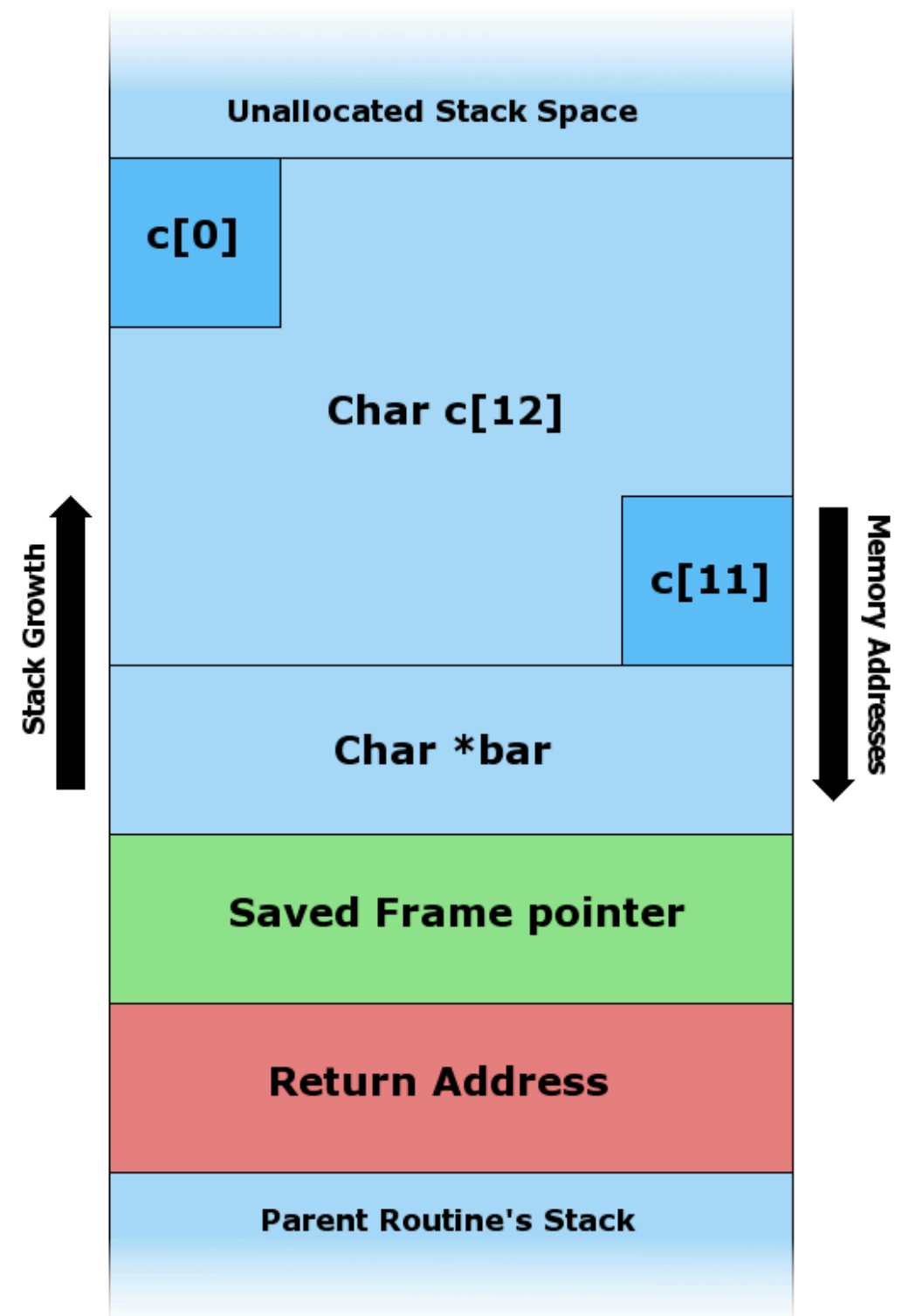
```
01  /* buf[12] */
02  00 00 00 00
03  00 00 00 00
04  00 00 00 00
05
06  /* %ebp */
07  00 00 00 00
08
09  /* return address */
10  78 fd ff bf
11
12  /* "/usr/bin/cal" */
13  2f 75 73 72
14  2f 62 69 6e
15  2f 63 61 6c
16  00 00 00 00
17
18  /* null pointer */
19  74 fd ff bf
20
21  /* NULL */
22  00 00 00 00
23
24  /* exploit code */
25  b0 0b      /* mov $0xb, %eax */
26  8d 1c 24   /* lea (%esp), %ebx */
27  8d 4c 24 f0 /* lea -0x10(%esp), %ecx */
28  8b 54 24 ec /* mov -0x14(%esp), %edx */
29  cd 50      /* int $0x50 */
```

Example II

```
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar);
}

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

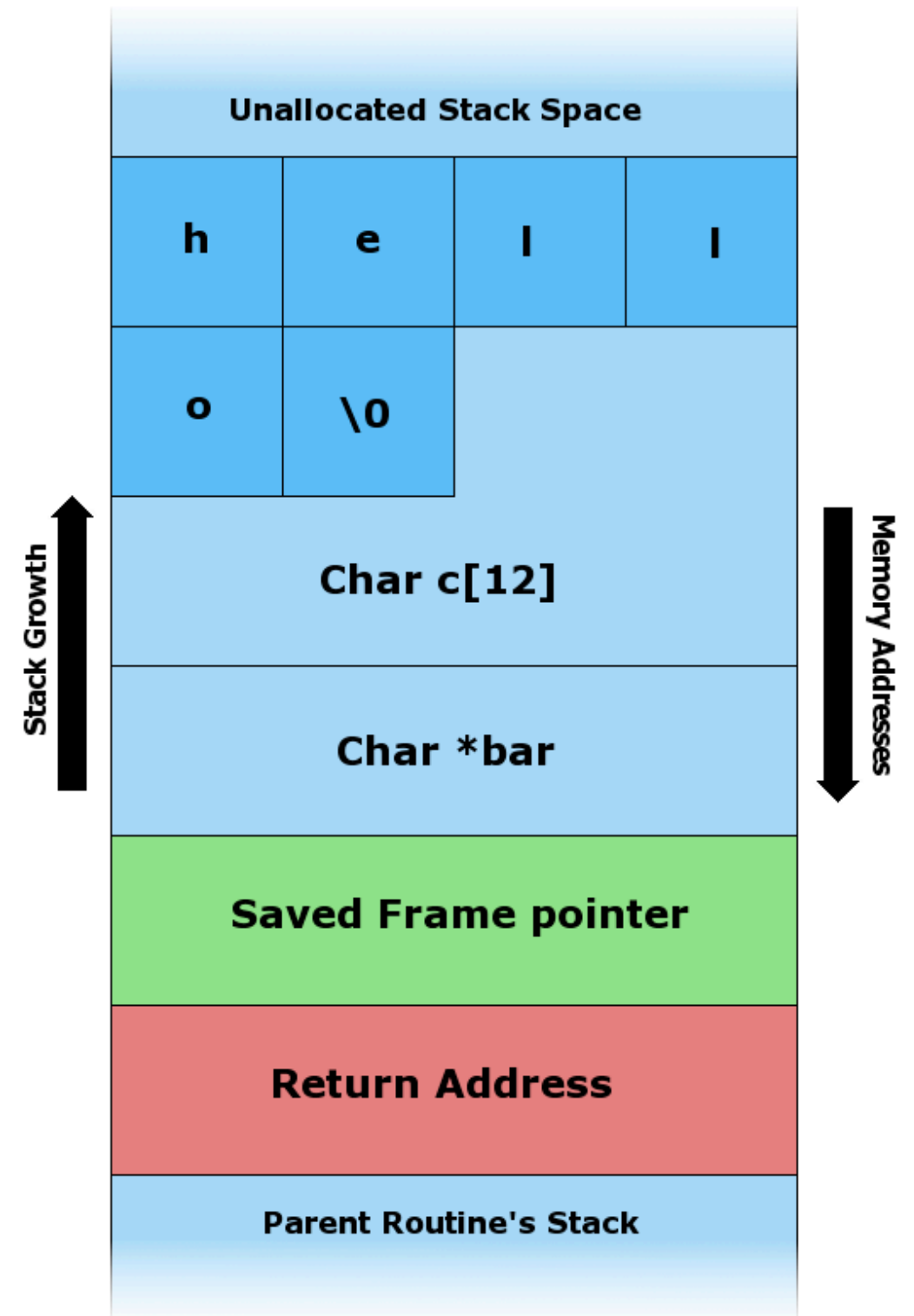


Example II

```
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar);
}

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

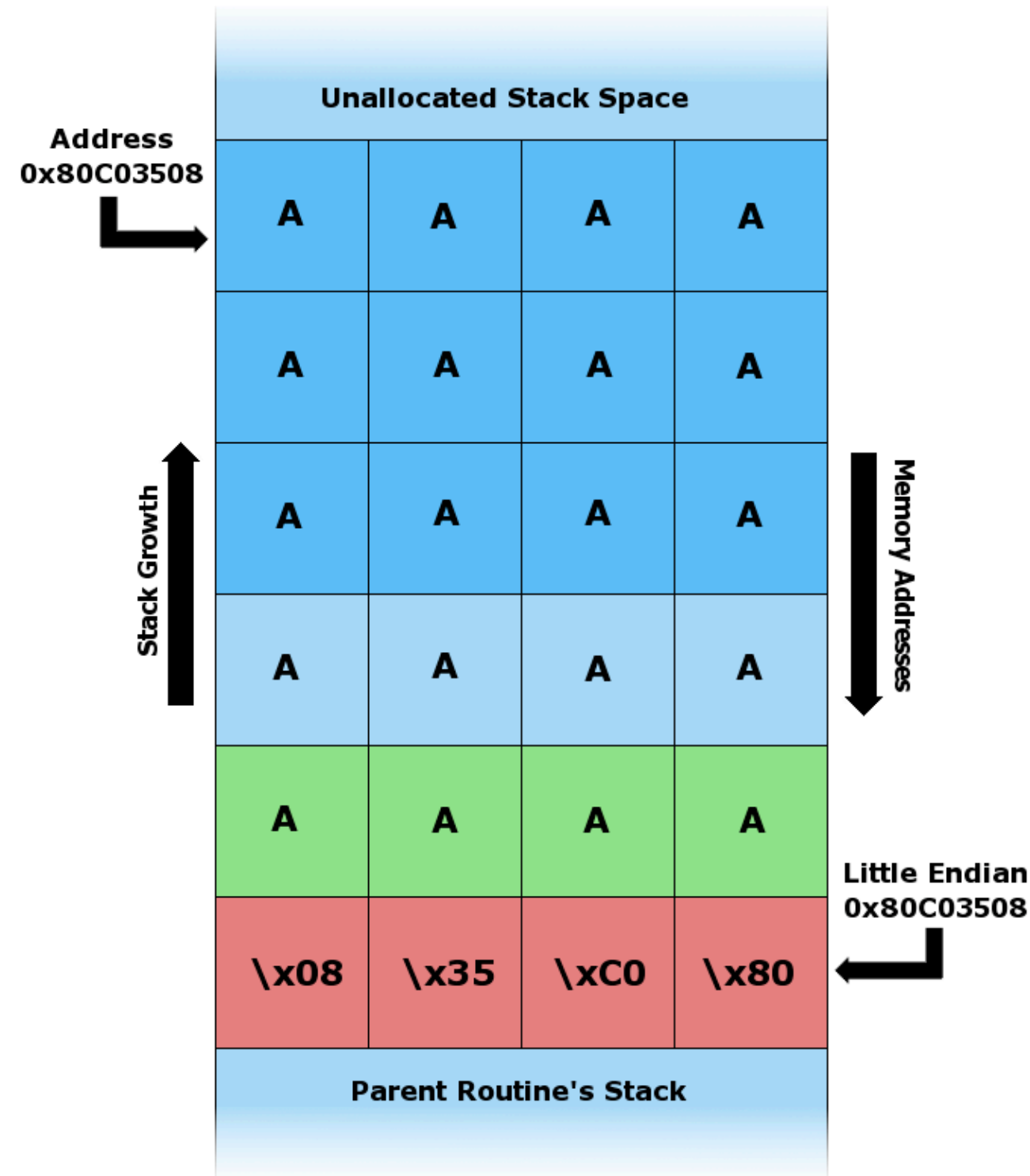


Example II

```
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar);
}

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



Protection

- Programming practices
 - Input validation, safe libraries, testing, safe languages,...
 - More at Secure Software Engineering
- **System-level Mechanisms**

Stack Canaries

- Canaries: known values placed between a buffer and control data on the stack
 - Terminator canaries
 - Contain special “termination” characters to target string functions (like strcpy())
 - Random canaries
 - Random, unpredictable to adversary, values
 - Random XOR canaries
 - Random canaries XORed with the control data
- Implemented by SSP/ProPolice GCC extensions
 - `-fstack-protector*`
 - Safer stack layout
- In some cases adversary can guess, brute-force, or read the stack

Bound Checking

- Known in other languages (e.g., `IndexError` in Python)
- C and C++ need compiler support
- Enforced at run-time
 - Keep information for each allocated block of memory
 - Check all pointers against this information

Nonexecutable Stacks

- Write XOR Execute (W^X)
 - Memory page can be either writable or executable (not both)
 - No-eXecute (NX) bit
 - mark memory as data (cannot be executed)
 - Relatively simple
 - Widely supported
 - PaX, Exec Shield, OpenBSD, DEP, ..
 - Firefox
- How to bypass it?

Bypassing NX

- Return-oriented Programming
 - Idea: Create a fake function stack frame
- Return-to-libc
 - Although NX does not allow to execute code from stack, adversary can “jump” to some standard calls (libc)

Memory Space Randomization

- Address Space Layout Randomization (ASLR)
 - Observation: adversary needs to know where executable code is located
 - Idea: randomize memory such that adversary does not know where to “jump”
 - Randomize memory segments, stack, heap, libraries, ...
 - Usually, not everything is randomized (adversary can still jump to libraries, etc...)
 - Adversary can try to brute-force for a “good” address
 - Widely implemented (Android, iOS, Linux, ...)

Other mechanisms

- Combination of techniques
 - ASLR + Position-Independent executables
 - ASLR + NX
 - ...
- Fat pointers
 - Pointers contain bounds information as well
- ...

Other Vulnerabilities

Null-termination Errors

```
1  int main(void) {  
2      char a[16];  
3      char b[16];  
4      char c[16];  
5      strncpy(a, "0123456789abcdef", sizeof(a));  
6      strncpy(b, "0123456789abcdef", sizeof(b));  
7      strcpy(c, a);  
8      /* ... */  
9  }
```

HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE?
IF SO, REPLY "POTATO" (6 LETTERS).



...s pages about "boats". User Erica requests
secure connection using key "4538538374224".
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435.
Maggie (chrome user) sends this message: "H



POTATO



...s pages about "boats". User Erica requests
secure connection using key "4538538374224".
User Meg wants these 6 letters: **POTATO**. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435.
Maggie (chrome user) sends this message: "H

SERVER, ARE YOU STILL THERE?
IF SO, REPLY "BIRD" (4 LETTERS).



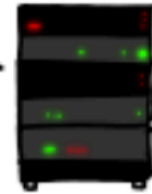
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2c0cb9ff89b-d3b-f8



HMM...



BIRD



User Olivia from London wants pages about "na
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: **BIRD**. There are currently 348
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2ccb9ff89b-d3bffa8)

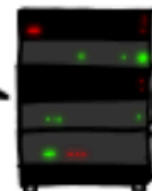
SERVER, ARE YOU STILL THERE?
IF SO, REPLY "HAT" (500 LETTERS).



a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: **HAT**. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User



HAT. Lucas requests the "missed conne
ctions" page. Eve (administrator) wan
ts to set server's master key to "148
35038534". Isabel wants pages about "
snakes but not too long". User Karen
wants to change account password to "
CoHoBaSt". User Brendan requests pages



a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: **HAT**. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User

Apple's SSL/TLS

```
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

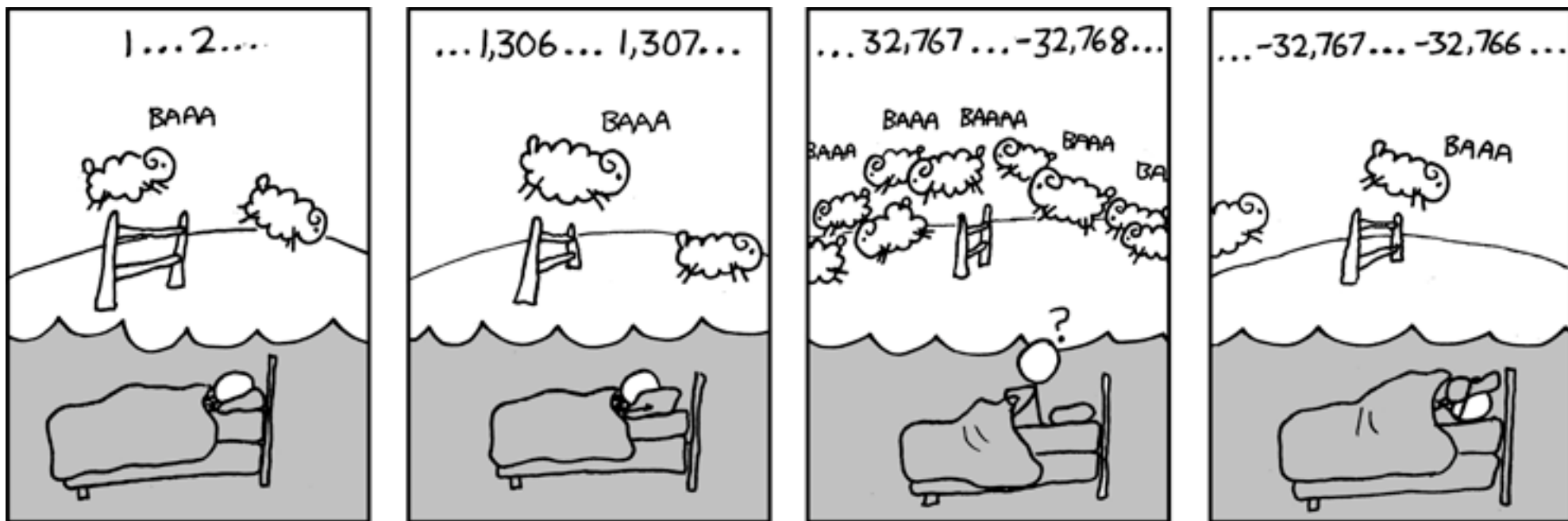
err = sslRawVerify(ctx,
                  ctx->peerPubKey,
                  dataToSign,
                  dataToSignLen,
                  signature,
                  signatureLen);
/* plaintext */
/* plaintext length */

if(err) {
    sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
               "returned %d\n", (int)err);
    goto fail;
}

fail:
SSLFreeBuffer(&signedHashes);
SSLFreeBuffer(&hashCtx);
return err;
```

Integer Overflow

- int is signed
- int a, b;
 - How to compute average of a and b?



Other vulnerabilities

- Heap overflows
- Logic errors
- Double free()
- Use after free()
- Memory leaks
- Format string bugs
- ...

Questions?

Resources

- Reading: Robert C. Seacord, Secure coding in C and C++, Addison-Wesley Professional, 2005. Chapters: 2.1, 2.2, 2.3, and 2.6 (most of the examples are from this book)
- Related wiki pages
- <https://sploitfun.wordpress.com/2015/06/26/linux-x86-exploit-development-tutorial-series/>