Attacks

Part I
Hacking in C 2020
Thom Wiggers

iCIS | Digital Security
Radboud University

Notes:

Based on slides by Peter Schwabe.

Demos:

- `printf.c`
- `buffer.c`
- `print_buf.c`

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.

Notes:

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)

Notes:

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 heap memory

Notes:

The blue text is clicky.

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~~ screws up heap memory
  - Double `free()`
  - Use-after-`free()`
  - Memory leaks
  - Pointers that point to `free()`d memory
  - ...

Notes:

The blue text is clicky.

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~~ 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.

Notes:

The blue text is clicky.

Program arguments

- Remember that a program is often used with arguments:
  `./prog bla -foo ...`

Notes:

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[]){
```


Notes:

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

Notes:

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**`)

Notes:

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

Notes:

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!

Notes:

Notes:

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]`.

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]`.

Notes:

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]`.
- ...

Notes:

Overview

Everything is in memory

Breaking stuff with printf

Buffer overflows
- Heartbleed
- Ping

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

Inserting our own code

Homework
- This week
- Last week's homework

Notes:

Table of Contents

Notes:

Von Neumann Architecture

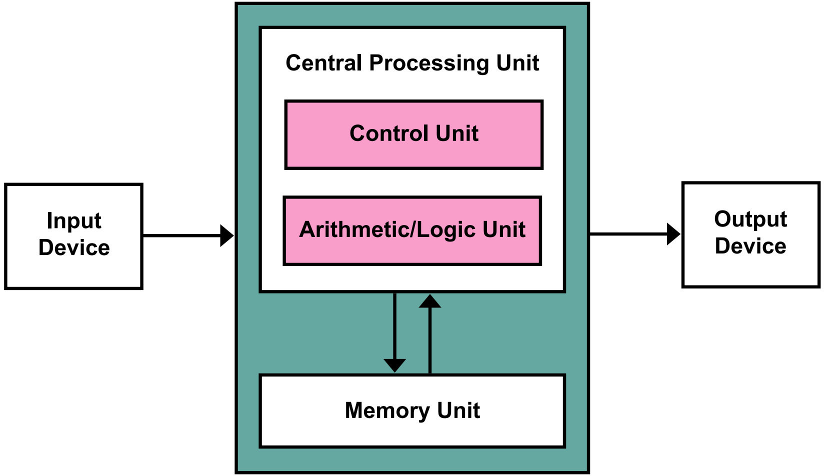


Figure: Von Neumann Architecture

Notes:

The Von Neumann Architecture is the theoretical model behind most, if not all, modern computers. It is easy to see that this model applies to your pc. It is nice and simple, and "cheap" hardware-wise.

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.

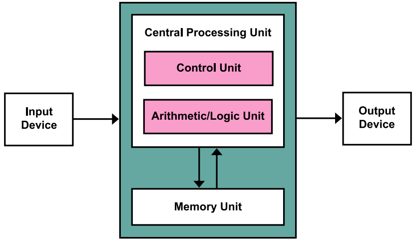


Figure: Von Neumann Architecture

Notes:

Don't yet mention self-modifying code, that's for the next slide.

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. . .

Notes:

- The foundation of the course is that if we can abuse what's happening when we modify memory in bad ways, we can then redirect the program.
- Sometimes that modifying the flow by overwriting parts of the program is a feature that is desired (and then people call it self-modifying code), but often it's a bug.
- We can even put our own code into memory, code that's not even part of the program, which we will talk about in the next lecture.
- Obviously, there are some protection mechanisms because this is all too silly, but we can turn those off.

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...

Notes:

- The foundation of the course is that if we can abuse what's happening when we modify memory in bad ways, we can then redirect the program.
- Sometimes that modifying the flow by overwriting parts of the program is a feature that is desired (and then people call it self-modifying code), but often it's a bug.
- We can even put our own code into memory, code that's not even part of the program, which we will talk about in the next lecture.
- Obviously, there are some protection mechanisms because this is all too silly, but we can turn those off.

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. . .

Notes:

- The foundation of the course is that if we can abuse what's happening when we modify memory in bad ways, we can then redirect the program.
- Sometimes that modifying the flow by overwriting parts of the program is a feature that is desired (and then people call it self-modifying code), but often it's a bug.
- We can even put our own code into memory, code that's not even part of the program, which we will talk about in the next lecture.
- Obviously, there are some protection mechanisms because this is all too silly, but we can turn those off.

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.

Notes:

- The “dirty sock” Linux vulnerability in the side bar is *not* a memory safety issue. The program was written in Go, a memory-safe language. Instead, they messed up how they parse strings, allowing an attacker to inject "I am root". (https://shenaniganslabs.io/2019/02/13/Dirty-Sock.html)
- Article: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-
- Nice follow-up blog post: https://medium.com/@sgrif/no-the-problem-isnt-bad-coders-ed4347810270.

Things we will be doing at in the next weeks

- Read data from memory that we shouldn't be able to see

Notes:

This last part will resemble how you will be graded.

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.

Notes:

This last part will resemble how you will be graded.

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

Notes:

This last part will resemble how you will be graded.

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

Notes:

This last part will resemble how you will be graded.

Table of Contents

Notes:

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`)
`%0Nx`	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.

Notes:

- The `%0Nx` syntax can be very helpful: `%02x` will for example make sure that `0xC` is printed as `0x0C`.
- The length modifiers, used for example as `%ld` or `%hu` can be used to print larger or smaller integers: e.g.
  - `hh` for `char` integers
  - `h` for `short` integers
  - `l` for `long` integers
  - `ll` for `long long` integers

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]);
}
```

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

Notes:

So what do we see again?

- 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.

Notes:

- The `%N$` syntax starts counting at 1.
- Make sure to escape or properly quote (single quotes) the $ on the shell!

So what do we see again?

- 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.
- If we want 8 bytes, zero-padded, without `0x` we can use `%016lx`.

Notes:

- The `%N$` syntax starts counting at 1.
- Make sure to escape or properly quote (single quotes) the $ on the shell!

So what do we see again?

- 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.
- If we want 8 bytes, zero-padded, without `0x` we can use `%016lx`.
- The addresses are randomized each time, because of ASLR!

Notes:

- The `%N$` syntax starts counting at 1.
- Make sure to escape or properly quote (single quotes) the $ on the shell!

So what do we see again?

- 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.
- If we want 8 bytes, zero-padded, without `0x` we can use `%016lx`.
- The addresses are randomized each time, because of ASLR!
  - Turn off ASLR in a shell using `setarch -R bash`.

Notes:

- The `%N$` syntax starts counting at 1.
- Make sure to escape or properly quote (single quotes) the $ on the shell!

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]);
}
```


Notes:

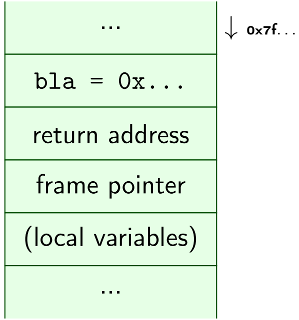
- Demo time!
- You can see the value of `bla` clearly in the output of the command on the slide.
- The return address is also in the output. One of the more significant ways to recognise this, is the fact that it doesn't start with `0x7f`, like the stack addresses.
- Demo that we can confirm this by using `gdb`.
  - `gdb -q printf.c`
  - `break do_print`
  - `run`
  - `info frame`

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')"
```



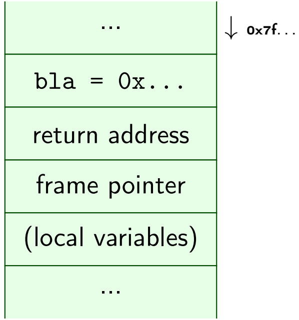
Notes:

- Demo time!
- You can see the value of `bla` clearly in the output of the command on the slide.
- The return address is also in the output. One of the more significant ways to recognise this, is the fact that it doesn't start with `0x7f`, like the stack addresses.
- Demo that we can confirm this by using `gdb`.
  - `gdb -q printf.c`
  - `break do_print`
  - `run`
  - `info frame`

printf is a powerful debugger

```c
#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
```

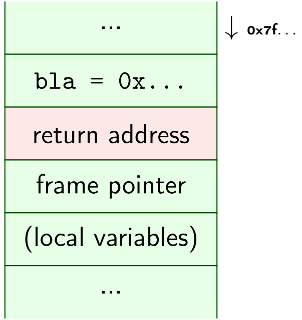

Notes:

- Demo time!
- You can see the value of `bla` clearly in the output of the command on the slide.
- The return address is also in the output. One of the more significant ways to recognise this, is the fact that it doesn't start with `0x7f`, like the stack addresses.
- Demo that we can confirm this by using `gdb`.
  - `gdb -q printf.c`
  - `break do_print`
  - `run`
  - `info frame`

printf is a powerful debugger

```c
#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
```


Notes:

- Demo time!
- You can see the value of `bla` clearly in the output of the command on the slide.
- The return address is also in the output. One of the more significant ways to recognise this, is the fact that it doesn't start with `0x7f`, like the stack addresses.
- Demo that we can confirm this by using `gdb`.
  - `gdb -q printf.c`
  - `break do_print`
  - `run`
  - `info frame`

Turning it into an arbitrary read

- If we can only read up the stack, this bug would not be as powerful as it is

Notes:

Demo time again
The 9th argument was the right one.

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

Notes:

Demo time again
The 9th argument was the right one.

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

Notes:

Demo time again
The 9th argument was the right one.

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.

Notes:

Demo time again

The 9th argument was the right one.

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!

Notes:

Demo time again
The 9th argument was the right one.

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!

Notes:

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*

Notes:

More on `printf`

Q: So know we … displays things!
We can't modif…

%n The … stored into the integer … ent. That argument s… natches the (optionally) … 3 printf

Figure: C standard library designers

Notes:

Writing to arbitrary addresses

- Much like the arbitrary read, we can write data to an arbitrary place in memory.

Notes:

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.

Notes:

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*`

Notes:

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!

Notes:

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

Notes:

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).

Notes:

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.

Notes:

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%653300u%n

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

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

Notes:

- Ask students to figure out why 653300 + stuff.'
- This attack writes zero (because of integer overflow) to some place in memory, where the current user id happens to be stored...

A note on old exploits

- This old exploit was, in many ways a lot easier to do

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- . . . But they're stored in 8 bytes

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- ...But they're stored in 8 bytes
- This means that the top two bytes are `0x0000`.

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- . . . But they're stored in 8 bytes
- This means that the top two bytes are `0x0000`.
- null bytes terminate strings!

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- . . . But they're stored in 8 bytes
- This means that the top two bytes are `0x0000`.
- null bytes terminate strings!
- Exploits using `%n` are a bit harder to pull off. . .

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- ...But they're stored in 8 bytes
- This means that the top two bytes are `0x0000`.
- null bytes terminate strings!
- Exploits using `%n` are a bit harder to pull off...
  - Overwriting the return address byte-by-byte means you'll need more than one `%n` and thus more than one address...

Notes:

A note on old exploits

- This old exploit was, in many ways a lot easier to do
- Reason: on x86 addresses were 4 bytes exactly
- On AMD64, a user-space address is 6 bytes
- ...But they're stored in 8 bytes
- This means that the top two bytes are `0x0000`.
- null bytes terminate strings!
- Exploits using `%n` are a bit harder to pull off...
  - Overwriting the return address byte-by-byte means you'll need more than one `%n` and thus more than one address...
  - If you only need to overwrite a single byte, still easy.

Notes:

Table of Contents

Notes:

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
```


Notes:

- Example is in Python, because that was just easier.
- https://rust-lang.org
- https://golang.org
- https://python.org
- There are other options, of course: don't feel limited to this list! Just make sure that you understand what your chosen environment does and does not offer.

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*.

Notes:

- Example is in Python, because that was just easier.
- https://rust-lang.org
- https://golang.org
- https://python.org
- There are other options, of course: don't feel limited to this list! Just make sure that you understand what your chosen environment does and does not offer.

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`?

Notes:

- Example is in Python, because that was just easier.
- https://rust-lang.org
- https://golang.org
- https://python.org
- There are other options, of course: don't feel limited to this list! Just make sure that you understand what your chosen environment does and does not offer.

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).

Notes:

- Example is in Python, because that was just easier.
- https://rust-lang.org
- https://golang.org
- https://python.org
- There are other options, of course: don't feel limited to this list! Just make sure that you understand what your chosen environment does and does not offer.

Buffers on the stack

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


Notes:

- Let's take a look at the memory layout, see where `buf` is located
- We will read a byte from whatever is *before* `buf`, because the first element of `buf` is at the *low* address.
- If we write too many bytes, we can overwrite the stack pointer!

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`!

Notes:

- Let's take a look at the memory layout, see where `buf` is located
- We will read a byte from whatever is *before* `buf`, because the first element of `buf` is at the *low* address.
- If we write too many bytes, we can overwrite the stack pointer!

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`!

Notes:

- Let's take a look at the memory layout, see where `buf` is located
- We will read a byte from whatever is *before* `buf`, because the first element of `buf` is at the *low* address.
- If we write too many bytes, we can overwrite the stack pointer!

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]`?

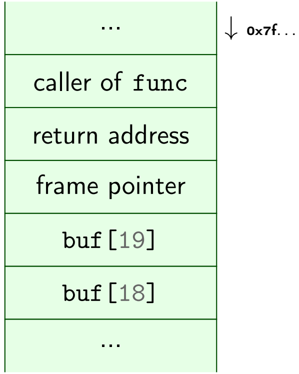
Notes:

- Let's take a look at the memory layout, see where `buf` is located
- We will read a byte from whatever is *before* `buf`, because the first element of `buf` is at the *low* address.
- If we write too many bytes, we can overwrite the stack pointer!

Buffers on the stack

```c
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!

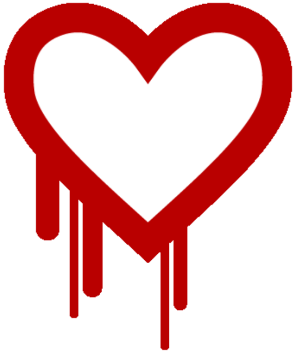


Notes:

- Let's take a look at the memory layout, see where buf is located
- We will read a byte from whatever is *before* buf, because the first element of buf is at the *low* address.
- If we write too many bytes, we can overwrite the stack pointer!

No bounds checking — what could go wrong?

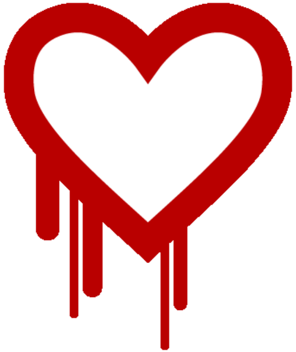
- April 7, 2014, OpenSSL discloses "Heartbleed" bug



Notes:

No bounds checking — what could go wrong?

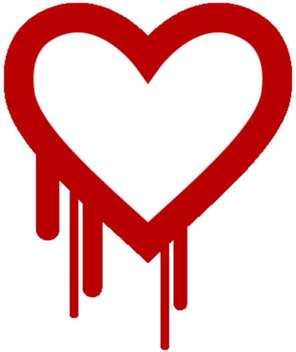
- April 7, 2014, OpenSSL discloses "Heartbleed" bug
- Heartbleed allows remote attacker to read out OpenSSL memory



Notes:

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.



Notes:

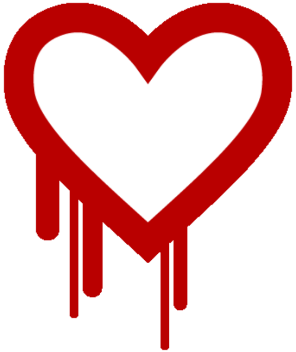
No bounds checking — what could go wrong?

- 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

Notes:

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



Notes:

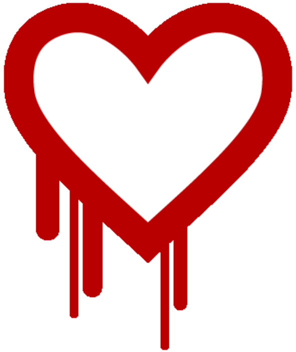
No bounds checking — what could go wrong?

- 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

Notes:

No bounds checking — what could go wrong?

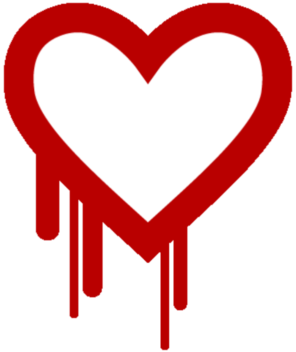
- April 7, 2014, OpenSSL discloses “Heartbleed” bug
- Heartbleed allows remote attacker to read out OpenSSL memory
- Content typically includes cryptographic keys, passwords, etc.
- Bug was in OpenSSL for more than 3 years
- Introduced on December 31, 2010
- First bug with a logo, T-shirts
- Major media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



Notes:

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



Notes:

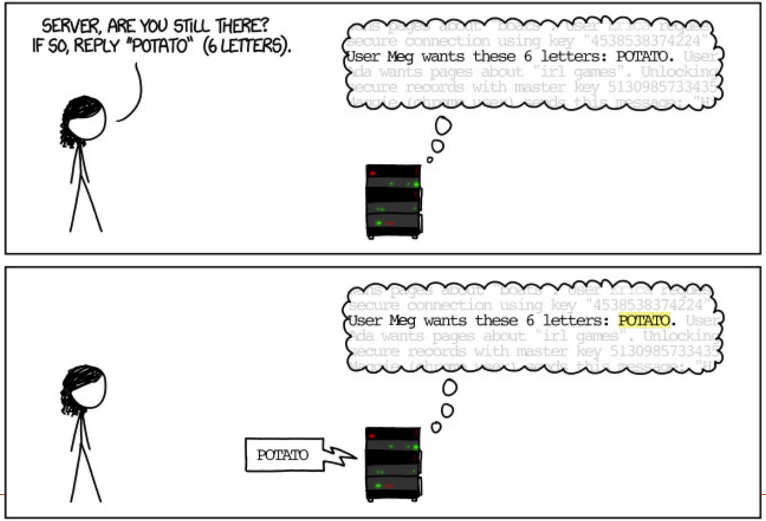
No bounds checking — what could go wrong?

- 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

Notes:

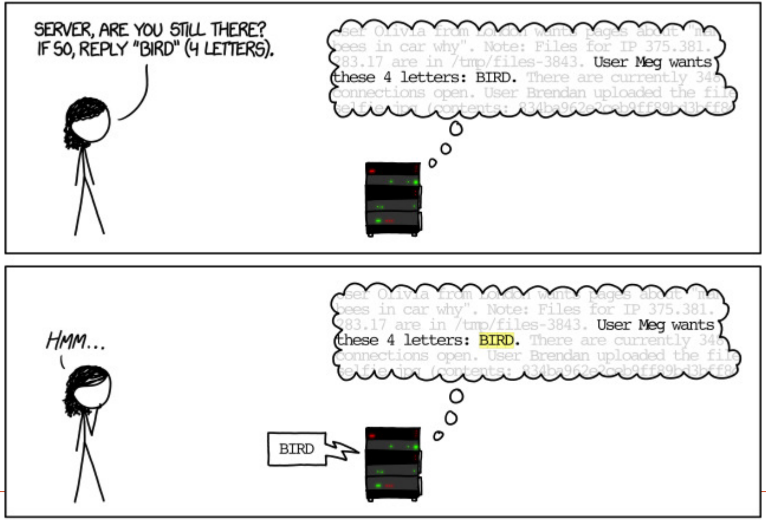
How Heartbleed works



https://xkcd.com/1354/

Notes:

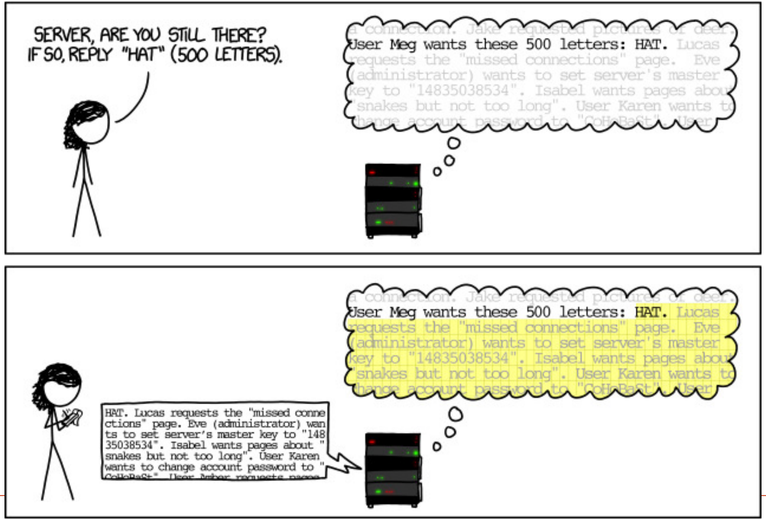
How Heartbleed works



https://xkcd.com/1354/

Notes:

How Heartbleed works



https://xkcd.com/1354/

Notes:

Ping

- ping is a protocol that lets you check if a server is online and what the round-trip latency is.

Notes:

You can't try this out on the university network, as they block ICMP. I pinged through my VPN, hence the `10.8.x.x` address.

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
```


Notes:

You can't try this out on the university network, as they block ICMP. I pinged through my VPN, hence the `10.8.x.x` address.

Assumptions in IP

- IPv4 packets are limited to a length of 65535 bytes

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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)

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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`

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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:

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

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

Notes:

- https://en.wikipedia.org/wiki/Ping_of_death

IPv6

- Late 90s, early 2000s: introduction of IPv6.

Notes:

IPv6

- Late 90s, early 2000s: introduction of IPv6.
- You see where this is going. . .

Notes:

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

Notes:

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

Notes:

Table of Contents

Notes:

Why does this even work?

- The C specification contains descriptions of how things should behave

Notes:

Undefined behaviour probably makes the lives of the people who build compilers easier – but is that worth it compared to the number of hours lost to debugging weird issues?

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.

Notes:

Undefined behaviour probably makes the lives of the people who build compilers easier – but is that worth it compared to the number of hours lost to debugging weird issues?

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

Notes:

Undefined behaviour probably makes the lives of the people who build compilers easier – but is that worth it compared to the number of hours lost to debugging weird issues?

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 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.*

Notes:

Undefined behaviour probably makes the lives of the people who build compilers easier – but is that worth it compared to the number of hours lost to debugging weird issues?

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.*

Notes:

Undefined behaviour probably makes the lives of the people who build compilers easier – but is that worth it compared to the number of hours lost to debugging weird issues?

Examples of undefined behaviour

Division by zero `x / 0`

Notes:

Notes:

Examples of undefined behaviour

Division by zero `x / 0`
Modifying between *sequence points* `i = i++ + 1;`

Examples of undefined behaviour

Division by zero `x / 0`
Modifying between *sequence points* `i = i++ + 1;`
Null pointer dereferencing `char *i = NULL; *i`

Notes:

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);`

Notes:

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]`

Notes:

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;
}
```


Notes:

Never trust (user) input

- Unfortunately, we usually have to expose our software to those people who will always find ways to break it: users.

Notes:

Remember when your mom installed all those toolbars?

`printf-filename.c`:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
 gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/

Internet stats: https://www.internetworldstats.com/stats.htm

Never trust (user) input

- Unfortunately, we usually have to expose our software to those people who will always find ways to break it: users.

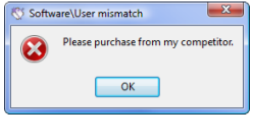


Figure: PEBKAC

Notes:

printf-filename.c:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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.

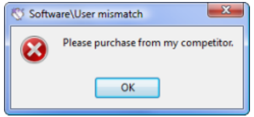


Figure: PEBKAC

Notes:

printf-filename.c:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
 gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 billion users through the internet

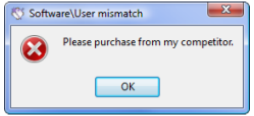


Figure: PEBKAC

Notes:

printf-filename.c:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 billion users through the internet
- User input may arrive into your program in many different ways

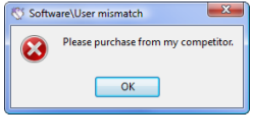


Figure: PEBKAC

Notes:

printf-filename.c:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 billion users through the internet
- User input may arrive into your program in many different ways
  - Keyboard input
  - Network packets

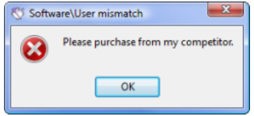


Figure: PEBKAC

Notes:

printf-filename.c:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/

Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 billion users through the internet
- User input may arrive into your program in many different ways
  - Keyboard input
  - Network packets
  - Files

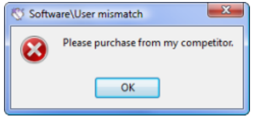


Figure: PEBKAC

Notes:

printf-filename.c:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 billion users through the internet
- User input may arrive into your program in many different ways
  - Keyboard input
  - Network packets
  - Files
  - Database content

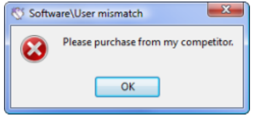


Figure: PEBKAC

Notes:

printf-filename.c:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

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 4.5 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]`

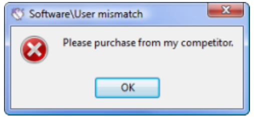


Figure: PEBKAC

Notes:

`printf-filename.c`:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf(argv[0]);
}
```

```
gcc -o "%x" printf-filename.c
./%x
./dfb03e78
```

image: https://successfulsoftware.net/2010/11/21/problem-exists-between-keyboard-and-chair/
Internet stats: https://www.internetworldstats.com/stats.htm

How do we fix this?

- Use memory-safe languages

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

How do we fix this?

- Use memory-safe languages
- If you have to use an unsafe language:

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

How do we fix this?

- Use memory-safe languages
- If you have to use an unsafe language:
  - Turn on every warning you can.

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

How do we fix this?

- Use memory-safe languages
- If you have to use an unsafe language:
  - Turn on every warning you can.
    - `-Wall`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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)

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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:

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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`

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

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 analyze at compile-time.

Notes:

- `-Weverything` will probably complain about more than what is reasonable.
  - It also only works with the Clang compiler, not with gcc.
- Clang gives better warnings in general, consider using it if you can get away with it. Unfortunately, it's not installed on the computers of the university.
- Clang also has `-Wformat-security` enabled by default!
- More information about sanitizers: https://github.com/google/sanitizers/

Table of Contents

Notes:

Inspecting a buffer with `printf`

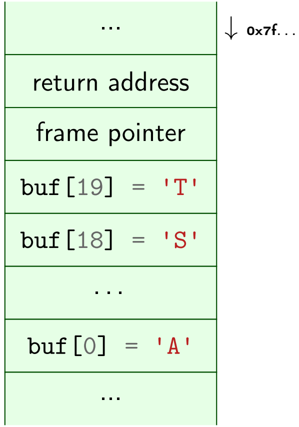
```
void func(char* string) {
    char buf[20];
    for (int i = 0; i < 20; i++)
        buf[i] = 'A' + i;
    printf(string);  // our debugger
}
int main(int argc, char* argv[]) {
    func(argv[1]);
}
```


Notes:

- **Demo** again how we can use `printf` to figure out what's going on again.
- We will extend this to become a buffer overflow attack with the found address.

Inspecting a buffer with `printf`

```c
void func(char* string) {
    char buf[20];
    for (int i = 0; i < 20; i++)
        buf[i] = 'A' + i;
    printf(string);  // our debugger
}
int main(int argc, char* argv[]) {
    func(argv[1]);
}
```



Notes:

- **Demo** again how we can use `printf` to figure out what's going on again.
- We will extend this to become a buffer overflow attack with the found address.

man gets

```
GETS(3)                     Linux Programmer's Manual                    GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets() reads a line from stdin into the buffer pointed to by s until either a termi-

       nating newline or EOF, which it replaces with a null byte ('\0').  No check for buf-

       fer overrun is performed (see BUGS below).

BUGS
       Never use gets().  Because it is impossible to tell  without  knowing  the  data  in
       advance  how  many  characters gets() will read, and because gets() will continue to
       store characters past the end of the buffer, it is extremely dangerous to  use.   It
       has been used to break computer security.  Use fgets() instead.
```


Notes:

Overflowing a buffer

```
void func() {
    char *result;
    char buf[100];
    printf("Enter your name: ");
    result = gets(buf);
    printf(result);  // our debugger
}
int main(int argc, char* argv[]) {
    func();
}
```

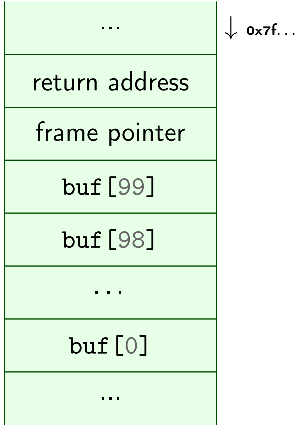

Notes:

- **Demo** `buffer-vuln.c`
  - Show how we can control the return address
  - Nice example is to overwrite it with itself to show that this works
- Make sure to run this with ASLR off: run `setarch $(uname -m) -R`!

Overflowing a buffer

```c
void func() {
    char *result;
    char buf[100];
    printf("Enter your name: ");
    result = gets(buf);
    printf(result);  // our debugger
}
int main(int argc, char* argv[]) {
    func();
}
```

```
./buffer-vuln.c:6: warning: the ‘gets’
function is dangerous and should not be
used.
```

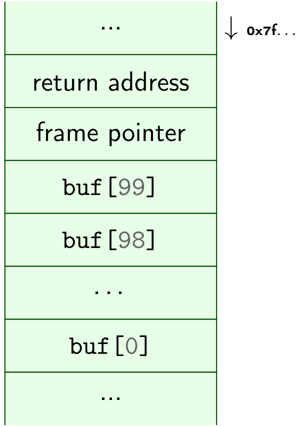


Notes:

- **Demo** `buffer-vuln.c`
  - Show how we can control the return address
  - Nice example is to overwrite it with itself to show that this works
- Make sure to run this with ASLR off: run `setarch $(uname -m) -R`!

Taking control of the return address

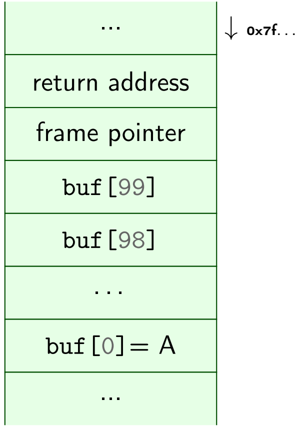
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

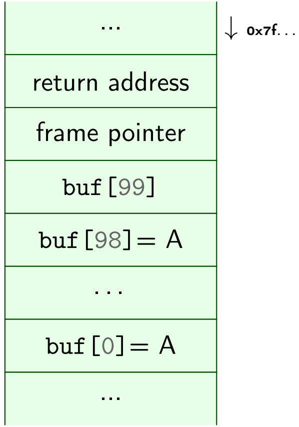
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

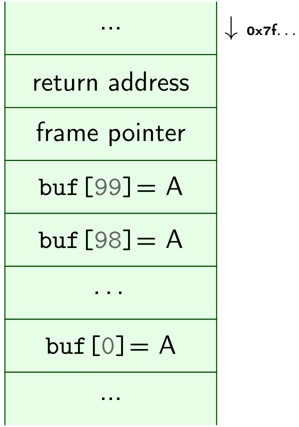
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

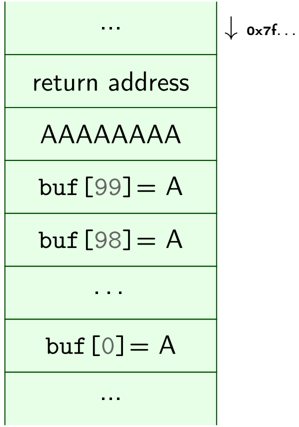
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

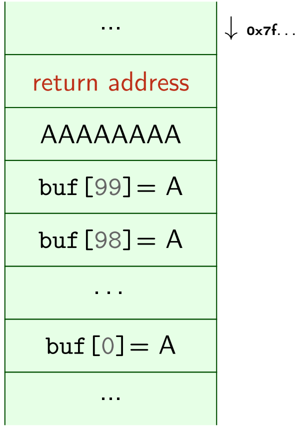
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

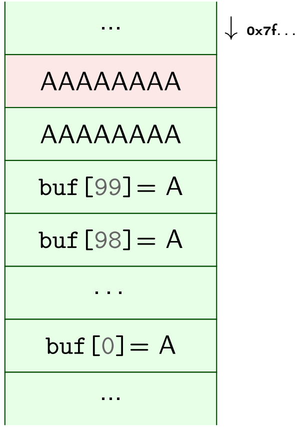
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

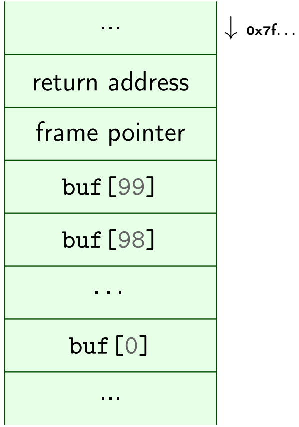
So what if we feed this program `'A'x116`?



Notes:

Taking control of the return address

So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?

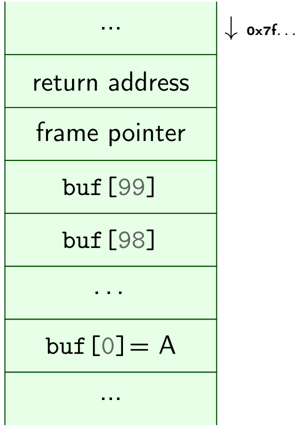


iCIS | Digital Security
Radboud University

Notes:

Taking control of the return address

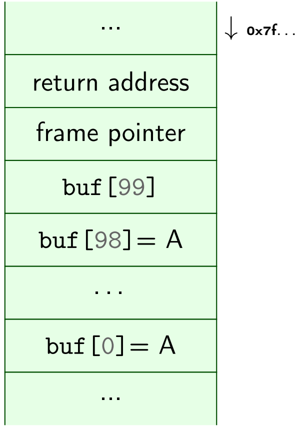
So what if we feed this program
'A'x108 +"\xDE\x0D\xDC\xAD\x0B"?



Notes:

Taking control of the return address

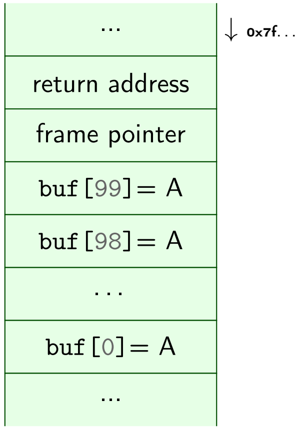
So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?



Notes:

Taking control of the return address

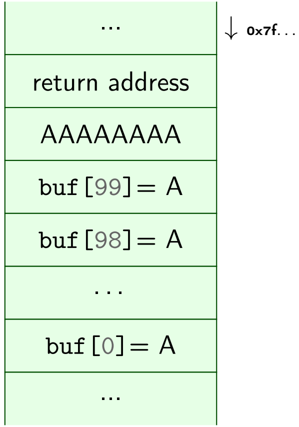
So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?



Notes:

Taking control of the return address

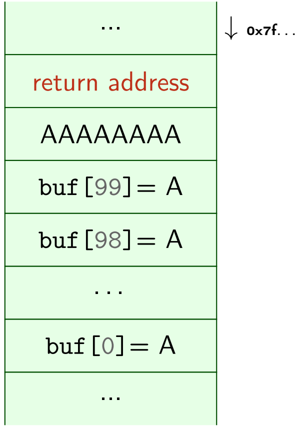
So what if we feed this program
'A'x108 +"\xDE\x0D\xDC\xAD\x0B"?



Notes:

Taking control of the return address

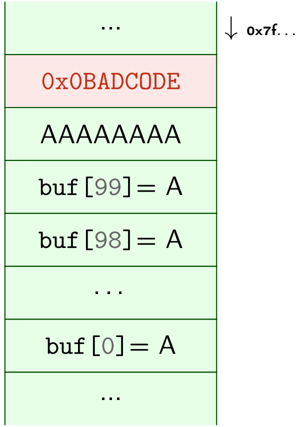
So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?



Notes:

Taking control of the return address

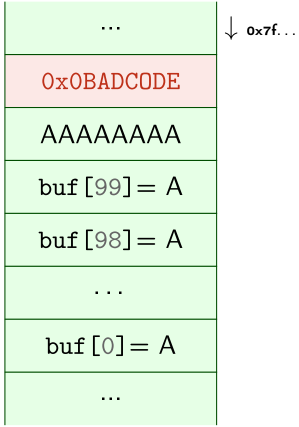
So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?



Notes:

Taking control of the return address

So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?

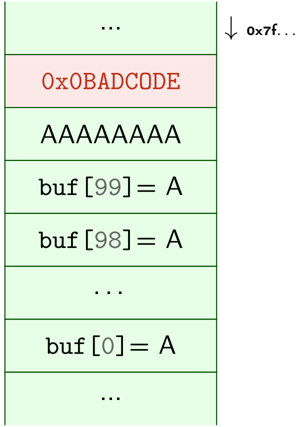


Notes:

Taking control of the return address

So what if we feed this program
`'A'x108 +"\xDE\x0D\xDC\xAD\x0B"`?

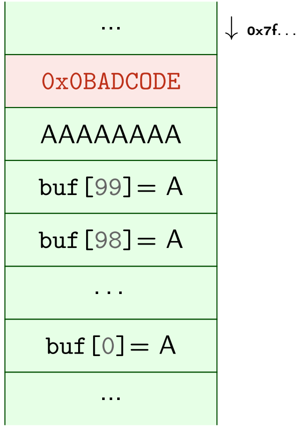
Note the endianness!



Notes:

Taking control of the return address

So what if we feed this program
`'A'x108`[1]`+"\xDE\x0D\xDC\xAD\x0B"`?



Note the endianness!

1) actual values for the offset will vary with alignment, sizes of buffers and other local variables.

Notes:

Table of Contents

Notes:

This week's homework

- Simple buffer overflow to corrupt memory

Notes:

This week's homework

- Simple buffer overflow to corrupt memory
- Find a vulnerability using `gdb` and exploit it

Notes:

This week's homework

- Simple buffer overflow to corrupt memory
- Find a vulnerability using `gdb` and exploit it
  - Use the links and follow a `gdb` tutorial!

Notes:

This week's homework

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

Notes:

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];
```


Notes:

Crashes

- Exercise 2 (`malloc`) shouldn't crash.
- Exercise 4 does crash: it's leaking memory

Notes: