Attacks

Part I
Hacking in C 2018–2019
Thom Wiggers

iCIS | Digital Security
Radboud University

`$ whoami`

- Thom Wiggers
- `thom@thomwiggers.nl`
- PhD candidate working for Peter
- Project: making the internet post-quantum
  - Post-quantum TLS
  - Post-quantum SSH
  - Post-quantum bank cards
  - ...
- First *hoorcollege*
  - Need to do some teaching for the *BKO* / *UTQ*
  - Please give me feedback

Recap of last week

Programs are partitioned into different segments

- The code segment `.text` for program code
- `.data` and `.bss` for global and static variables
- These segments are usually found at the low addresses.

Recap of last week (Stack)

Stack stores local function variables

- Starts at high addresses, grows towards lower addresses
- Typically addresses start with `0x7ff` on 64-bit Linux.
- Contains return addresses, function arguments, frame pointer
- Stack is automatically managed (via stack pointer), data is gone when function returns
- Stack overflow: exceed the maximum stack size (often via recursion)

Recap of last week (Heap)

Heap for persistent or large data

- `char *x = malloc(sizeof(char));`
- Resize with `realloc()`
- Always, always check if the returned pointer is `NULL`!
- Return used memory with `free()`
- Programmer manages~~manages~~ screws up heap memory
  - Double `free()`
  - Use-after-`free()`
  - Memory leaks
  - Pointers that point to `free()`d memory
  - ...
- Use `calloc()` to non-lazily allocate zeroed memory.

Program arguments

- Remember that a program is often used with arguments:
  `./prog bla -foo ...`
- These are passed to the `main` function of your C program.
  ```
  int main(int argc, char* argv[]){
  ```
- `argc` contains the number of arguments
- `argv` is an array of character pointers (equivalent type: `char**`)
- `argv[0]` is the name of the program
  - Thus, `argc` will be at least 1!
- First command line argument will be `argv[1]`.
- Second command line argument will be `argv[2]`.
- ...

Overview

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

Table of Contents

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

Von Neumann Architecture

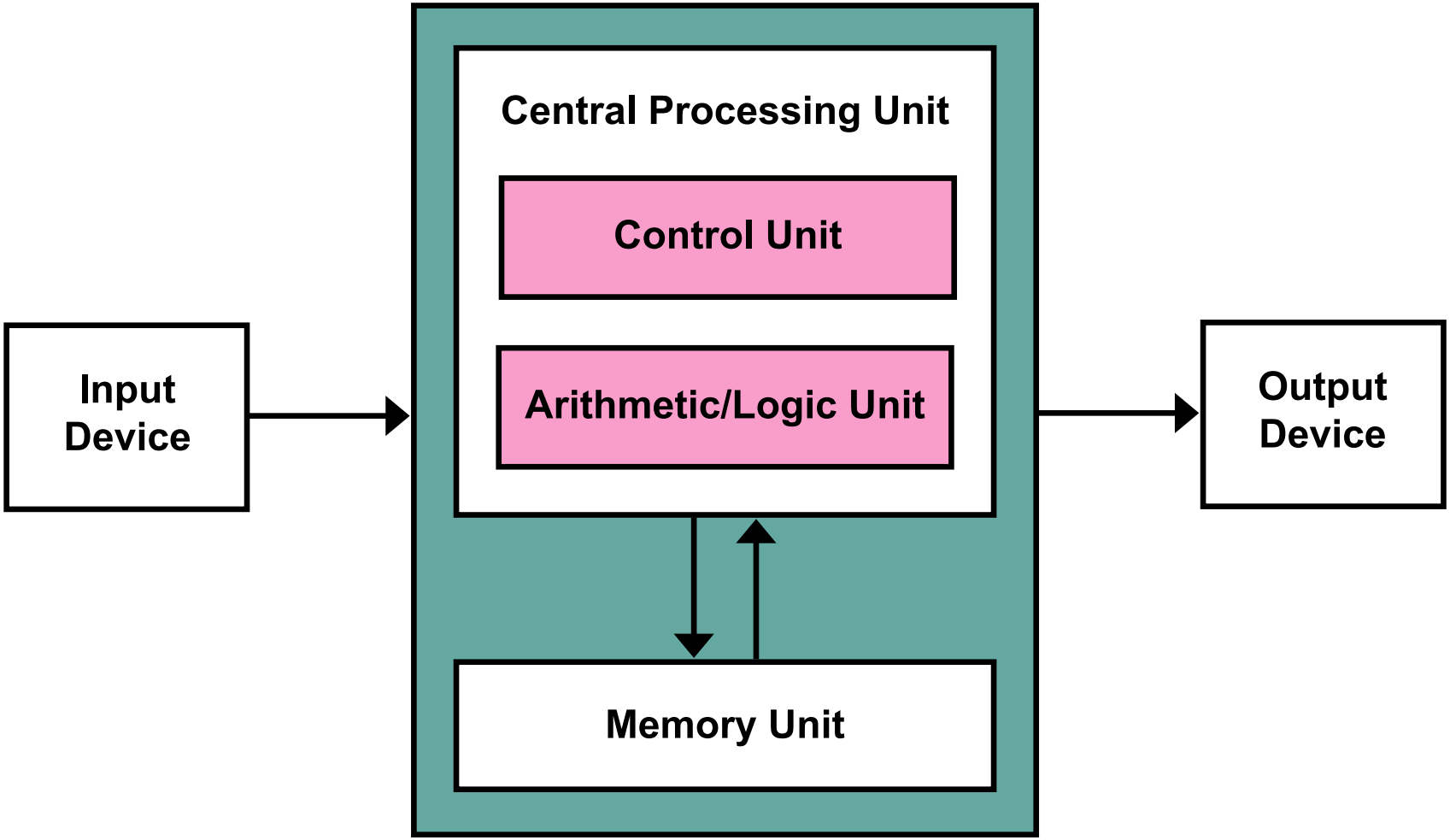


Figure: Von Neumann Architecture

(Kapooht, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=25789639)

Everything is data

- The Von Neumann architecture doesn't treat programs any different from program data!
- This means that the memory unit is shared between the code of the program and whatever the program does in memory.
- Control data such as return addresses are stored in between your program data.
- The memory bookkeeping is not just about the data of your program, but also the program itself.

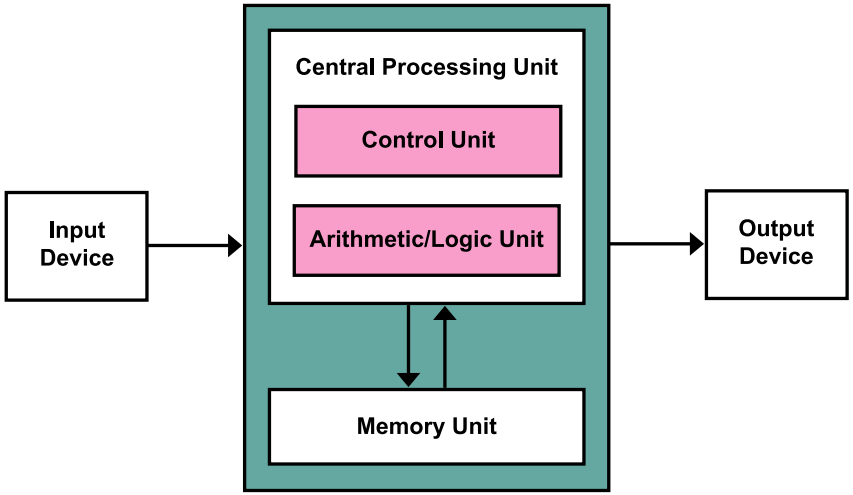


Figure: Von Neumann Architecture

(Kapooht on Wikimedia Commons, CC BY-SA 3.0)

Programs are data

So we now know that programs are controlled by what is in the same memory as the variables that we are reading and writing...
And C does not check if what we are doing to the memory makes sense...

EDITION: EU

ZDNet

CENTRAL EUROPE MIDDLE EAST SCANDINAVIA AFRICA UK ITALY SPAIN MORE NEWSLETTERS ALL

MUST READ: I like Windows 7: Why should I pay to move to Windows 10?

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

By Catalin Cimpanu for Zero Day | February 11, 2019 -- 15:48 GMT (15:48 GMT) | Topic: Security

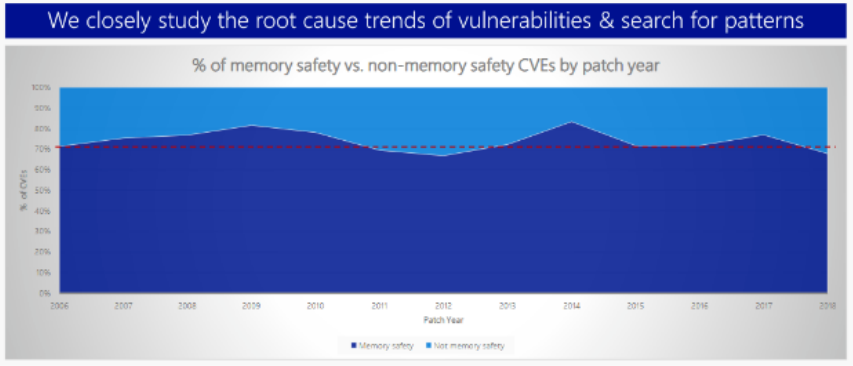


Image: Matt Miller

Around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues; a Microsoft engineer revealed last week at a security conference.

MORE FROM CATALIN CIMPANU

Security
New macOS security flaw lets malicious apps steal your Safari browsing history

Security
Dirty Sock vulnerability lets attackers gain root access on Linux systems

Security
Microsoft February Patch Tuesday fixes 77 security flaws, including IE zero-day

Security
Researchers hide malware in Intel SGX enclaves

NEWSLETTERS

ZDNet Security

Your weekly update on security around the globe, featuring research, threats, and more.

Things we will be doing at in the next weeks

- Read data from memory that we shouldn't be able to see
- Getting a program to call functions it shouldn't.
- Inject our own code into a program
- Hack into a remote machine

Table of Contents

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

Recall: `printf`

> *`int printf(const char *format, ...);`*
> *[printf] writes the output under the control of a* ***format string*** *that specifies how subsequent arguments are converted for output.* *`src: man 3 printf`*

If the attacker controls `format`, they can do a lot of nasty things.

Remember:

`%d`	Print `int` as decimal
`%u`	Print `unsigned int` as decimal
`%x`	Print `int` as hexadecimal
`%ld`	Print `long int` as decimal
`%hu`	Print `short int` as unsigned decimal
`%p`	Print variable as pointer (`void*`)
`%s`	Print string from `char*` (ie. characters until we run into `NULL`)
`%ONx`	Print as hexadecimal integer such that it's at least *N* characters wide. Fill with zeros.
`%N$x`	Print the *N*th argument of `printf` as hexadecimal integer.

Having fun with `printf`

What does the following program do *wrongly*?

```
// program.c
int main(int argc, char* argv[]) {
    // should have been printf("%s", argv[1]);
    printf(argv[1]);
}
```

```
~$ gcc -Wall -Wextra -Wpedantic -o program program.c
(gcc8 complains **only** about unused variable argc)
~$ ./program hi
hi
```

What happens if we run `./program %x`?
It will print the second argument of `printf`, even if it's not there!

Format string attacks

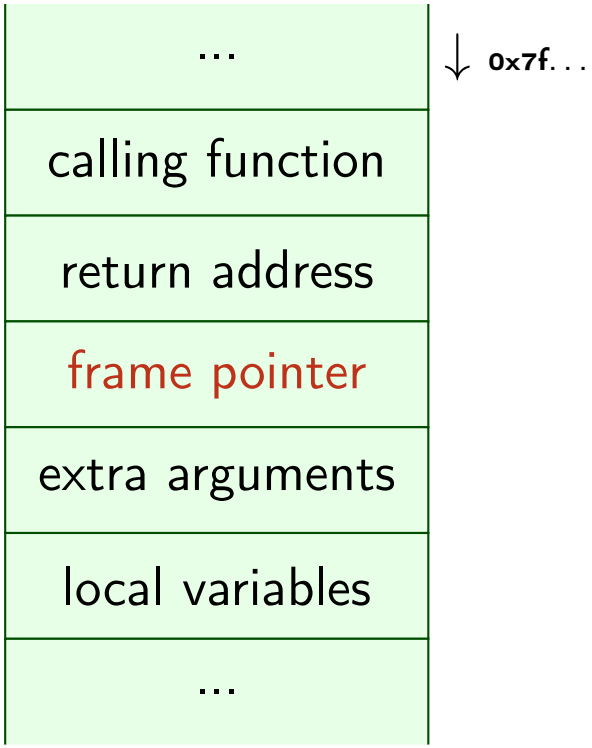
- Reading data known since 1989
- First attack that broke something in 1999
- Remember, C is from 1972!
- Allows to read data from the stack and heap.
- Easy to spot: if there is no `"` after `printf(`, it's suspicious
- If we want compiler warnings from `gcc`, we need to use `-Wformat=2`, because of course why switch this on by default.
- The clang compiler *does* report these by default.

Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a...f) in registers
4. Put the remaining arguments (g, h) on the stack.
5. Jump to the address of `func`

Why do we put arguments into registers?



Putting the first few arguments in registers saves a lot of time waiting for memory.

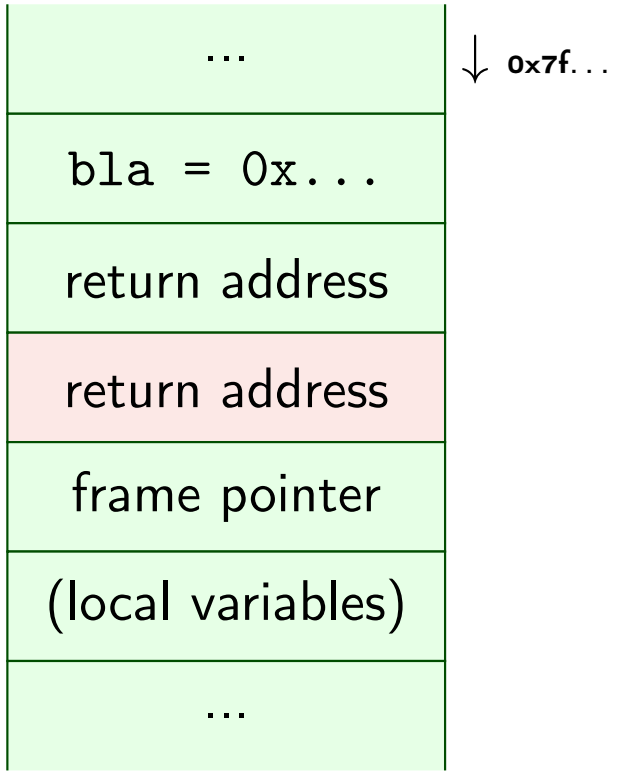
So what do we see?

- So if we run `./printf %p`, we will print the value of the second register that would contain an argument.
- If we print `./printf '%7$p'`, we will print the first 8 bytes on the stack.

printf is a powerful debugger

```
#include <stdio.h>
void do_print(char* string)
  { printf(string); }

int main(int argc, char** argv) {
  long bla = 0xDEADC0DECAFEF00D;
  do_print(argv[1]);
}
```




```
./printf "$(perl -e 'print "%p "x14')"
```

```
0x7fffffffe4e8 0x7fffffffe500 0x7ffff7f82578 0x7ffff7f83be0
0x7ffff7f83be0 (nil) 0x7fffffffe810 0x7fffffffe400 0x555555555199
0x7fffffffe4e8 0x255555050 0x7fffffffe4e0 0xdeadc0decafef00d
0x5555555551d0
```

Turning it into an arbitrary read

- If we can only read up the stack, this bug would not be as powerful as it is
- Typically, the string being input is somewhere on the stack
  - In the same range as where `printf` is reading its arguments
- Remember the %s format character: it gets the argument, interprets it as a `char*`, and reads the string at that address.
- If we put an address in the place where `printf` will read the argument from, we control where `printf` reads!

More on `printf`

Q: So know we know how to read stuff, but `printf` only displays things! We can't modify the program if we can only read things!

> *`%n` The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.* *`man 3 printf`*

Writing to arbitrary addresses

- Much like the arbitrary read, we can write data to an arbitrary place in memory.
- Again, we need the string being input somewhere up the stack, such that `printf` can read it.
- Again: `%n` writes into a `int*`
- Put an address in the place where `printf` will read the argument from, and we can control where we write!
- `%n` writes the number of characters written so far
  - Writing $\pm 2^{47}$ characters to write a 48-bit (Linux, amd64) address is *impractical* ($\pm$16 TiB).
  - **Solution:** Instead use length modifiers and write in parts: `%hn` writes 16 bits instead.

First format string exploit

```
Exploit for proftpd 1.2.0pre6

From: tymm () COE MISSOURI EDU (Tymm Twillman)
Date: Mon, 20 Sep 1999 14:31:51 -0500

Tested on Linux with standard RedHat 6.0 install (w/glibc 2.0
compatability), proftpd installed with configure/make/make install...

- ftp to host
- login (anonymous or no)

(this should be all on one line, no spaces)

ftp> ls aaaXXXX%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u
%u%u%u%u%u%u%u%u%u%653300u%n

(replace the X's with the characters with ascii values 0xdc,0x4f,0x07,0x08
consecutively)
```

https://seclists.org/bugtraq/1999/Sep/328

Table of Contents

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

In a more perfect world

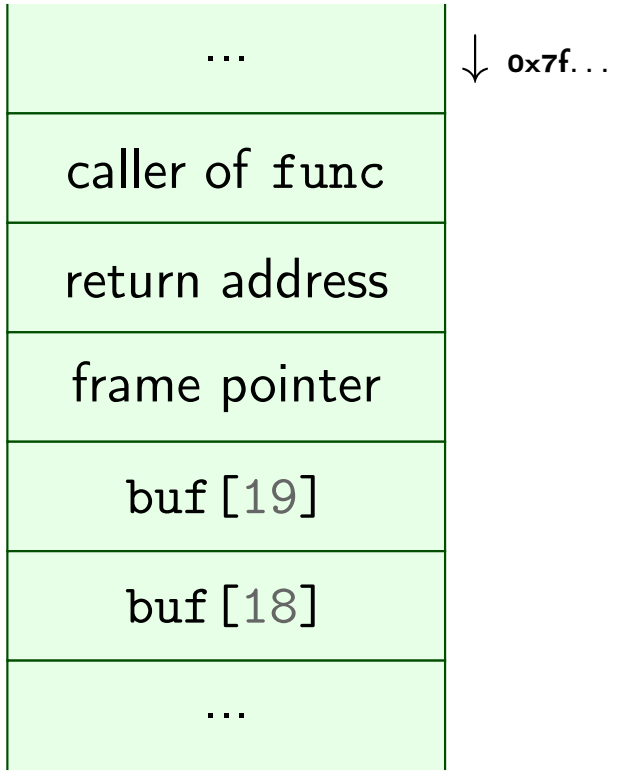
```
>>> my_list = [1, 2, 3]
>>> my_list[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Of course, the overhead of checking this and providing sensible errors to programmers is *much too big*.
Remember the last time you spent hours debugging some `segmentation error`?
If you ever face a decision to choose a programming language, please think about if you really need C(++) or if you can use a safer language such as Rust (good alternative for C), Go (good with concurrency) or Python (if you can take the performance hit).

Buffers on the stack

```
void func() {
    char buf[20];
}
```

Any C programmer quickly learns that reading `buf[20]` will happily work, but is outside of `buf`! What are we reading when we read `buf[20]`? Remember, `buf[20] == *(buf+20)`, so we read up the stack!

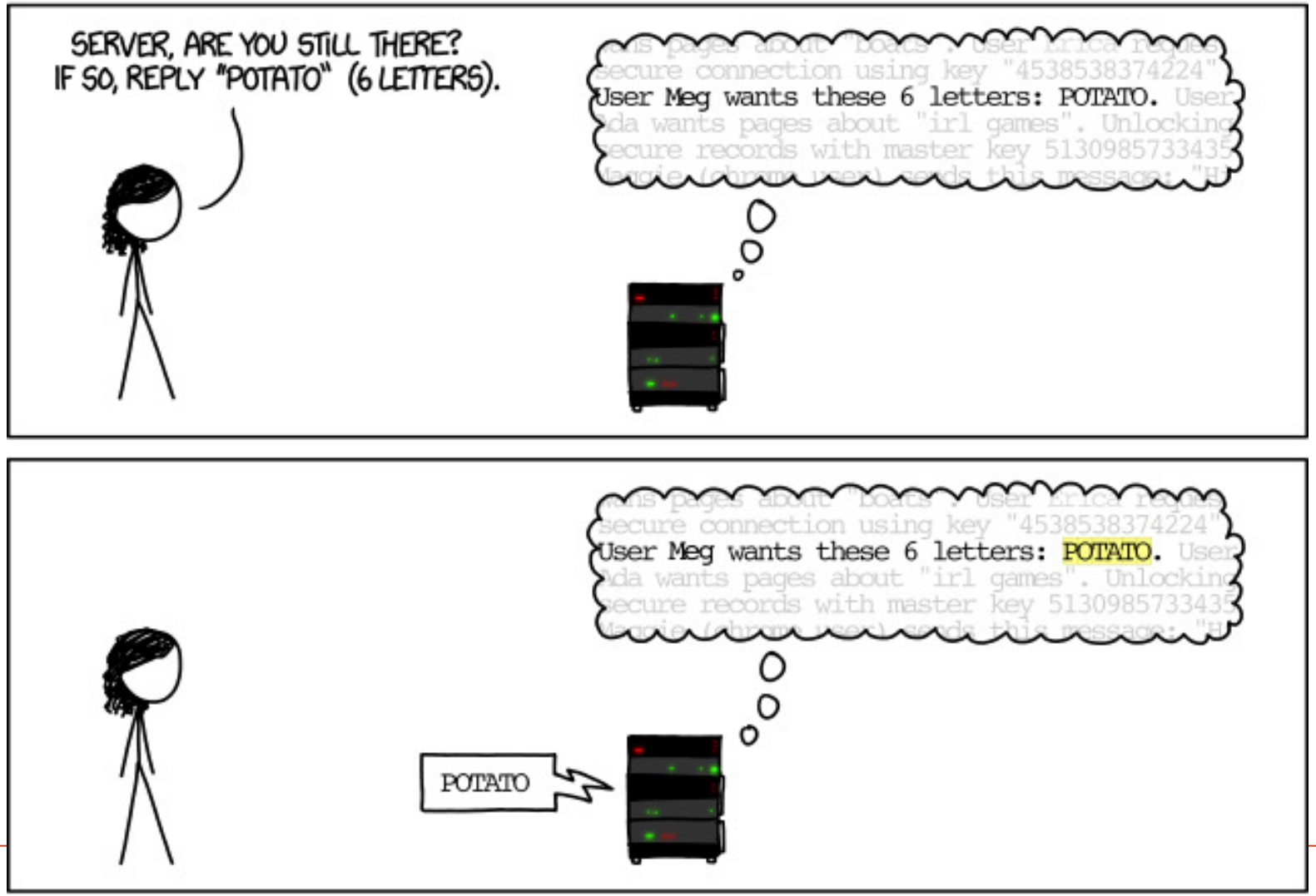


No bounds checking – what could go wrong?

- April 7, 2014, OpenSSL discloses "Heartbleed" bug
- Heartbleed allows remote attacker to read out OpenSSL memory
- Content typically includes cryptographic keys, passwords, etc.
- Bug was in OpenSSL for more than 3 years
- Introduced on December 31, 2010
- First bug with a logo, T-shirts
- Major media coverage
- Initiated major changes in OpenSSL

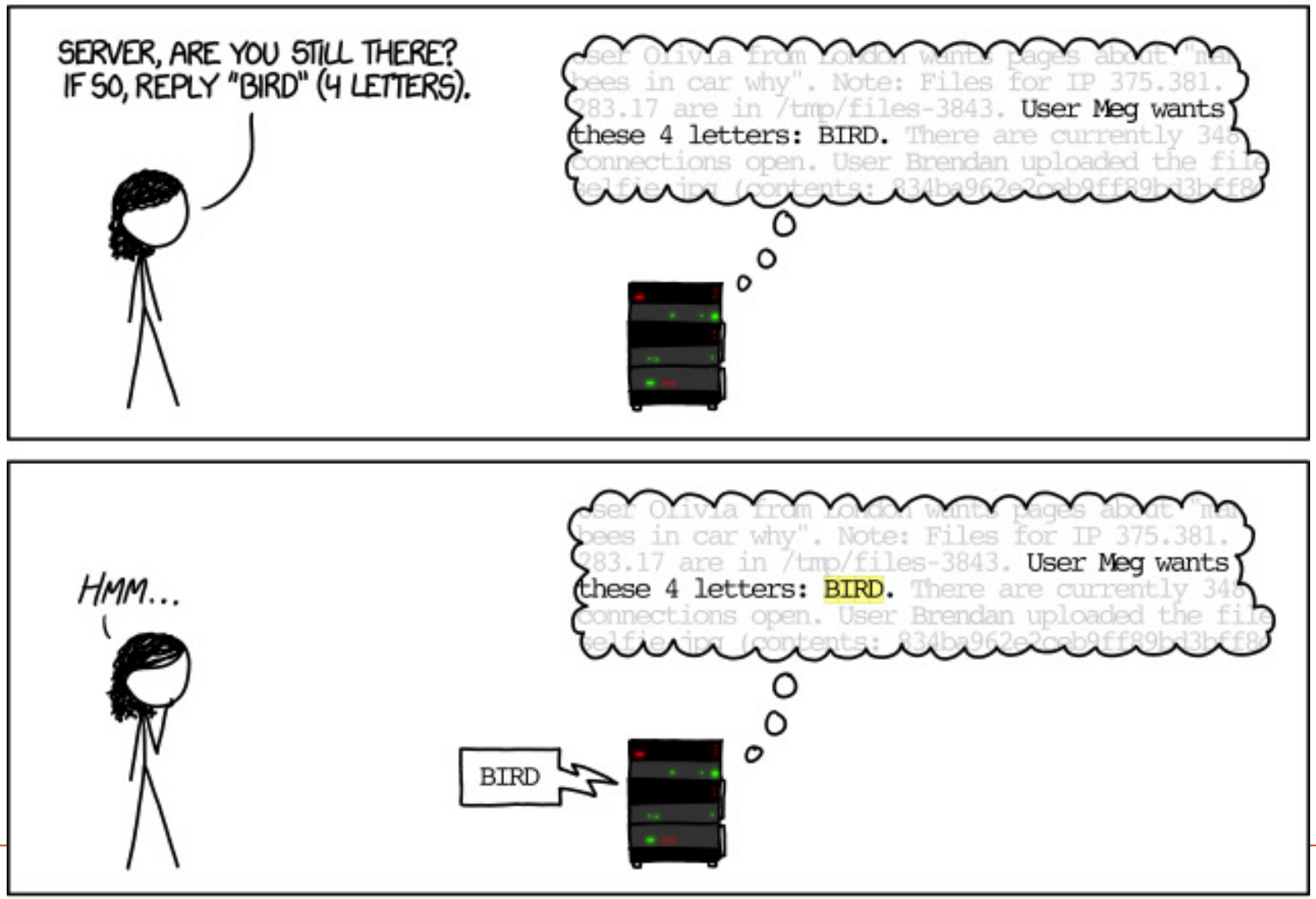
Underlying problem: Out of bounds array access in OpenSSL

How Heartbleed works



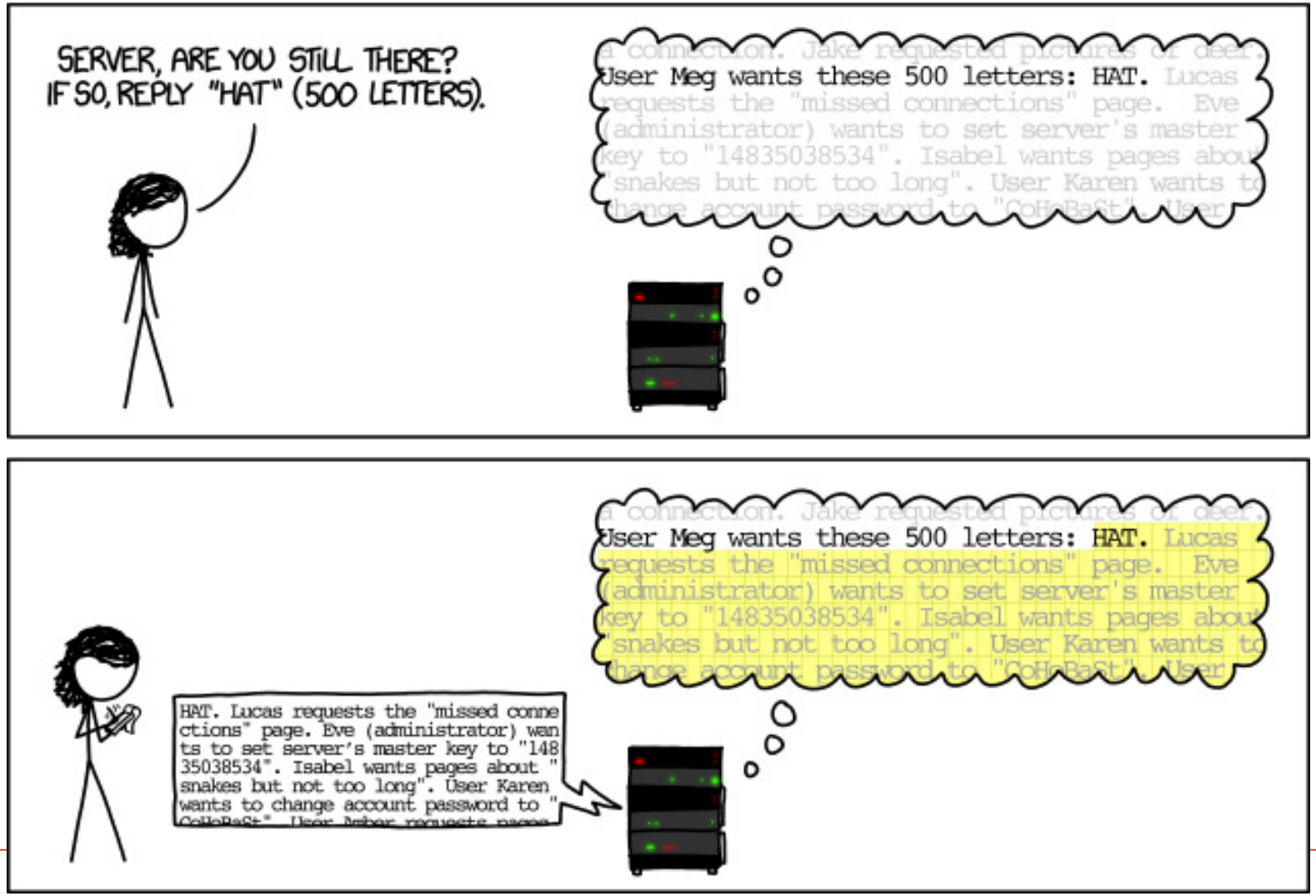
https://xkcd.com/1354/

How Heartbleed works



https://xkcd.com/1354/

How Heartbleed works



https://xkcd.com/1354/

Ping

- ping is a protocol that lets you check if a server is online and what the round-trip latency is.
- Sends an icmp packet to the server, server sends the same thing back.

```
~ $ ping -c2 10.8.0.1
PING 10.8.0.1 (10.8.0.1) 56(84) bytes of data.
64 bytes from 10.8.0.1: icmp_seq=1 ttl=64 time=15.4 ms
64 bytes from 10.8.0.1: icmp_seq=2 ttl=64 time=14.10 ms

--- 10.8.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 14.992/15.213/15.435/0.253 ms
```

Assumptions in IP

- IPv4 packets are limited to a length of 65535 bytes
- IPv4 packets get "chopped" into fragments for transportation through, e.g., Ethernet
- IPv4 header has a fragment offset
- Fragment offset + packet size must not exceed 65535
- But of course, we can forge a larger packet
- Ping of Death (mid 90s)
- Receiving host assembled the fragments into a buffer of size 65535
- Bug present in UNIX, Windows, printers, Mac OS, routers
- With some implementations of `ping`, crashing a computer was as easy as `ping -s 65510 target`
- Lessons:
  - Assume anything you get from outside your program is broken, including the specifications
  - Check if `fragment offset + packet size` $<$ 65536

IPv6

- Late 90s, early 2000s: introduction of IPv6.
- You see where this is going...
  - CVE-2013-3183: IPv6 ping of death against Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows Server 2012, and Windows RT
  - CVE-2016-1409: IPv6 ping of death against Cisco's IOS, IOS XR, IOS XE, and NX-OS software

Table of Contents

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

Why does this even work?

- The C specification contains descriptions of how things should behave
  - e.g. `i++` gives the value of `i` and increments it afterwards.
- It also defines that the behaviour of some things is undefined
  - *anything* may happen for undefined behaviour
- Undefined behaviour enables some compiler optimizations

> *Undefined behavior – behavior, upon use of a nonportable or erroneous program construct, ... for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."*
> *John F. Woods, `comp.std.c`, 1992-2-25.*

Examples of undefined behaviour

Division by zero `x / 0`
Modifying between *sequence points* `i = i++ + 1;`
Null pointer dereferencing `char *i = NULL; *i`
Use of unitialized variables `char x; printf("%c", x);`
Indexing out of bounds `char x[20]; x[21]`
Signed integer overflow Compilers may assume that `x` will never be smaller than `INT_MAX` and remove the `if` block, but `func(1)` will *probably* return a large negative number.

```
#include <limits.h>
void func(unsigned int foo) {
    int x = INT_MAX;
    x += foo;
    // probably removed:
    if (x < INT_MAX) bar();
    return value;
}
```

Never trust (user) input

- Unfortunately, we usually have to expose our software to those people who will always find ways to break it: users.
- Users will not respect your assumptions when you write your program.
- A lot of software is exposed to over 4 billion users through the internet
- User input may arrive into your program in many different ways
  - Keyboard input
  - Network packets
  - Files
  - Database content
  - *The file name of your program*: `argv[0]`

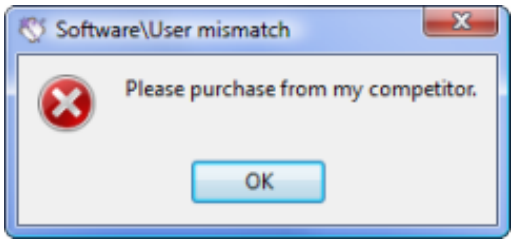


Figure: PEBKAC

How do we fix this?

- Use memory-safe languages
- If you have to use an unsafe language:
  - Turn on every warning you can.
    - `-Wall`
    - `-Wextra`
    - `-Wpedantic`
    - `-Wformat -Wformat-security`
    - `-Weverything` (Clang only)
  - Compile with run-time sanitizers:
    - `-fsanitizer=address`
    - `-fsanitizer=undefined`
  - Test with dynamic analysis tools like Valgrind
  - Check out static analysis tools that analyse at compile-time.

Table of Contents

Everything is in memory

Breaking stuff with printf

Buffer overflows

Why?
- Why does it work
- Why do we care

Homework

This week's homework

- Simple buffer overflow to corrupt memory
- Find a vulnerability using `gdb` and exploit it
  - `gdb` tutorial in the tutorial sessions on Thursday!
- Redirect a program to call a function that it shouldn't have called.

Hint about last week's homework

For the `magic_function.c` exercise:

- Draw some pictures about what's going on on the stack when you call `magic_function()`
- Make sure that the compiler doesn't remove unused variables!
  - For example, print the result to make it 'used'
  - You could try to mark a buffer as `volatile`

```
volatile char bla[1000];
```