CPEN 442 – Introduction to Cybersecurity

Module 2

Program Security

This material in these slides is largely taken from the "CS458: Computer Security and Privacy" course at the University of Waterloo, and it has been originally designed by Profs. Ian Goldberg and Urs Hengartner, with contributions of other instructors.

Flaws, faults, and failures

- Programs have bugs (Murphy)
- Which means security-relevant programs have bugs
- A flaw is a problem with a program.
- A security flaw is a problem that affects security in some way:
  - Confidentiality, integrity, availability
- Flaws come in two types: faults and failures
- A fault is a potential problem
  - An error in the code, data, specification, process, etc.
- A failure is when something actually goes wrong
  - "Goes wrong" means a deviation from the desired behavior, not necessarily from the specified behavior!
  - The specification itself might be wrong!

As in Module 1, do not stress; there will not be trick questions about these

Finding and fixing faults

- How do you find a fault?
  1. Unintentionally: if a user experiences a failure, you can try to work backwards to uncover the underlying fault
  2. Intentionally: you can try to cause failures, then proceed as above!
     - Think like an attacker!

- Once you find a fault, fix it:
  - By making small edits (patches) to the program
  - This is called "penetrate and patch"
  - Microsoft's "Patch Tuesday" is a well-known example

Problems with patching

- Patching sometimes makes things worse!
- Why?
  - Pressure to patch is often high, causing a narrow focus on the observed failure (are we fixing the underlying problem?)
  - The fault may have caused other, unnoticed failures, and a partial fix may cause inconsistencies or other problems.
  - The patch itself usually adds new code, which may introduce new faults, here or elsewhere!
- Alternatives?
  - Security-by-design, and not as an add-on

Unexpected behavior

- When a program behavior is specified, the spec usually lists the things the program must do:
  - For example, what should the spec for the `ls` command be?
  - "list the names of files in the directory whose name is given on the command line, if the user has permissions to read that directory"
- However, it usually does not specify what it must not do:
  - A programmer could think: "sorting the list of filenames alphabetically before outputting them is fine…"
  - This is an additional thing that is not in the spec, and this could be bad!
    - "after displaying the filenames, post the list to a public website"
    - "after displaying the filenames, delete the files"
- When implementing a security or privacy relevant program, you should consider "and nothing else" to be implicitly added to the spec
  - If you cannot implement the program because of this, the specification might be wrong (incomplete)

Module overview: Taxonomy of security flaws

Security flaws

Unintentional

Buffer overflows
Integer overflows
Format string vulnerabilities
Incomplete mediation
TOCTTOU errors

Part 1 (today)

Intentional

Malicious

Virus
Worm
Trojan
Logic bomb

Web bugs
Backdoors
Salami attack
Privilege escalation
Rootkits
Key loggers
Interface illusions

Part 2

Non-malicious

Covert channels
Side channels
Debugging backdoors
Faulty design specifications

Part 3

Security Controls

Part 4

CPEN 442 – Introduction to Cybersecurity

Module 2 – Program Security

Part I – Unintentional Flaws

Unintentional Security Flaws

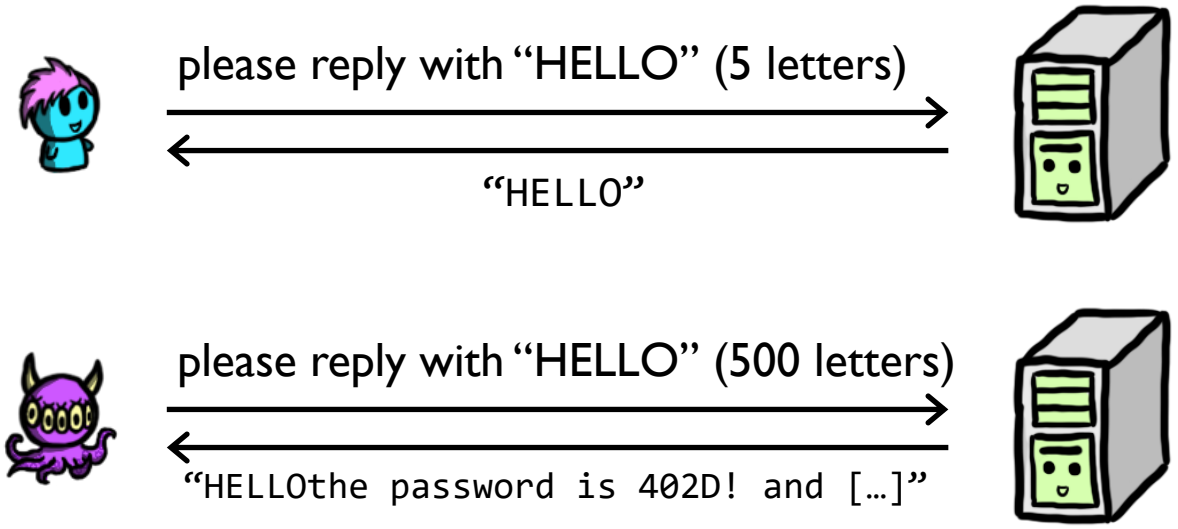
We will see the following unintentional security flaws:

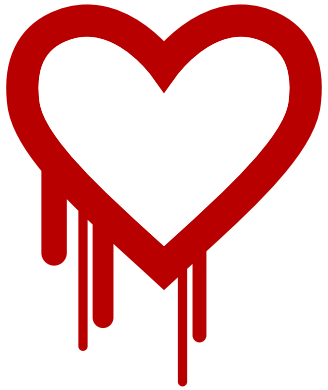
- Buffer overflows
- Integer overflows
- Format string vulnerabilities
- Incomplete mediation (and consequences, like XSS)
- TOCTTOU errors

- But let's start with two examples of unintentional flaws (that lead to failures)!

Example I: the Heartbleed Bug in OpenSSL (April 2014)

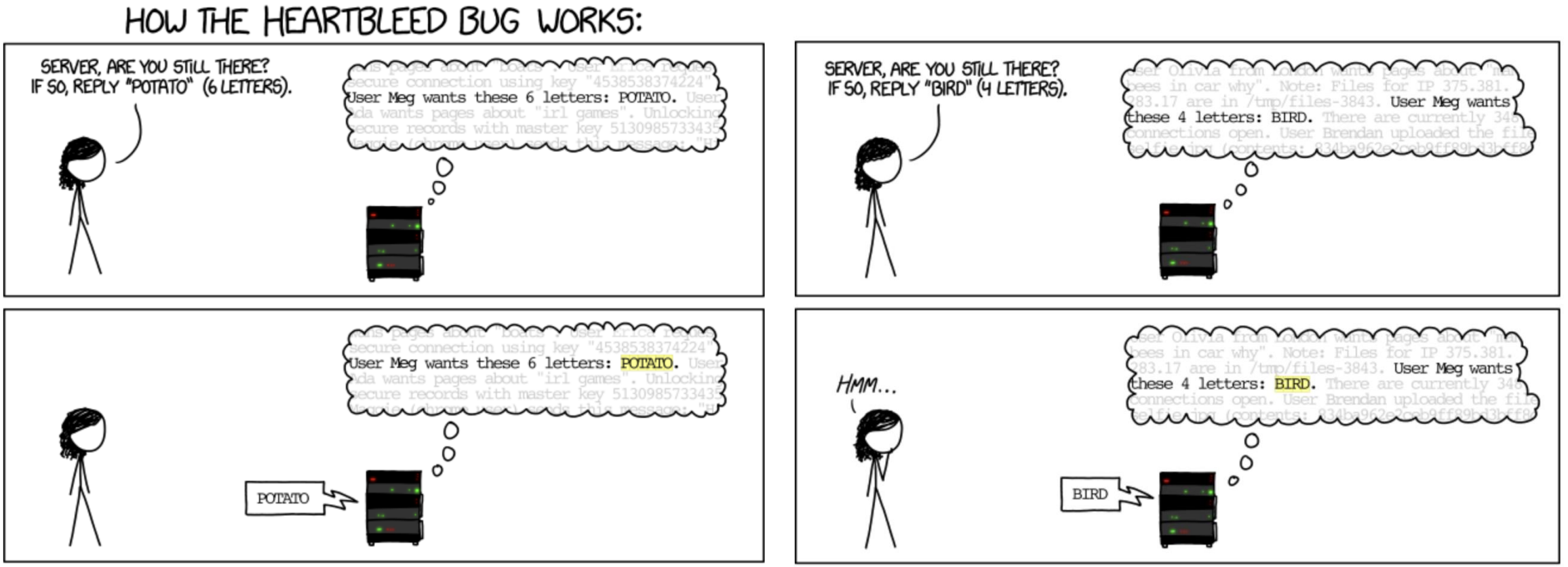
- Affected web servers all over the world
- The TLS Heartbeat mechanism is design to keep SSL/TLS connections alive even when no data is being transmitted
- The client tells the server "please reply to me with this exact string (which is of this length)"
- Lesson here: when designing a protocol, never include redundant info!





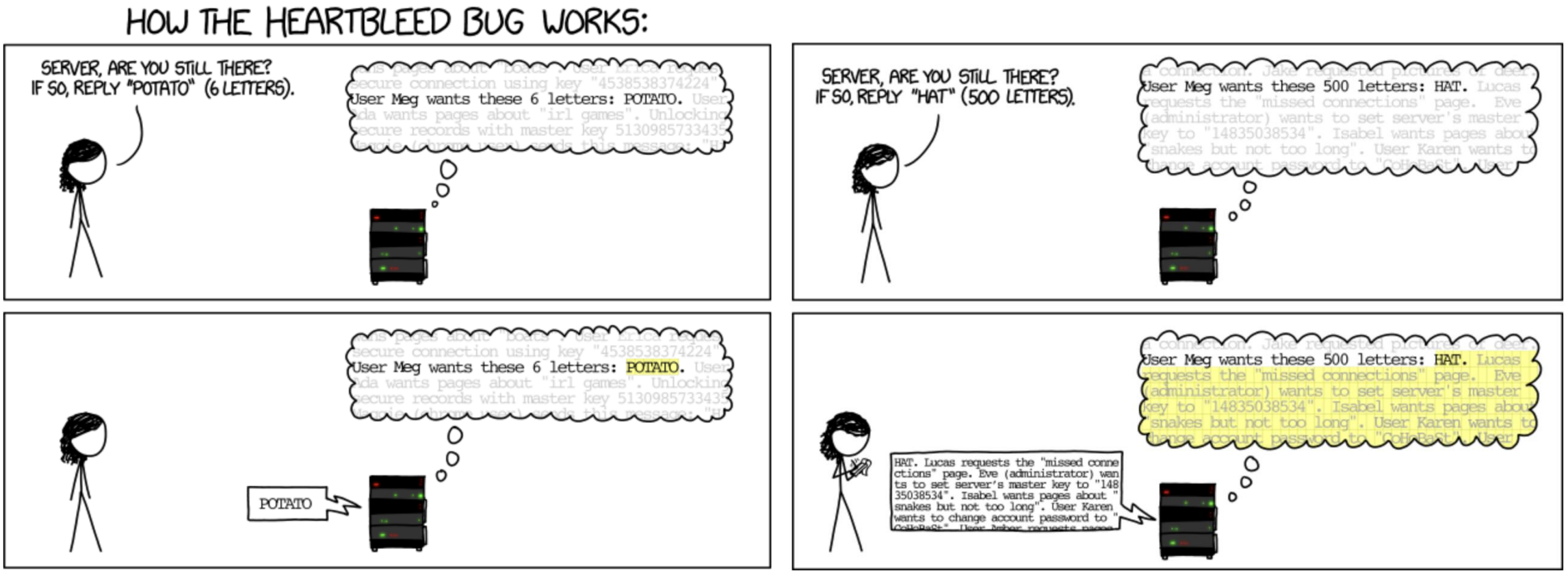
The Heartbleed bug (by xkcd)

https://xkcd.com/1354/



The Heartbleed bug (by xkcd)

https://xkcd.com/1354/



Example II: Apple's SSL/TLS Bug (Feb 2014)

- The bug occurs in code that is used to check the validity of the server's signature on a key used in an SSL/TLS connection (OSX 10.9 and iOS 6.1 and 7.0)

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

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

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

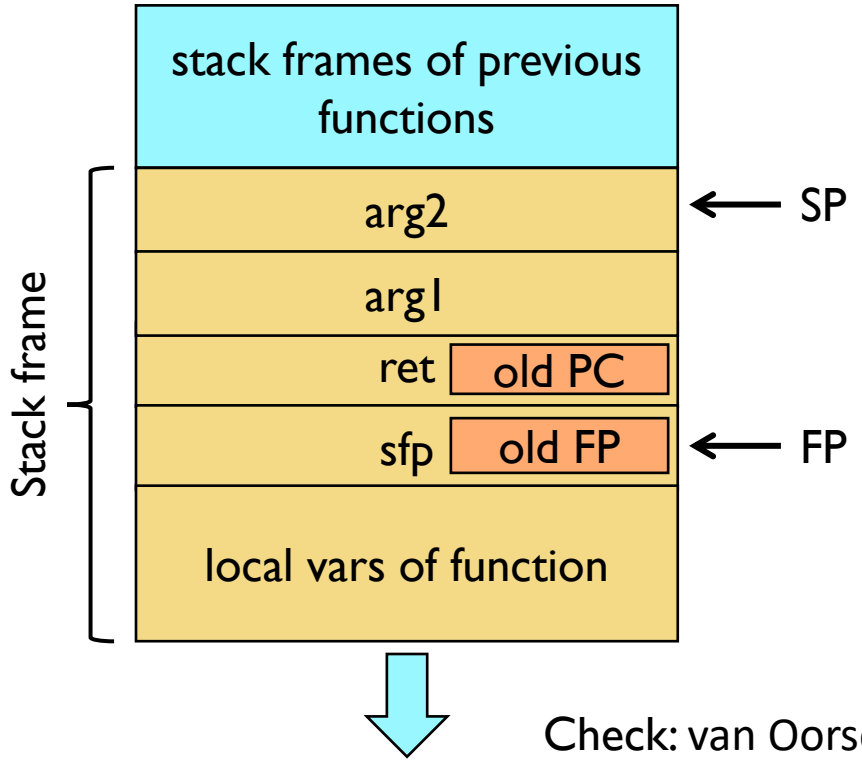
What is wrong here?

- An active attacker could potentially exploit this flaw to get a user to accept a counterfeit key that was chosen by the attacker

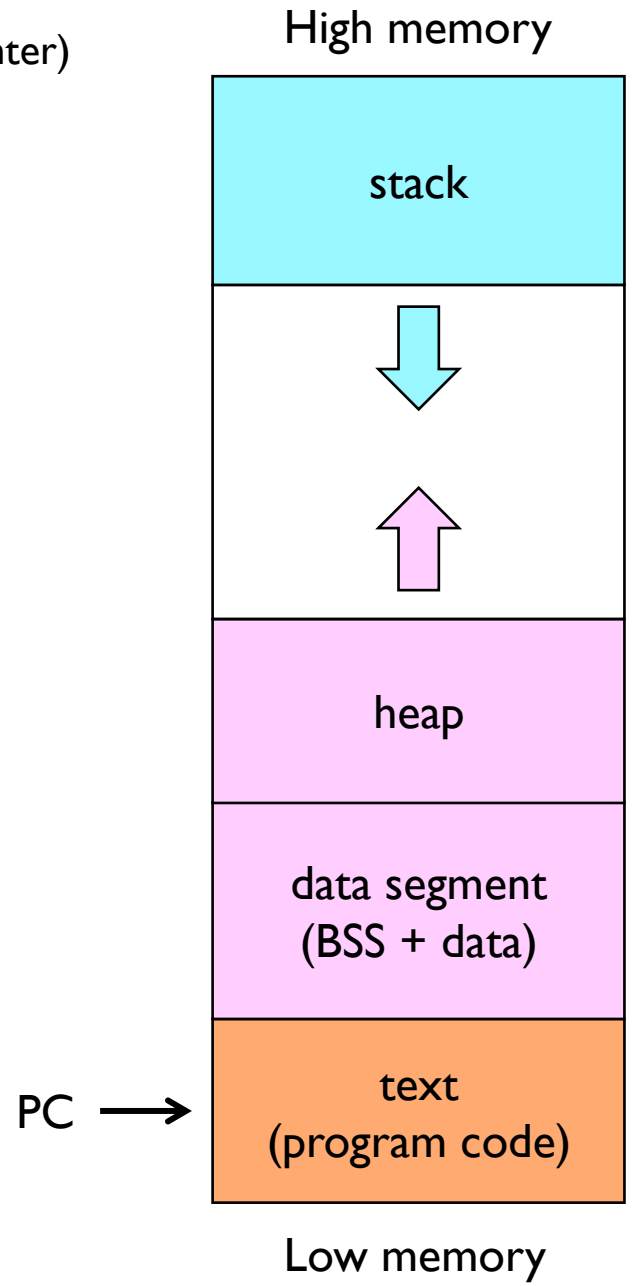
Buffer overflows

PC (Program Counter) = IP (Instruction Pointer)
SP = Stack Pointer
FP = Frame Pointer
old PC = PC before running this function
old FP = FP of previous stack frame

- What does the memory layout of a process look like? (x86)
- What happens when a function is called?



1. push args onto the stack
2. push PC as return address
3. push FP for later recovery, and set FP to that address
4. make space for local vars
5. called function executes (FP+k for args, FP-k local vars)
6. called function cleans up stack, we go back to ret (PC ← PC old) and SP goes back to previous state



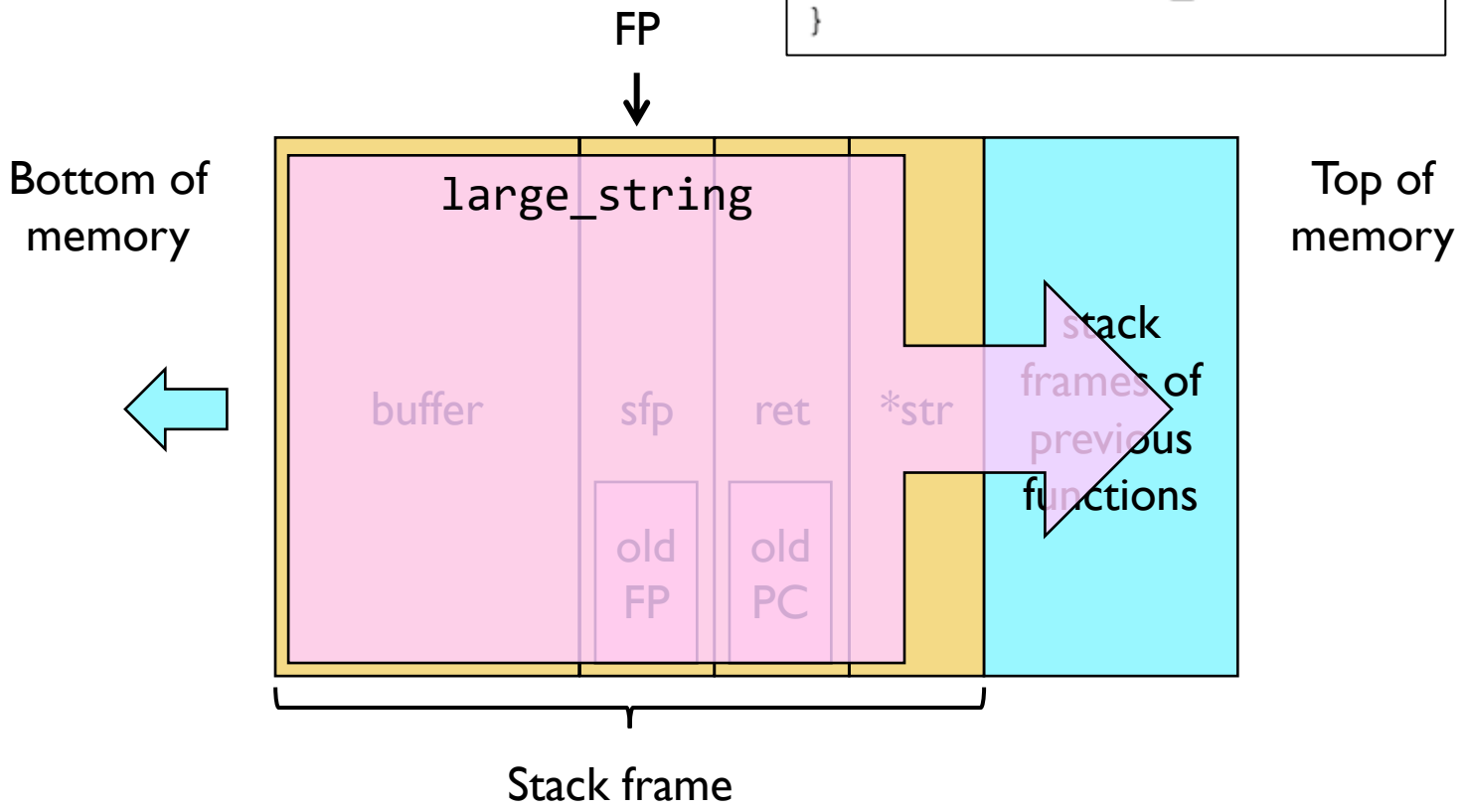
Check: van Oorschot textbook, Chapter 6, https://people.scs.carleton.ca/~paulv/toolsjewels.html

Buffer overflows

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```

```
void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

- What can go wrong?
  - `strcpy` copies until a null character is found on the string… we can "spill" outside of `buffer`
  - There's many other "dangerous functions", e.g., `gets`
- If we modify `ret`, after the function finishes, we can make the program jump to any address!!
  - usually it causes a segmentation fault
  - can we use this maliciously?



Example from: "*Smashing The Stack For Fun And Profit*" by *Aleph One* **(recommended read!)**

Buffer overflows

- We can have the malicious code in the stack!

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                           |
           |___________________________|
top of                                                    bottom of
stack                                                        stack
```

Example from: "*Smashing The Stack For Fun And Profit*" by *Aleph One* **(recommended read!)**

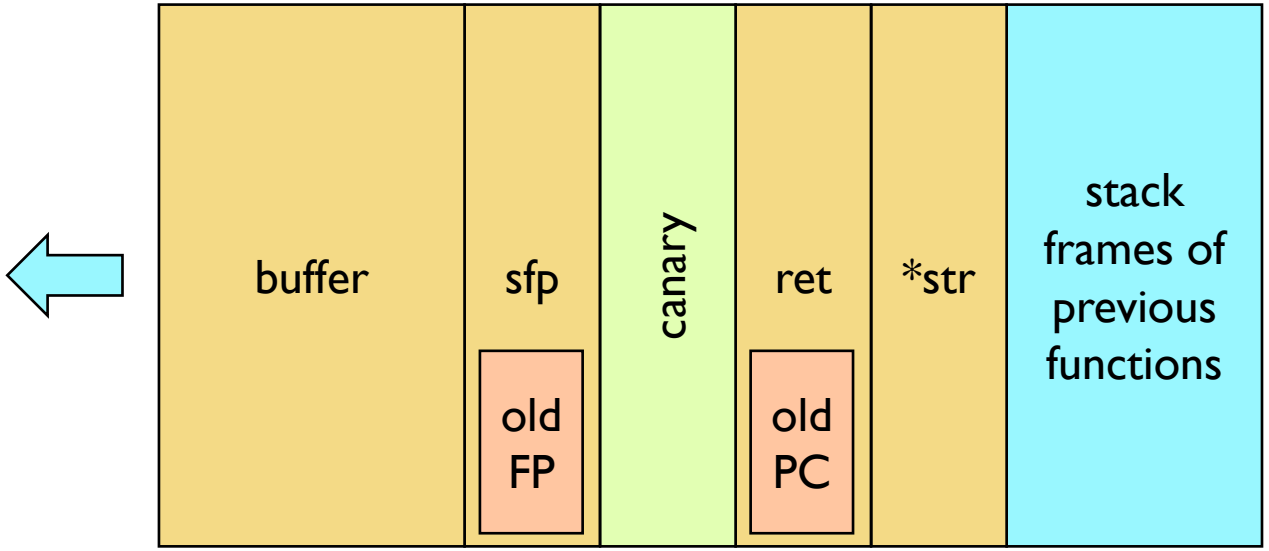
- C, the most used language for systems programming, does not raise an exception here, it continues in its merry way
- Usual targets: programs on a local machine that run with setuid (superuser) privileges
- This was a simplified description of buffer overflows, in practice there's many unpredictable factors, and requires trial and error

Kinds of buffer overflows

- In addition to the classic attack which overflows a buffer on the stack to jump to shellcode, there are many variants:
  - Attacks which work when a single byte can be written past the end of the buffer (often cause by a common off-by-one error)
  - Overflows of buffers on the heap instead of the stack
  - Jump to other parts of the program, or parts of standard libraries, instead of shellcode
    - e.g., jump inside an if statement, skipping the if check
  - More elaborate examples: Return-Oriented Programming (ROP)

Defenses against buffer overflows

- *Programmer*: use a language with bounds checking
- *Compiler*: place padding between data and return address ("Canaries")
  - Random number is placed in memory just before the stack return pointer
  - Before return, check if this number has been changed, to detect if an overflow happened



Defenses against buffer overflows

- *Programmer*: use a language with bounds checking
- *Compiler*: place padding between data and return address ("Canaries")
  - Random number is placed in memory just before the stack return pointer
  - Before return, check if this number has been changed, to detect if an overflow happened
- *Memory*: non-executable stack (Data Execution Prevention – DEP)
  - Memory page is either writable or executable, but never both
- *OS*: Stack (and sometimes code, heap, libraries) at random virtual addresses for each process
  - Address Space Layout Randomization (ASLR)
  - All mainstream OSes do this now
- *Hardware assistance*: pointer authentication, shadow stack, memory tagging

Example: Twilight Hack

- Example in The Legend of Zelda: Twilight Princess, a popular game for the Nintendo Wii
- Someone modified a save file for the game, setting Link's horse name to something much longer than what the game would usually allow.
- The name would contain a small program.
- Loading the save would trigger an overflow, and the memory positions after the horse's name would be the next ones to be executed.
- This allowed to run custom software (and "hack" the Wii)

Integer overflows

- Machine integers can represent only a limited set of numbers, which might not correspond to a programmer's mental model.

  - Integer abstraction: a number that is not a fraction, arbitrarily large
  - Machine: it has some size limits

- The program assumes that an integer is always positive; an overflow will make a (signed) integer wrap and become negative, which will violate the assumption:
  - Program casts large unsigned integers to signed integers
  - The result of a mathematical operation causes overflow
- An attacker can pass values to the program that will trigger this overflow.
- Example: a value overflow incident in bitcoin

```
int main() {
  int a = INT_MAX;
  printf("a = %d\n", a);
  a = a + 1;
  printf("a = %d\n", a);
  return 0;
}
```

Output

a = 2147483647

a = -2147483648

Format string vulnerabilities

- This class of vulnerabilities was discovered in the 2000s
- It happens when an unfiltered user input is used as format string in printf(), fprintf(), sprint(),…
- printf(buffer) instead of printf("%s", buffer)
  - The first one will parse `buffer` for %'s, and use whatever is currently on the stack to process the found parameters
- printf("%s%s%s%s") likely crashes your program
- printf("%x%x%x%x") dumps part of the stack
- %n will write to an address found on the stack
- See more examples: Exploiting Format String Vulnerabilities

Incomplete mediation

- Inputs to programs are often specified by untrusted users
  - web-based applications are a common example of this
- Users sometimes mistype data in web forms:
  - phone numbers: 6048222111
  - email address: simon.oya#ece.ubc.ca
- The web application needs to ensure that what the user has entered constitutes a meaningful request.
- This is called mediation.

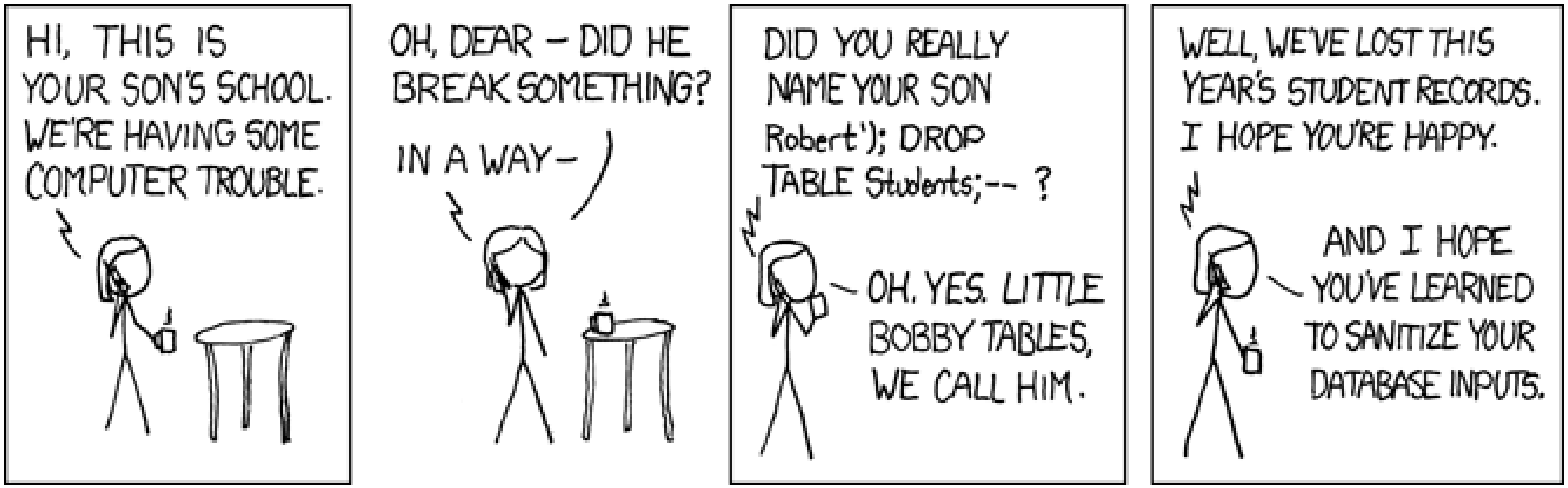
Incomplete mediation

- Incomplete mediation occurs when the application accepts incorrect data from the user.
- Sometimes this is hard to avoid:
  - Phone number: 604-882-2211
  - This is a reasonable entry, that happens to be wrong
- We focus on detecting inputs that are clearly wrong:
  - Not well formed:
    - DOB: 1980-04-31
  - Unreasonable values:
    - DOB: 1876-10-12
  - Inconsistent with other entries

Incomplete mediation

- Why is this an issue?
- Example, what happens if someone fills in:
  - DOB: 95834823765903823789593483267593 42
    - Buffer overflow?
  - DOB: `‘: DROP TABLE users; --`
    - SQL injection?
- We need to make sure that any user-supplied input falls within well-specified values, known to be safe

SQL Injection (xkcd)




```
SELECT grades FROM Students WHERE (NAME='%s')
```

https://xkcd.com/327/

Cross-Site Scripting (XSS) Attacks

- Data enters a web application through an untrusted source, most frequently a web request
- The data is included in dynamic content that is sent to a user
- The user's browser interprets the data as code
- Two common types:

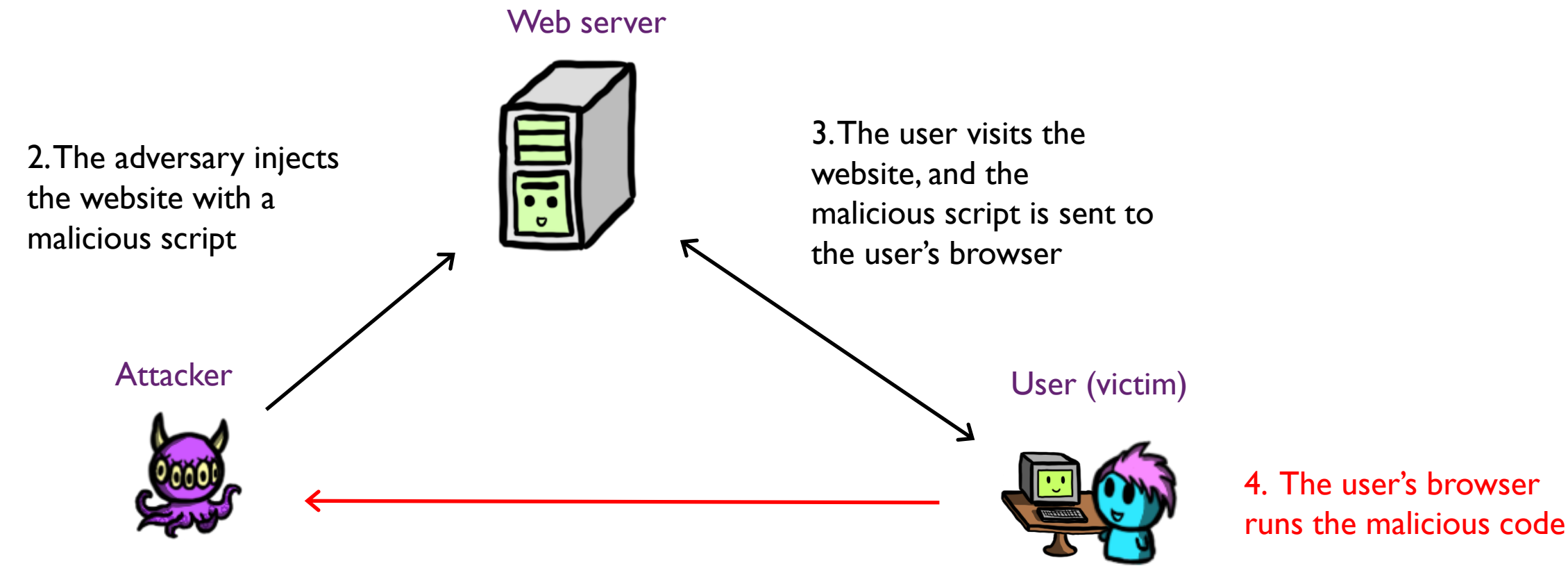
Stored XSS Attacks

- Stored attacks are those where the injected script is permanently stored on the target servers
- Database, log files, blog posts, etc.
- Data is retrieved and passed to the user upon query

Reflected XSS Attacks

- Happens when the web includes data received from an HTTP request in the response
- The adversary tricks the user into clicking on a carefully crafted link that points to a (benign) website
- The user clicks on that link, the website "reflects" the link's content, and the user's browser runs the malicious code

Stored XSS


Web server
2. The adversary injects the website with a malicious script
3. The user visits the website, and the malicious script is sent to the user's browser
Attacker
User (victim)
4. The user's browser runs the malicious code
1. The adversary discovers a website having a vulnerability that enables script injection

Reflected XSS

Web server

1. The adversary discovers a website having a vulnerability that enables script injection

3. The user clicks on the link and the injected script is reflected off the server

Attacker

User (victim)

2. The adversary crafts a malicious link with injected code and sends it to the user

4. The user's browser runs the malicious code

Client-side mediation

- Client-side mediation:
  - Some websites, when you click "submit", run a Javascript code that checks on the data you've entered.
  - If the data is invalid, you'll get a pop-up preventing you from submitting it
- Does this seem like a good solution?
  - What happens if the user turns off Javascript?
  - What if the user edits the form before submitting it? (Tampermonkey)
  - What if the user writes a script that interacts with the web server instead of using a web browser at all?
  - …
- This does not work for security purposes
- We need server-side mediation!! (of values entered by the users, and values stored by the server)

Real-life example of client-side mediation

- In a bookstore website, the user orders a copy of the course text.
- The server replies with a form asking the address to ship to. This form has hidden fields storing the user's order:

```
<input type="hidden" name="isbn" value="0-13-239077-9">
<input type="hidden" name="quantity" value="1">
<input type="hidden" name="unitprice" value="111.00">
```

- What happens if the user changes the "unitprice" value to "50.00" before submitting the form?

TOCTTOU errors

- TOCTTOU ("TOCK-too") errors:
  - Time-Of-Check To Time-Of-Use
  - Also known as "race condition" errors

- They occur when the following happens:
  1. The user requests the system to perform an action
  2. The system verifies the user is allowed to perform the action
  3. The system performs the action

- What happens if the state of the system changes between steps 2 and 3?

TOCTTOU Example

- Consider a Unix terminal program that has superuser privileges (setuid), so that it can allocate terminals to users (a privileged operation).
- This terminal supports a command to write the contents of the terminal to a log file.
  1. The user asks to write the contents of the terminal to a file
  2. The terminal program checks if the user has permissions to write to the requested file
  3. If so, it opens the file for writing and writes.
- The attacker makes a symbolic link:

  logfile -> file_she_owns
- Between the "check" and the "open", she changes it:

  logfile -> /etc/passwd

TOCTTOU Example

- We saw many program vulnerabilities in C, but TOCTTOU can happen in any language.
- You have to think about the atomicity of the check and the operation
- Toy example in Python:

```python
import os

def check_file(filename):
  if os.path.exists(filename):
    print("File exists, opening...")
    with open(filename, "r") as f:
      contents = f.read()
      # Do something
  else:
    print("File does not exist")

# Main code
filename = "example.txt"
check_file(filename)
```

example source

What went wrong?

- The state of the system has changed between the check for permission and the execution of the operation.
- The file whose permissions were checked for writeability by the user (file_she_owns) wasn't the same file that was later written to (/etc/passwd), even though they had the same name (logfile).

- Can the attacker really "win this race"?
  - YES

Defenses against TOCTTOU errors

- When performing a privileged action on behalf of another party, make sure all information relevant to the access control decision is constant between the *time of the check* and the *time of the action*.
- Keep a private copy of the request itself so that the request can't be altered during the race
- Where possible, act on the object itself, and not on some level of indirection
  - e.g., make access control decision based on filehandles, not filenames
- Where that's not possible, use locks to ensure the object is not changed during the race.