

# Building Security In

CMSC 330 Spring 2020

# Security breaches

Just a few:

- **TJX** (2007) - 94 million records\*
- **Adobe** (2013) - 150 million records, 38 million users
- **eBay** (2014) - 145 million records
- **Anthem** (2014) - Records of 80 million customers
- **Target** (2013) - 110 million records
- **Heartland** (2008) - 160 million records

*\*containing SSNs, credit card nums, other private info*

<https://www.oneid.com/7-biggest-security-breaches-of-the-past-decade-2/>



# The 2017 Equifax Data Breach

- **148** million consumers' personal information stolen
- They collect every details of your personal life
  - Your SSN, Credit Card Numbers, Late Payments...
- You did not sign up for it
- You cannot ask them to stop collecting your data
- You have to pay to credit freeze/unfreeze



# Defects and Vulnerabilities

- Many (if not all of) these breaches begin by exploiting a **vulnerability**
- This is a *security-relevant* **software defect** (**bug**) or **design flaw** that can be **exploited** to effect an undesired behavior

- The **use of software is growing**

Google

2B LOC

Windows

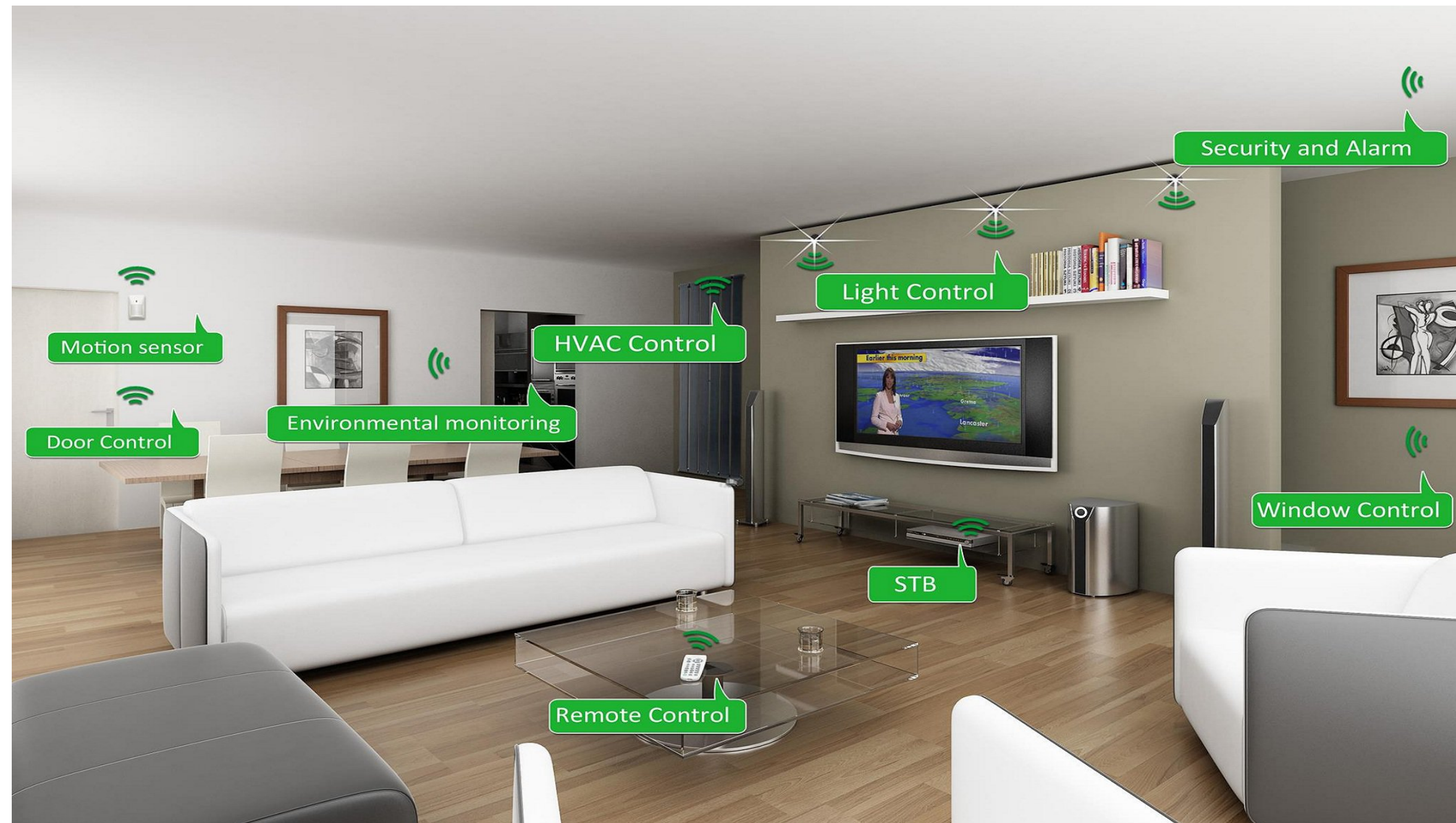
50M LOC

- So: **more bugs and flaws**
- Especially in places that are new to using software



# “Internet of Things” (IOT)

Amazon Alexa

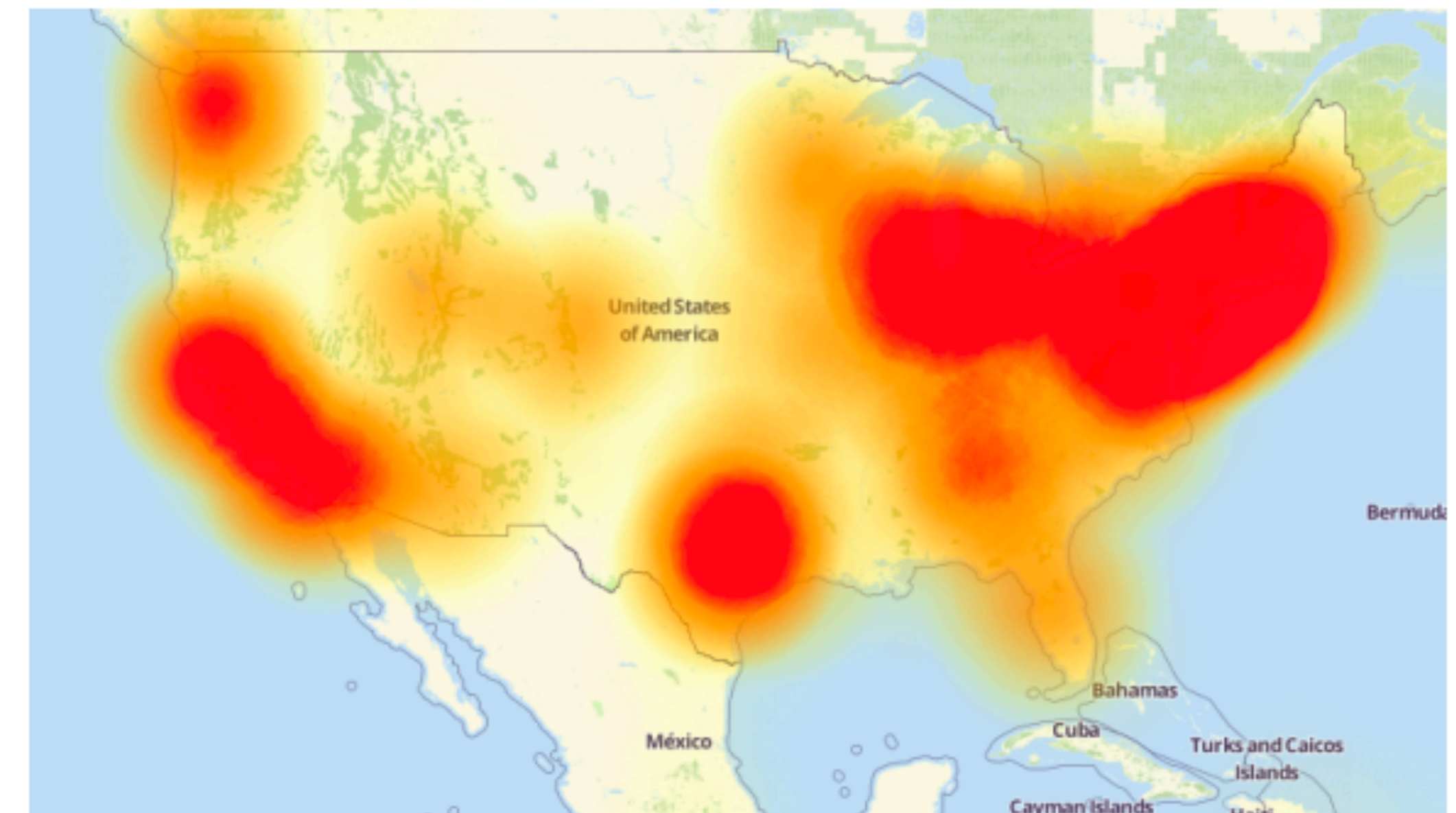


Google Home

## 21 Hacked Cameras, DVRs Powered Today's Massive Internet Outage

OCT 16

A massive and sustained Internet attack that has caused outages and network congestion today for a large number of Web sites was launched with the help of hacked “Internet of Things” (IoT) devices, such as CCTV video cameras and digital video recorders, new data suggests.



A depiction of the outages caused by today's attacks on Dyn, an Internet infrastructure company. Source: Downtetector.com.

<https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/>

## MIDDLE EAST

# Iran Fights Malware Attacking Computers

By DAVID E. SANGER SEPT. 25, 2010

Email

Share

Tweet

Save

More

WASHINGTON — The Iranian government agency that runs the country's nuclear facilities, including those the West suspects are part of a weapons program, has reported that its engineers are trying to protect their facilities from a sophisticated computer worm that has infected industrial plants across [Iran](#).

The agency, the Atomic Energy Organization, did not specify whether the worm had already infected any of its nuclear facilities, including Natanz, the underground enrichment site that for several years has been a main target of American and Israeli covert programs.

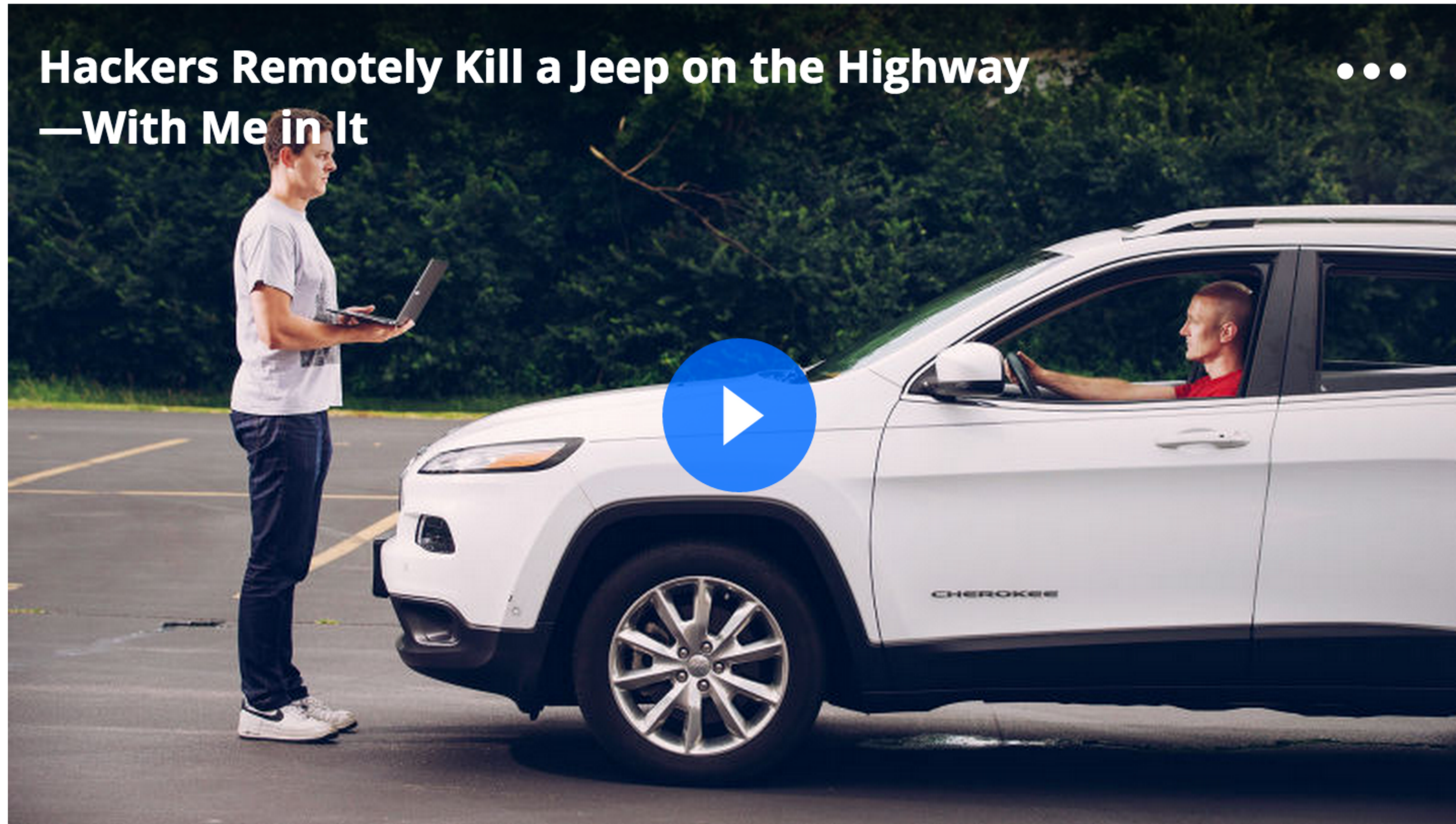
But the announcement raised suspicions, and new questions, about the origins and target of the worm, Stuxnet, which computer experts say is a far cry from common computer malware that has affected the Internet for years. A worm is a self-replicating malware computer program. A virus is malware that infects its target by attaching itself to programs or documents.

Stuxnet specifically targets ... processes such as those used to control ... **centrifuges for separating nuclear material**. Exploiting four zero-day flaws, **Stuxnet** functions by targeting machines using the Microsoft Windows operating system ..., then seeking out Siemens Step7 software.

<http://www.nytimes.com/2010/09/26/world/middleeast/26iran.html>



# HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

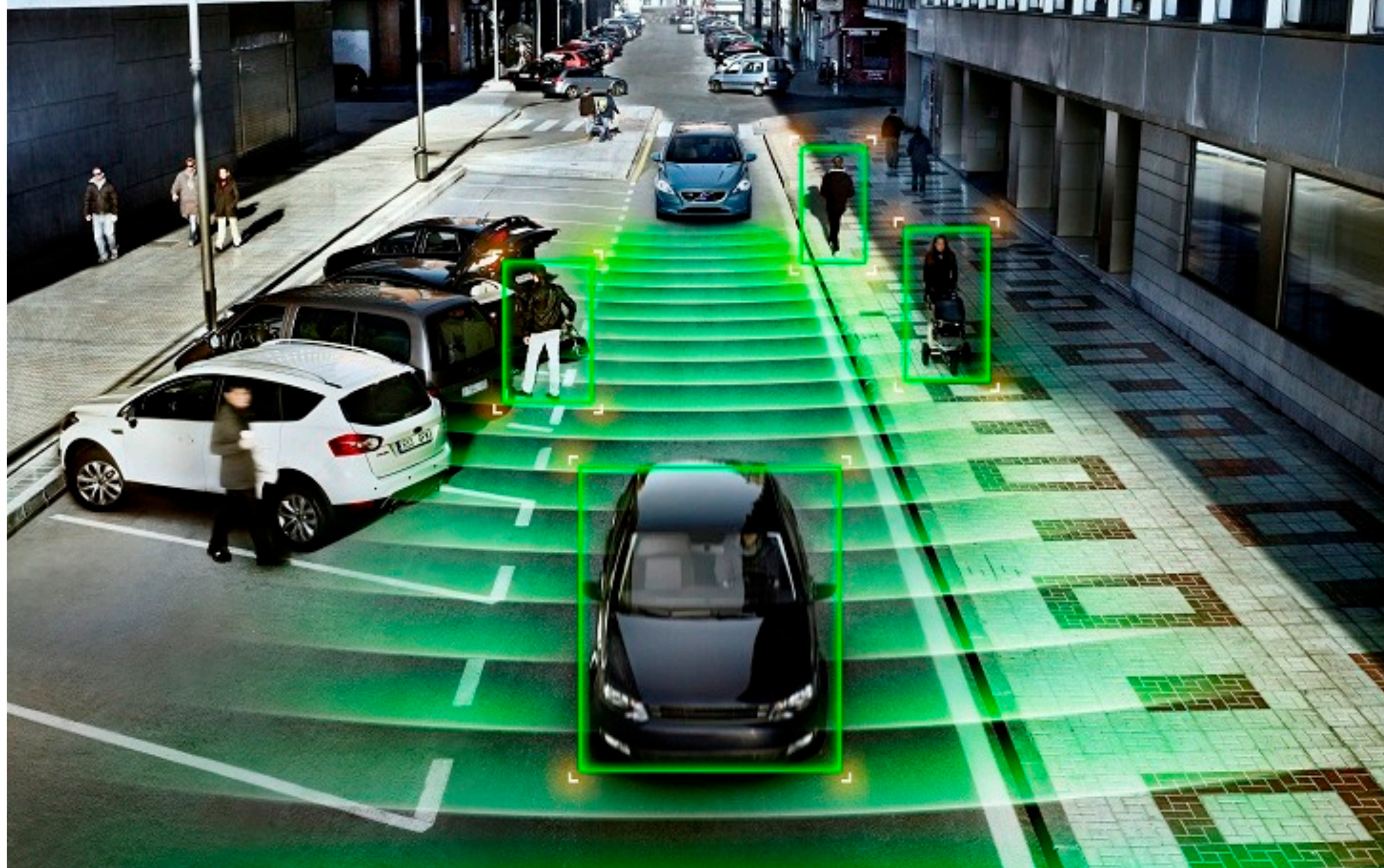


The result of their work was a hacking technique—what the security industry calls a zero-day exploit—that can **target Jeep Cherokees and give the attacker wireless control**, via the Internet, to any of thousands of vehicles.

<http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>

I WAS DRIVING 70 mph on the edge of downtown St. Louis when the exploit began

# Driverless Cars





# Considering **Correctness**

- **All software is buggy**, isn't it? Haven't we been dealing with this for a long time?
- A **normal user never sees most bugs**, or figures out how to **work around** them
- Therefore, **companies fix the most likely bugs**, to save money

# Considering **Security**

Key difference:

***An attacker is not a normal user!***

- The attacker **will actively attempt to find defects**, using unusual interactions and features
- A **typical interaction** with a bug results in a **crash**
- An attacker will work to **exploit** the bug to do **much worse**, to achieve his goals

# Cyber-defense?



MUST READ **THE NIGHT ALEXA LOST HER MIND: HOW AWS OUTAGE CAUSED ECHO MAYHEM**

## FireEye, Kaspersky hit with zero-day flaw claims

Researchers have disclosed severe security flaws within the firm's products over the holiday weekend.



By [Charlie Osborne](#) for [Zero Day](#) | September 8, 2015 -- 09:45 GMT (02:45 PDT) | Topic: [Security](#)



Researchers have revealed the existence of zero-day vulnerabilities within Kaspersky and FireEye's systems which could compromise customer safety.

Over the holiday weekend, security researcher Tavis Ormandy disclosed the existence of a vulnerability which impacts on Kaspersky products. Ormandy, known in the past for publicly revealing security flaws in Sophos and ESET antivirus products, said the vulnerability is "about as bad as it gets." [In a tweet](#), the researcher said:

and **bugs in security products** themselves!

Security researcher Tavis Ormandy disclosed the existence of **a vulnerability which impacts on Kaspersky [security] products.**

Hermansen, [another researcher,] publicly disclosed a zero-day **vulnerability within cyberforensics firm FireEye's security product**, complete with proof-of-concept code.

<http://www.zdnet.com/article/fireeye-kaspersky-hit-with-zero-day-flaw-claims/>

# Exploitable bugs

- Some **bugs** can be **exploited**
  - An attacker can control how the program runs so that any incorrect behavior serves the attacker
- **Many kinds of exploits** have been developed over time, with technical names like
  - Buffer overflow
  - Use after free
  - SQL injection
  - Command injection
  - Privilege escalation
  - Cross-site scripting
  - Path traversal
  - ...

# What is a buffer overflow?

- A buffer overflow is a dangerous bug that affects programs written in **C** and **C++**
- **Normally**, a program with this bug will simply **crash**
- But an **attacker** can alter the situations that cause the program to **do much worse**
  - **Steal** private information
  - **Corrupt** valuable information
  - **Run code** of the attacker's choice

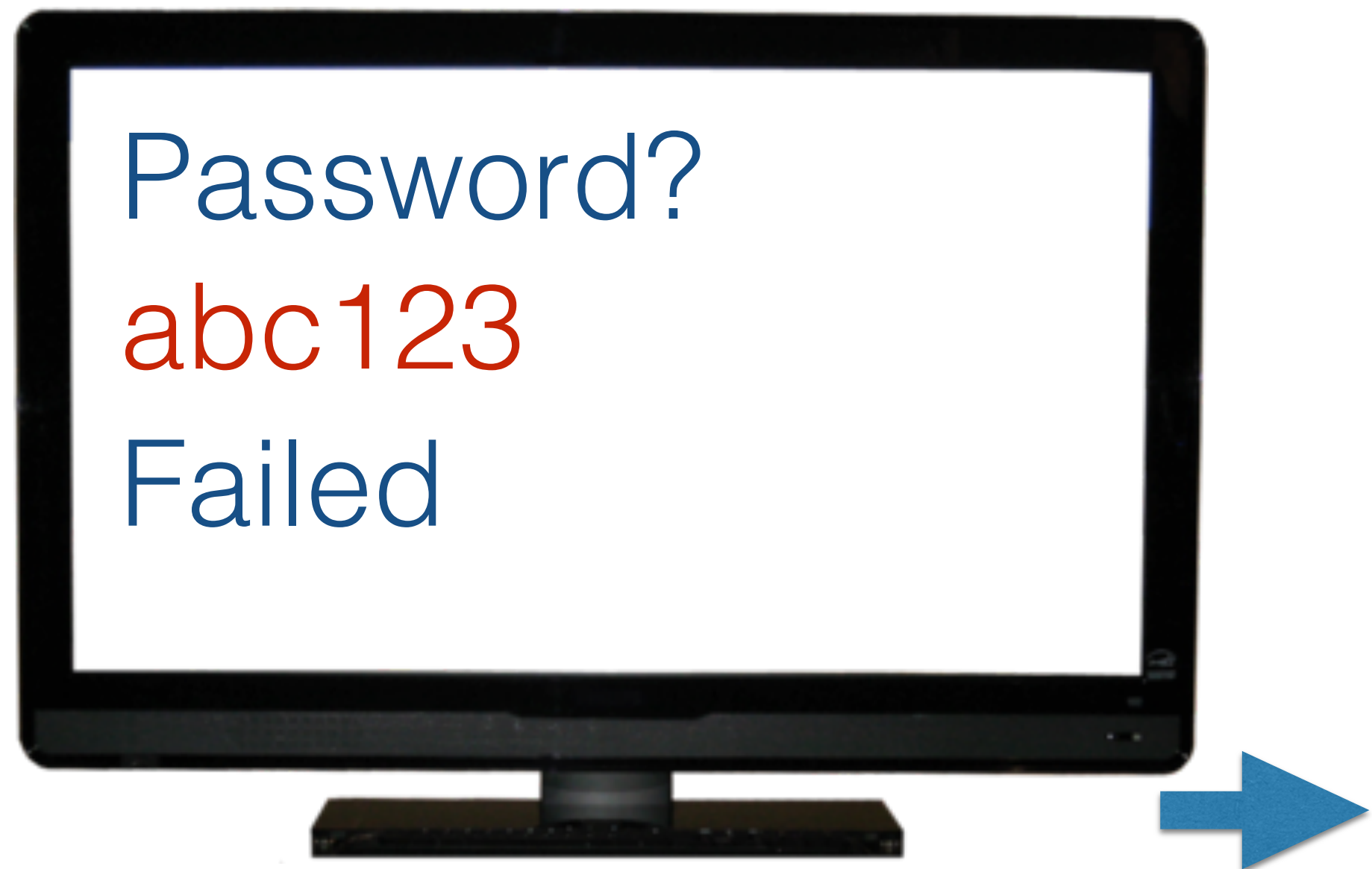


# Buffer overflows from 10,000 ft

- **Buffer =**
  - Block of memory associated with a variable
- **Overflow =**
  - Put more into the buffer than it can hold
- **Where does the overflowing data go?**

*Learn more in CMSC 414!*

# Normal interaction



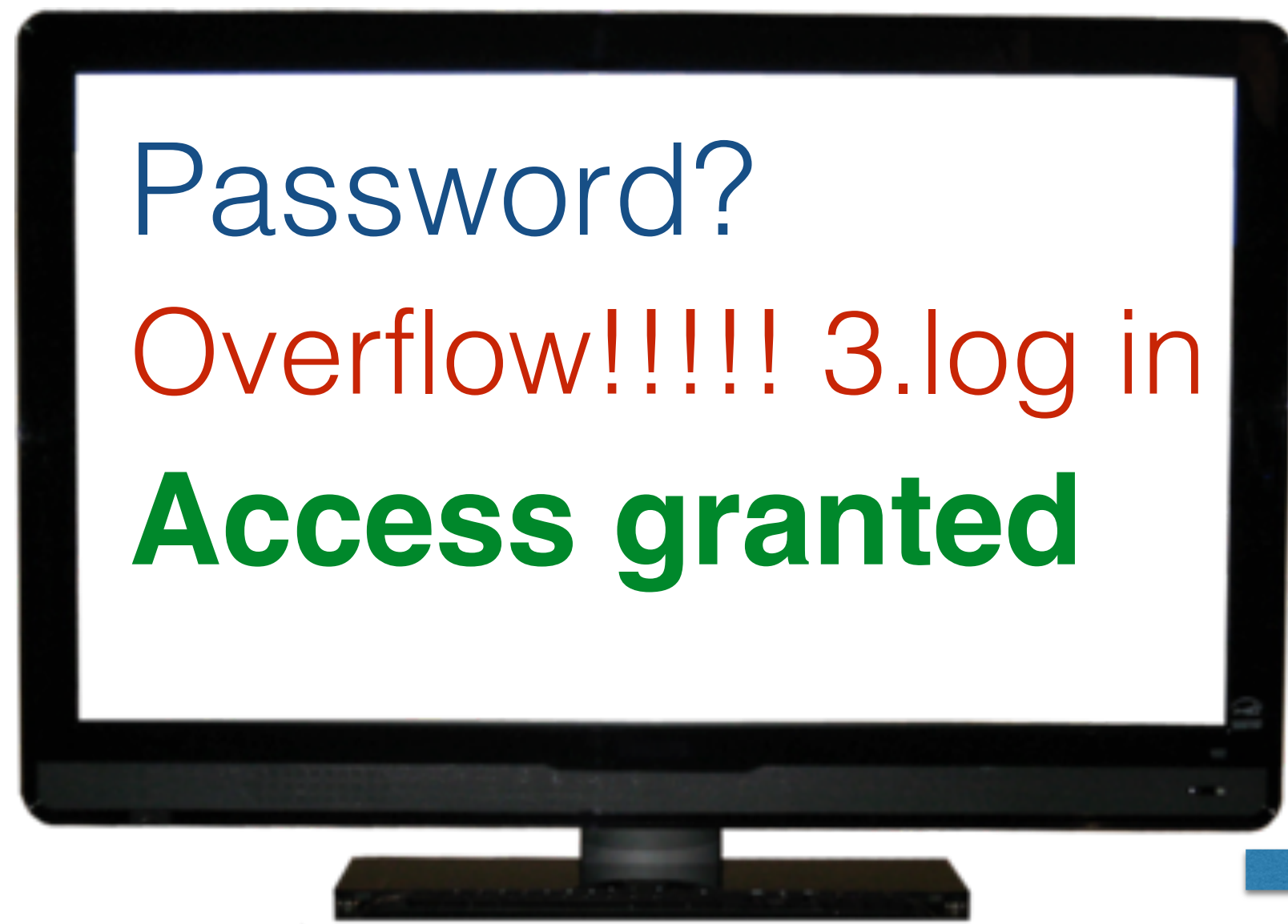
## Data

X = abc123

## Instructions

1. print "Password?" to the screen
2. read input into variable X
3. if X ~~matches~~ the password then log in
4. else print "Failed" to the screen





Password?

Overflow!!!! 3.log in

**Access granted**

Data

X = Overflow!!!! 3.log in

# Exploitation

Instructions

1. print "Password?" to the screen

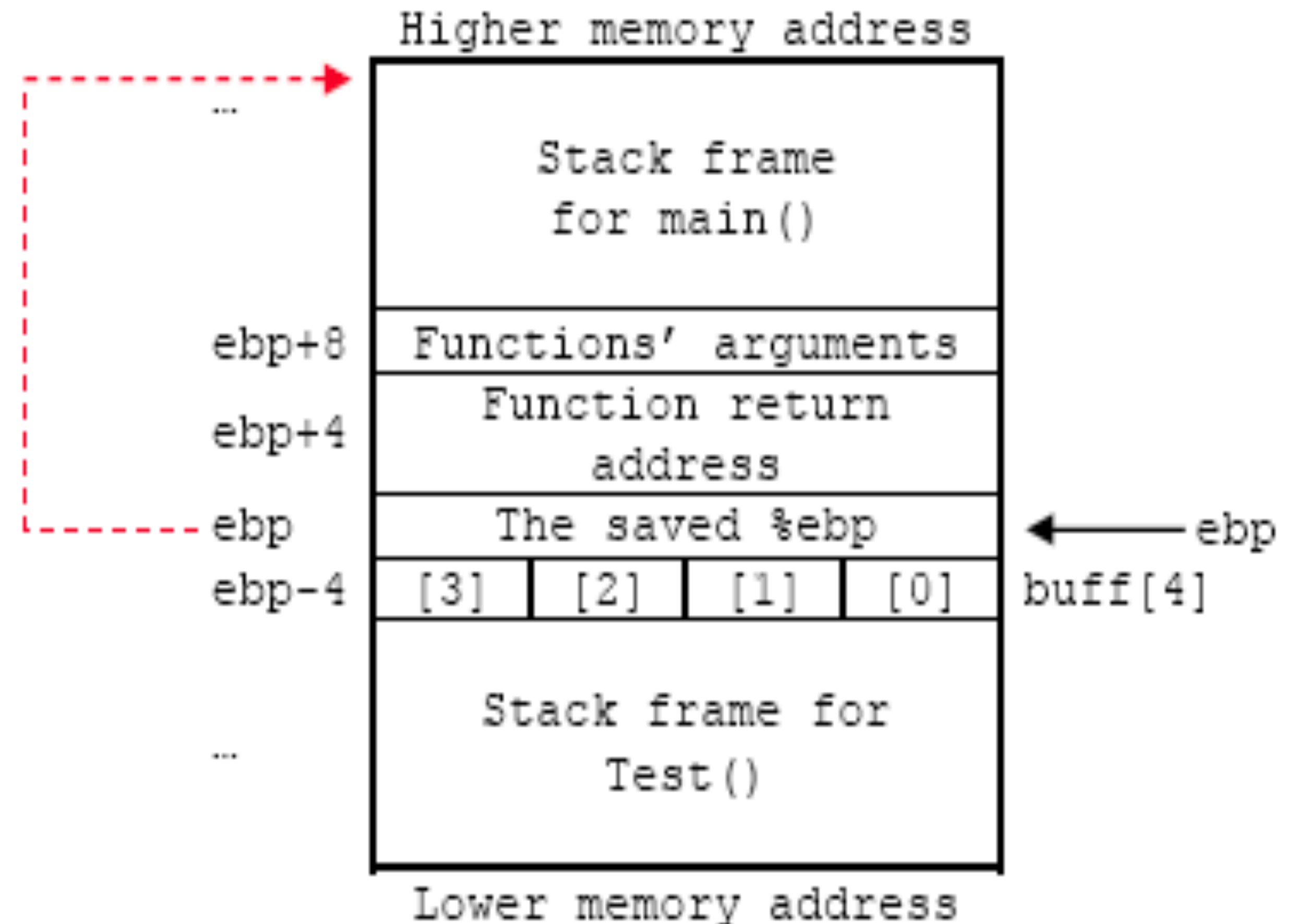
2. read input into variable X

4. else print "Failed" to the screen

# What happened?

- For C/C++ programs
  - A buffer with the password could be a local variable
- Therefore
  - The input is too long, and overruns the buffer
  - The attacker's input includes machine instructions
  - The overrun rewrites the return address to point into the buffer, at the machine instructions
  - When the call "returns" it executes the attacker's code

```
strcpy(buff, "abc");
```



# Stopping the attack

- **Buffer overflows** rely on the ability to **read or write outside the bounds of a buffer**
- **C and C++** programs expect the **programmer** to ensure this never happens
  - But humans (regularly) make mistakes!
- Other languages (like **Python, OCaml, Java**, etc.) ensure buffer sizes are respected
  - The **compiler** inserts checks at reads/writes
  - Such checks can halt the program
  - But will **prevent a bug from being exploited**

# Preventing Exploitation

## Instructions

1. print "Password?" to the screen
2. read input into variable X
3. if X matches the password then log in
4. else print "Failed" to the screen

## Data

X = Overflow!



*Program halted*

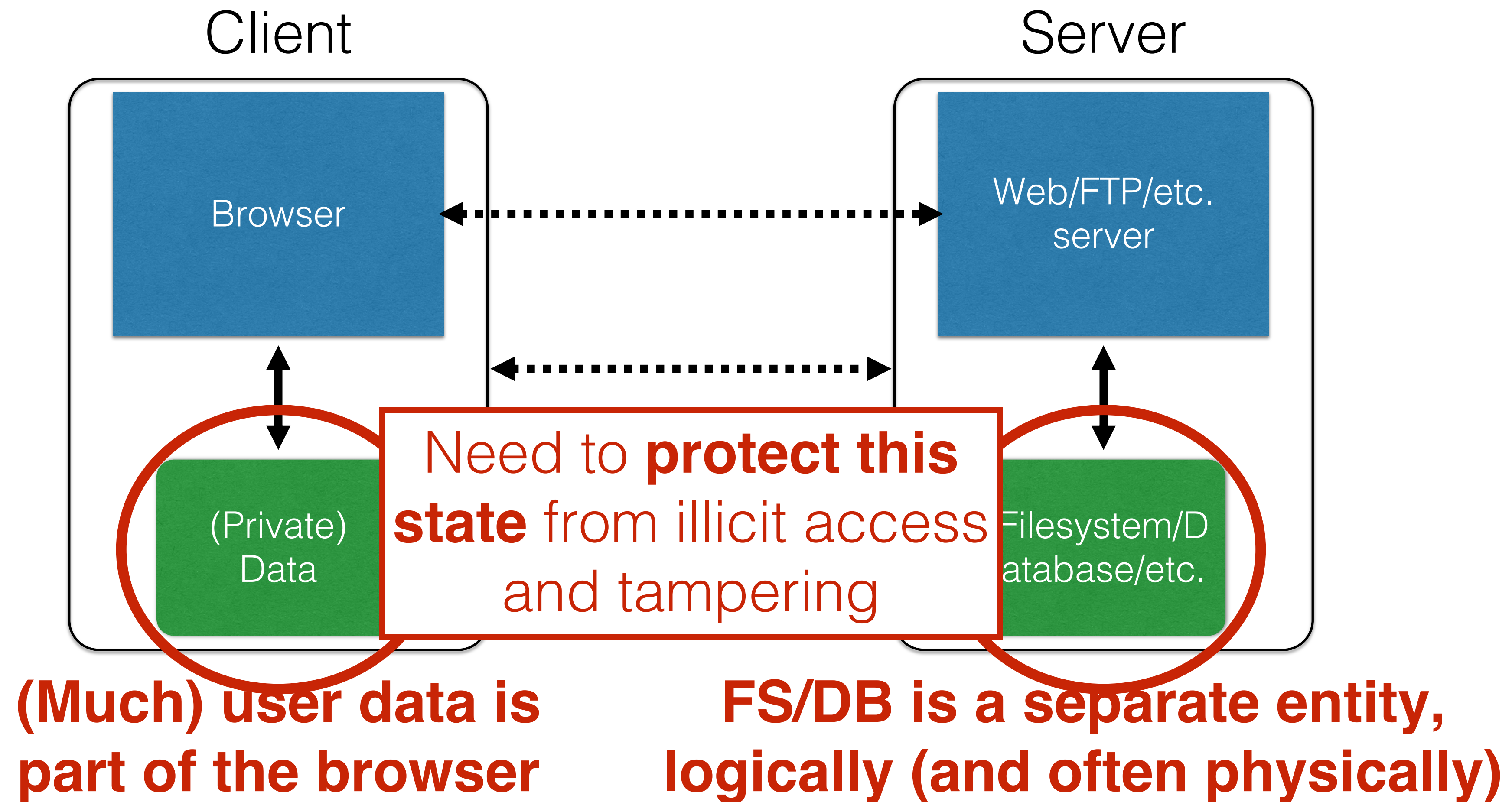


# Key idea

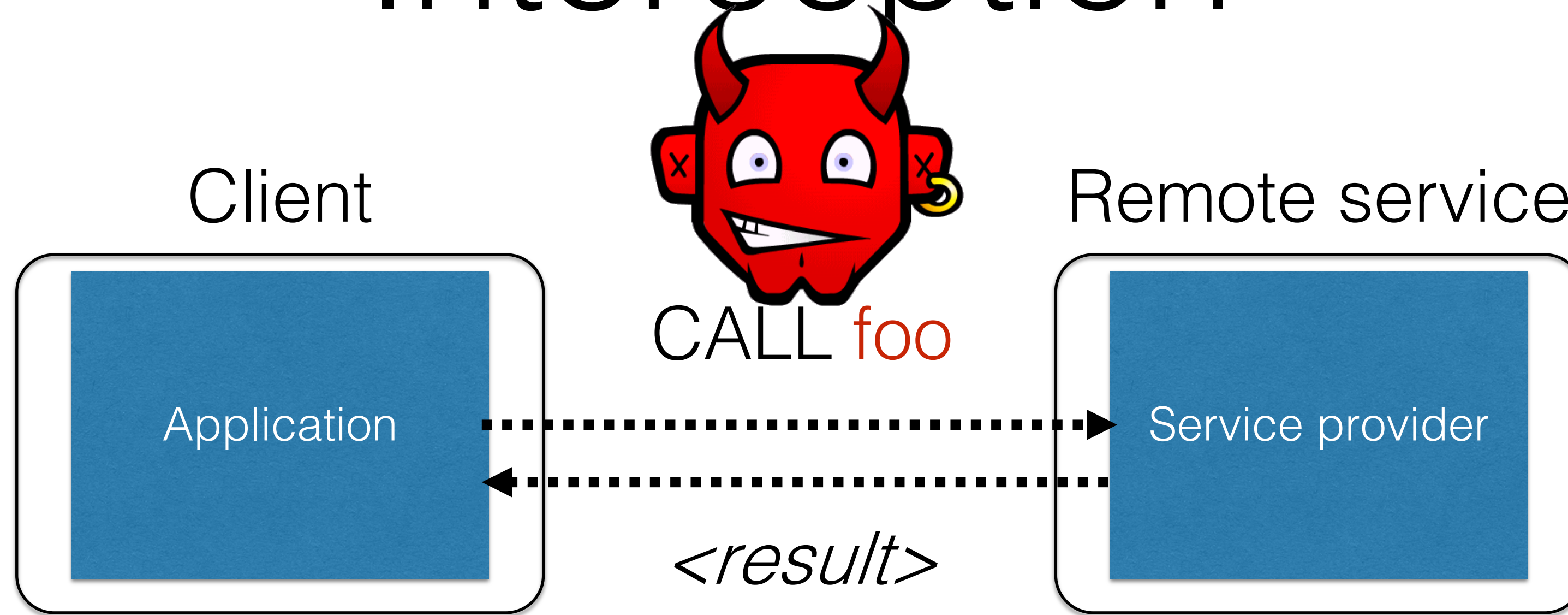
- The key feature of the buffer overflow attack is the attacker getting the application to **treat attacker-provided data as instructions (code) or code parameters**
- This feature appears in many **other exploits** too
  - SQL injection treats data as **database queries**
  - Cross-site scripting treats data as **browser commands**
  - Command injection treats data as **operating system commands**
  - Etc.
- Sometimes the language helps (e.g., type safety)
  - Sometimes the programmer needs to do more work

# Attack Scenarios

# The Internet, in one slide



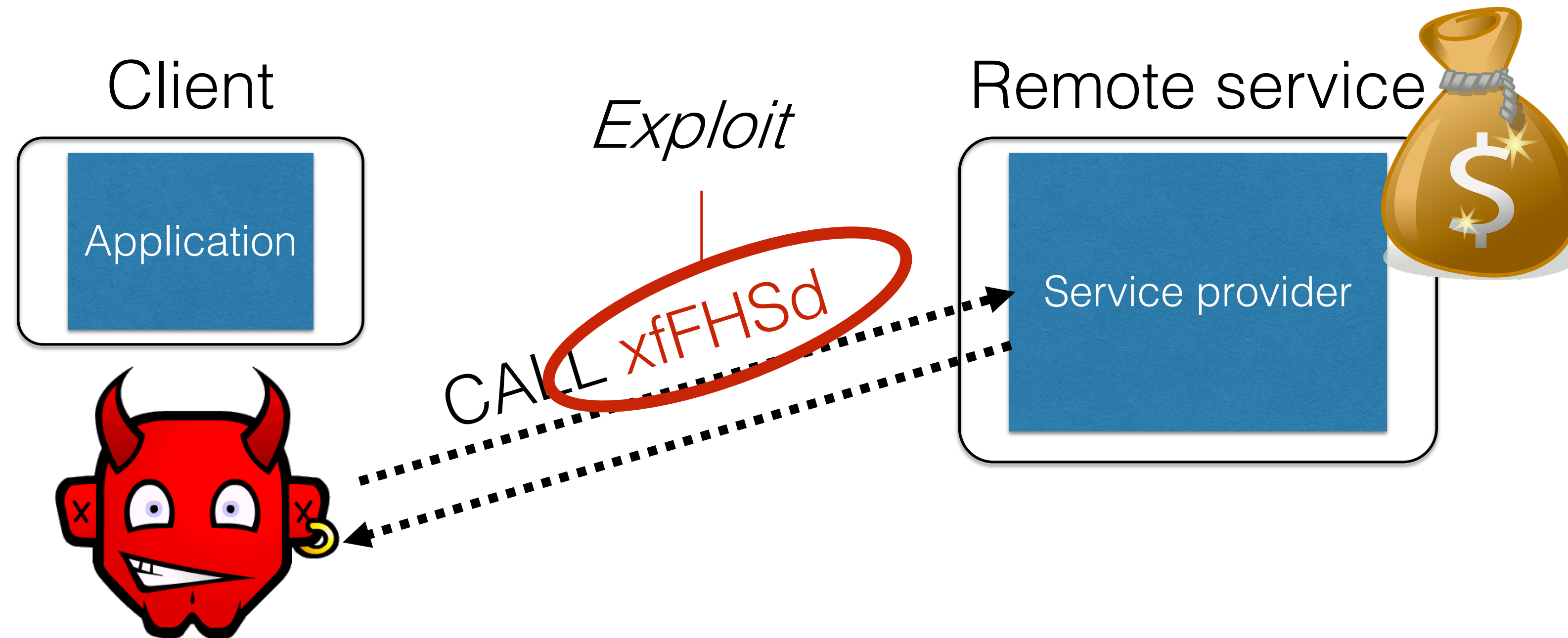
# Interception



- **Calls** to remote services could be **intercepted** by an adversary
  - **Snoop** on inputs/outputs
  - **Corrupt** inputs/outputs
- Avoid this possibility using **cryptology** (CMSC 414, CMSC 456)

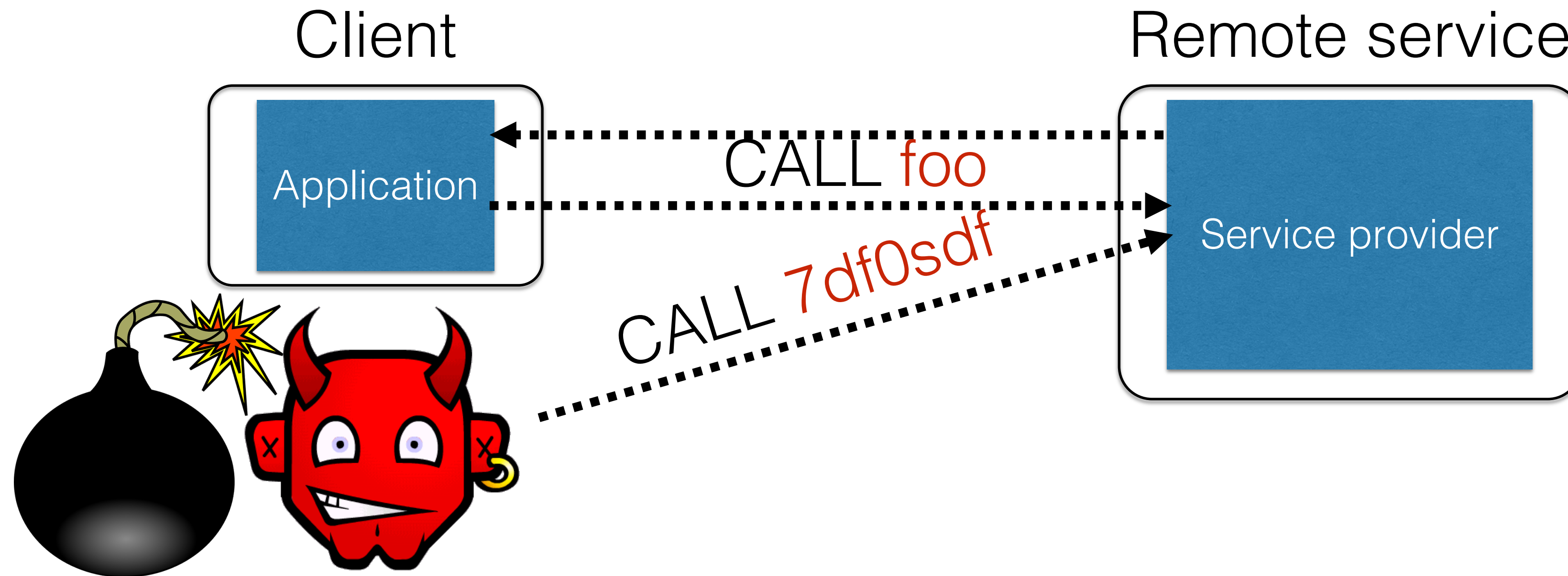


# Malicious clients



- Server needs to **protect itself against malicious clients**
  - Won't run the software the server expects
  - Will probe the limits of the interface

# Passing the buck



- **Server needs to protect good clients** from malicious clients that will try to launch attacks via the server
  - Corrupt the server state (e.g., uploading malicious files or code)
  - Good client interaction affected as a result (e.g., getting the malware)

# Defensive measures

- Two key actions the server can take:
- **Validate that client inputs are well formed**
  - Fallacy: Focus on testing that good inputs produce good behavior
  - Must (also) ensure that malformed inputs result in benign behavior
- Mitigate harm that might result by **minimizing the trusted computing base**
  - Isolate trusted components, or minimize privilege to precisely what is needed, in case something goes wrong

# Quiz 1: What are reasonable assumptions?

Suppose you are writing a PDF viewer that is leaner and better than Acrobat Reader. Which can you assume?

- A. PDF files given to your reader will always be well-formed
- B. PDF files will never exceed a particular size
- C. Your viewer will never be used as part of an Internet-hosted service
- D. None of the above

# Quiz 1: What are reasonable assumptions?

Suppose you are writing a PDF viewer that is leaner and better than Acrobat Reader. Which can you assume?

- A. PDF files given to your reader will always be well-formed
- B. PDF files will never exceed a particular size
- C. Your viewer will never be used as part of an Internet-hosted service
- D. None of the above**

# Validating inputs

# What's wrong with this Ruby code?

*catwrapper.rb:*

```
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

# Possible Interaction

```
> ls
```

```
catwrapper.rb
```

```
hello.txt
```

```
> ruby catwrapper.rb hello.txt
```

```
Hello world!
```

```
> ruby catwrapper.rb catwrapper.rb
```

```
if ARGV.Length < 1 then
```

```
  puts "required argument: textfile path"
```

```
...
```

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
```

```
Hello world!
```

```
> ls
```

```
catwrapper.rb
```



# Quiz 2: What happened?

- A. `cat` was given a file named `hello.txt`; `rm hello.txt` which doesn't exist
- B. `system()` interpreted the string as having two commands, and executed them both
- C. `cat` was given three files – `hello.txt`; and `rm` and `hello.txt` – but halted when it couldn't find the second of these
- D. `ARGV[0]` contains `hello.txt` (only), which was then catted

## *catwrapper.rb:*

```
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
```

```
Hello world!
```

```
> ls
```

```
catwrapper.rb
```

# Quiz 2: What happened?

- A. `cat` was given a file named `hello.txt`; `rm hello.txt` which doesn't exist
- B. `system()` interpreted the string as having two commands, and executed them both
- C. `cat` was given three files – `hello.txt`; and `rm` and `hello.txt` – but halted when it couldn't find the second of these
- D. `ARGV[0]` contains `hello.txt` (only), which was then catted

## *catwrapper.rb:*

```
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

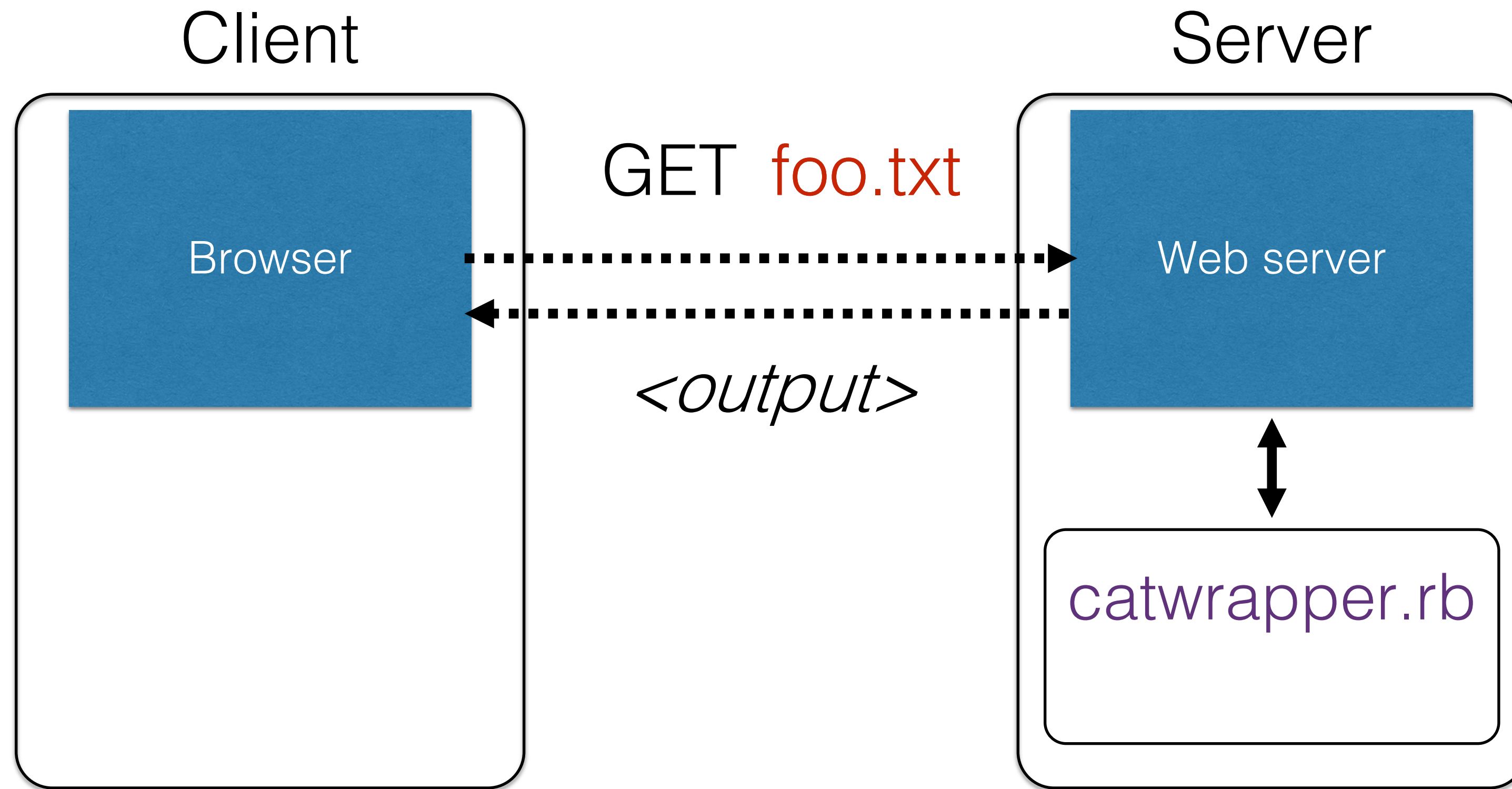
```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
```

```
Hello world!
```

```
> ls
```

```
catwrapper.rb
```

# Possible deployment



# Consequences?

- If `catwrapper.rb` is part of a web service
  - **Input is untrusted** — could be anything
  - But we only want requestors to read (see) the contents of the files, not to do anything else
  - Current code is too powerful: vulnerable to

## *command injection*

- How to fix it?

## **Need to validate inputs**

[https://www.owasp.org/index.php/Command\\_Injection](https://www.owasp.org/index.php/Command_Injection)

# Equifax: What happened

- Equifax used Struts which failed to properly vet input prior to using deserialization. Ruby had a similar bug sometime back.
- Vulnerability was discovered in a popular open-source software package Apache Struts, a programming framework for building web applications in Java
- The framework's popular REST plugin is vulnerable. The REST plugin is used to handle web requests, like data sent to a server from a form a user has filled out.
- The vulnerability relates to how Struts parses that kind of data and converts it into information that can be interpreted by the Java programming language.
- When the vulnerability is successfully exploited, malicious code can be hidden inside of such data, and executed when Struts attempts to convert it.
- Intruders can inject malware into web servers, without being detected, and use it to steal or delete sensitive data, or infect computers with ransomware, among other things.

# Input Validation

- We expect input of a certain form
  - But we cannot guarantee it always has it
    - it's under the attacker's control
  - So we must **validate it before we trust it**
- **Making input trustworthy**
  - **Sanitize it** by modifying it or using it in such a way that the result is correctly formed by construction
  - **Check it** has the expected form, and reject it if not

# Checking: Blacklisting

- **Reject** strings with possibly bad chars: ' ; --

```
if ARGV[0] =~ /;/ then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end
```

*reject inputs  
that have ; in  
them*

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
illegal argument
```

# Sanitization: Blacklisting

- Delete the characters you don't want: ' ; --

```
system("cat "+ARGV[0].tr(";",""))
```

*delete  
occurrences  
of ; from input  
string*

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"  
Hello world!  
cat: rm: No such file or directory  
Hello world!  
> ls hello.txt  
hello.txt
```



# Sanitization: Escaping

- **Replace problematic characters with safe ones**
  - change ' to \'
  - change ; to \;
  - change - to \-
  - change \ to \\
- Which characters are problematic depends on the interpreter the string will be handed to
  - Web browser/server for URIs
    - `URI::escape(str, unsafe_chars)`
  - Program delegated to by web server
    - `CGI::escape(str)`

# Sanitization: Escaping

```
def escape_chars(string)
  pat = /(\'|\"|\.|*|\/|\-|\\|;|\||\s)/
  string.gsub(pat){|match| "\\\" + match}
end

system("cat "+escape_chars(ARGV[0]))
```

**escape**  
**occurrences**  
of `'`, `"`, `;` etc.  
in input string

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"
cat: hello.txt; rm hello.txt: No such file or directory
> ls hello.txt
hello.txt
```

# Quiz 3: Is this escaping sufficient?

- A. No, you should also escape character &
- B. No, some of the escaped characters are dangerous even when escaped
- C. Both of the above
- D. Yes, it's all good

*catwrapper.rb:*

```
def escape_chars(string)
  pat = /(\'|\"|\.|*|\/|\-|\\|;|\||\s)/
  string.gsub(pat){|match|"\\" + match}
end
system("cat "+escape_chars(ARGV[0]))
```

# Quiz 3: Is this escaping sufficient?

- A. No, you should also escape character &
- B. No, some of the escaped characters are dangerous even when escaped
- C. Both of the above**
- D. Yes, it's all good

*catwrapper.rb:*

```
def escape_chars(string)
  pat = /(\'|\"|\.|*|\/|\-|\\|;|\||\s)/
  string.gsub(pat){|match|"\\" + match}
end
system("cat "+escape_chars(ARGV[0]))
```

# Escaping not always enough

```
> ls ../passwd.txt  
passwd.txt  
> ruby catwrapper.rb “../passwd.txt”  
bob:apassword  
alice:anotherpassword
```

- A web service probably only wants to give access to the files in the current directory
  - the .. sequence should have been disallowed
- Previous escaping doesn't help because . is replaced with \. which the shell interprets as .

# Path traversal

This is called a **path traversal** vulnerability. Solutions:

- Delete all occurrences of the . character
  - Will disallow legitimate files with dots in them (`hello.txt`)
- Delete occurrences of .. sequences
  - Safe, but disallows `foo/./hello.txt` where `foo` is a subdirectory in the current working directory (CWD)
- Ideally: Allow any path that is within the CWD or one of its subdirectories

[https://www.owasp.org/index.php/Path\\_Traversal](https://www.owasp.org/index.php/Path_Traversal)

# Checking: Whitelisting

- **Check that the user input is known to be safe**
  - E.g., only those files that exactly match a filename in the current directory
- **Rationale:** Given an invalid input, **safer to reject than to fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - *Principle of fail-safe defaults*

# Checking: Whitelisting

```
files = Dir.entries(".").reject {|f| File.directory?(f) }
```

```
if not (files.member? ARGV[0]) then  
  puts "illegal argument"  
  exit 1  
else  
  system("cat "+ARGV[0])  
end
```

*reject inputs that  
do not mention a  
legal file name*

```
> ruby catwrapper.rb "hello.txt; rm hello.txt"  
illegal argument
```



# Validation Challenges

- **Cannot always delete or sanitize problematic characters**
  - You may want dangerous chars, e.g., “Peter O’Connor”
  - How do you know if/when the characters are bad?
  - Hard to think of all of the possible characters to eliminate
- **Cannot always identify whitelist cheaply or completely**
  - May be expensive to compute at runtime
  - May be hard to describe (e.g., “all possible proper names”)

# Key Questions

- Which inputs in my program should not be trusted?
  - These start from input from untrusted sources
  - And these inputs influence (“taint”) other data that flows through my program
    - And could be stored in files, databases, etc.
- How to ensure that untrusted inputs, no matter what they are, will produce benign results?
  - Sanitization, checking, etc. as early as possible
    - How to do this depends on the program, and how the inputs are used

# Quiz 4: As a developer, security is

- A. Something I can help address by writing better code
- B. Something that writing better code can do little to address
- C. Something that is the purview of the government, e.g., DHS
- D. Something that will never be solved so long as market forces do not value security

(Pick an answer you think is best)

# Security for the **Web**

Thanks to Dave Levin for  
some slides



# The Web

- **Security for the World-Wide Web (WWW)** presents new vulnerabilities to consider:
  - **SQL injection**,
  - Cross-site Scripting (**XSS**),
- These share some common causes with memory safety vulnerabilities; like **confusion of code and data**
  - **Defense** also similar: **validate untrusted input**
- New wrinkle: **Web 2.0's use of mobile code**
  - How to protect your applications and other web resources?

# Interacting with web servers

## **Resources which are identified by a URL**

(Universal Resource Locator)

<http://www.cs.umd.edu/~mwh/index.html>

**Protocol      Hostname/server      Path to a resource**

ftp

https

tor

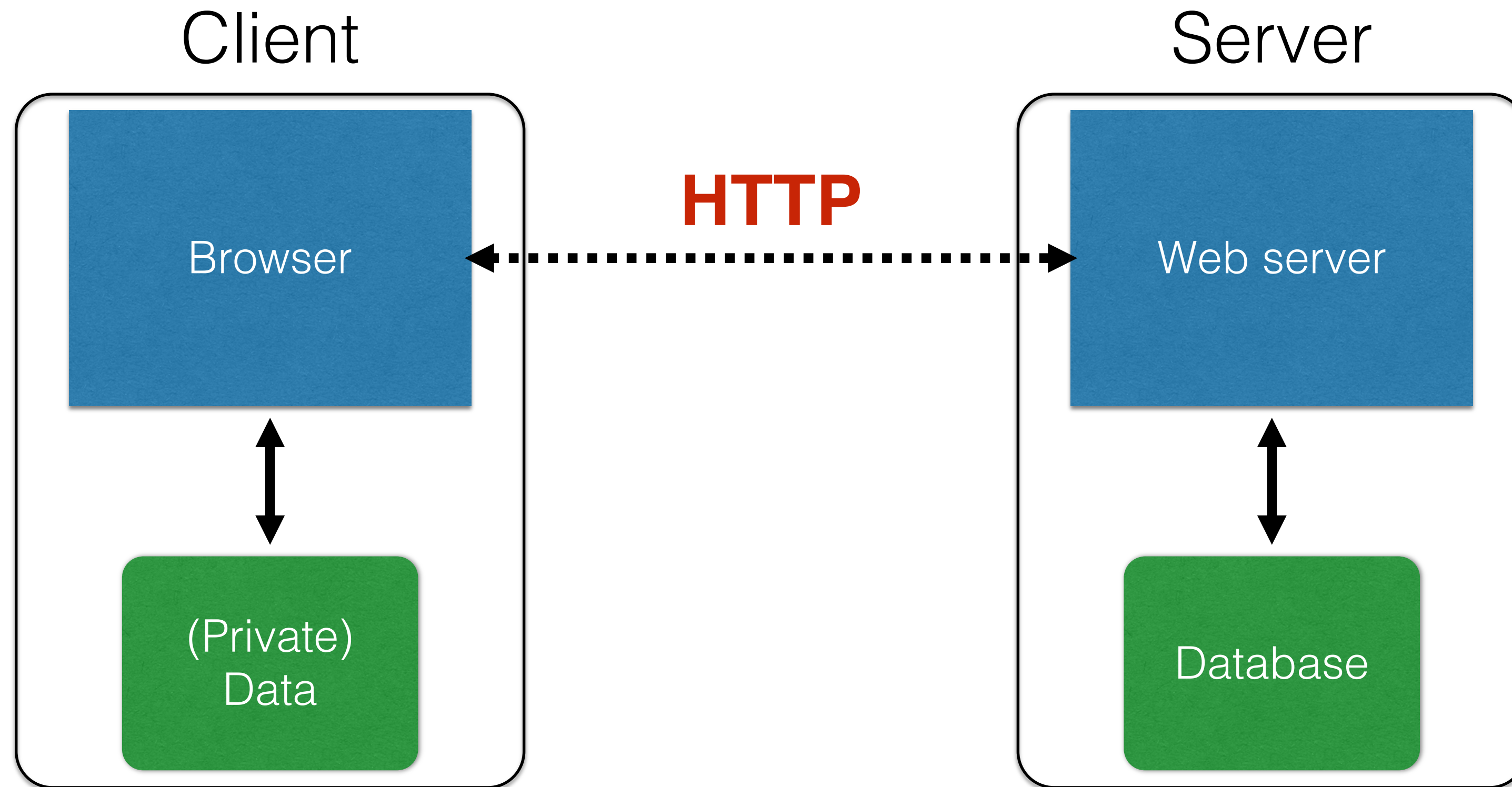
Here, the file `index.html` is **static content**  
Translated to an IP address by DNS  
i.e., (e.g. `128.8.127.3`) by the server

<http://facebook.com/delete.php?f=joe123&w=16>

**Path to a resource Arguments**

Here, the file `delete.php` is **dynamic content**  
i.e., the server generates the content on the fly

# *Basic* structure of web traffic



- HyperText Transfer Protocol (**HTTP**)
  - An “application-layer” protocol for exchanging collections of data

# *Basic* structure of web traffic



- **Requests contain:**
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do
- **Request types** can be **GET** or **POST**
  - **GET**: all data is in the URL itself (no server side effects)
  - **POST**: includes the data as separate fields (can have side effects)



# HTTP GET requests

<http://www.reddit.com/r/security>

## HTTP Headers

http://www.reddit.com/r/security

GET /r/security HTTP/1.1

Host: www.reddit.com

**User-Agent:** Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

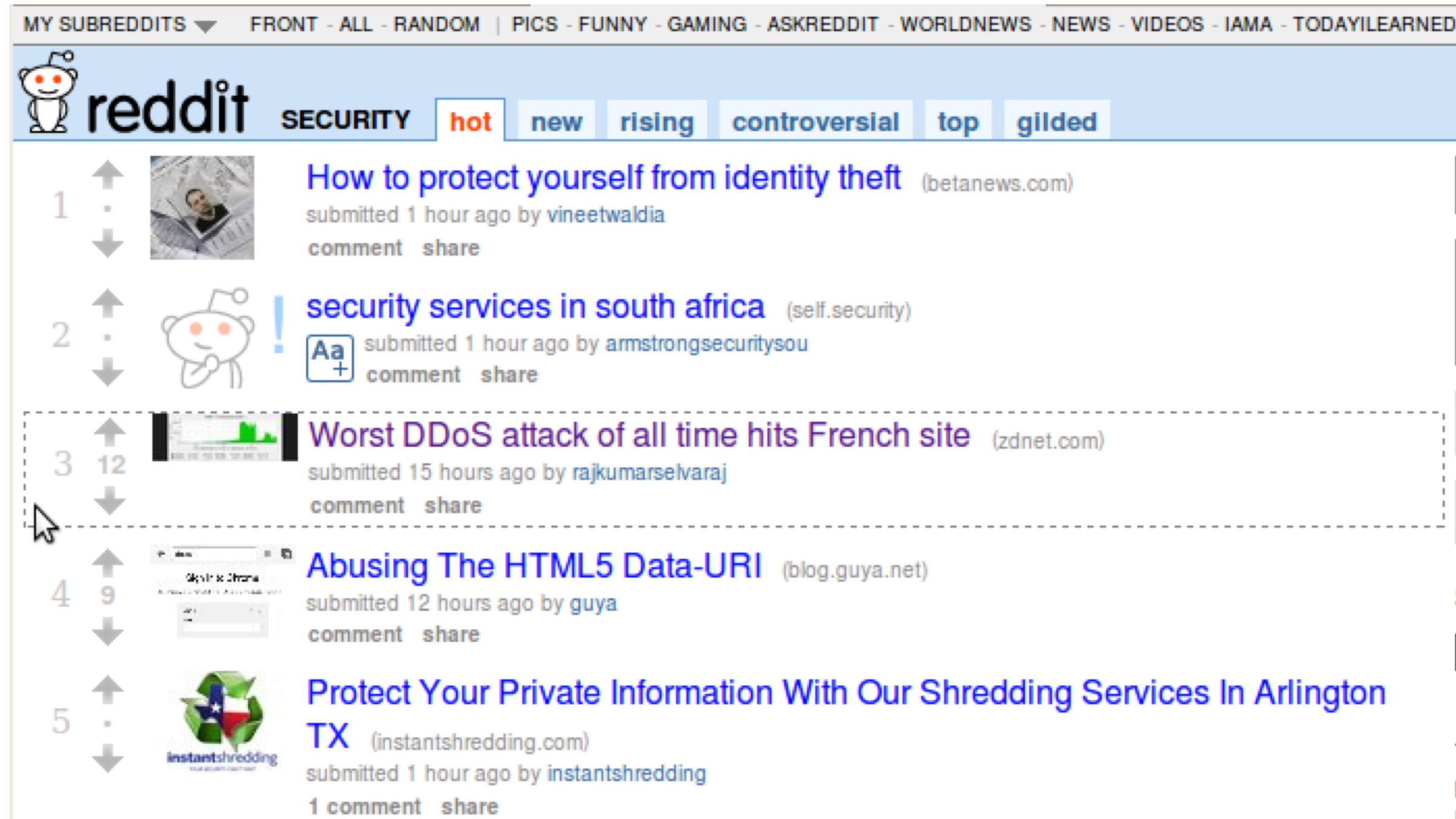
Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive: 115

Connection: keep-alive

\_\_utm=55650...

**User-Agent** is typically a **browser** but it can be `wget`, `JDK`, etc.



## HTTP Headers

<http://www.zdnet.com/worst-ddos-attack-of-all-time-hits-french-site-7000026330/>

GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1

Host: www.zdnet.com

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive: 115

Connection: keep-alive

Referer: <http://www.reddit.com/r/security>

**Referrer URL: the site from which this request was issued.**

# HTTP POST requests

## Posting on Piazza

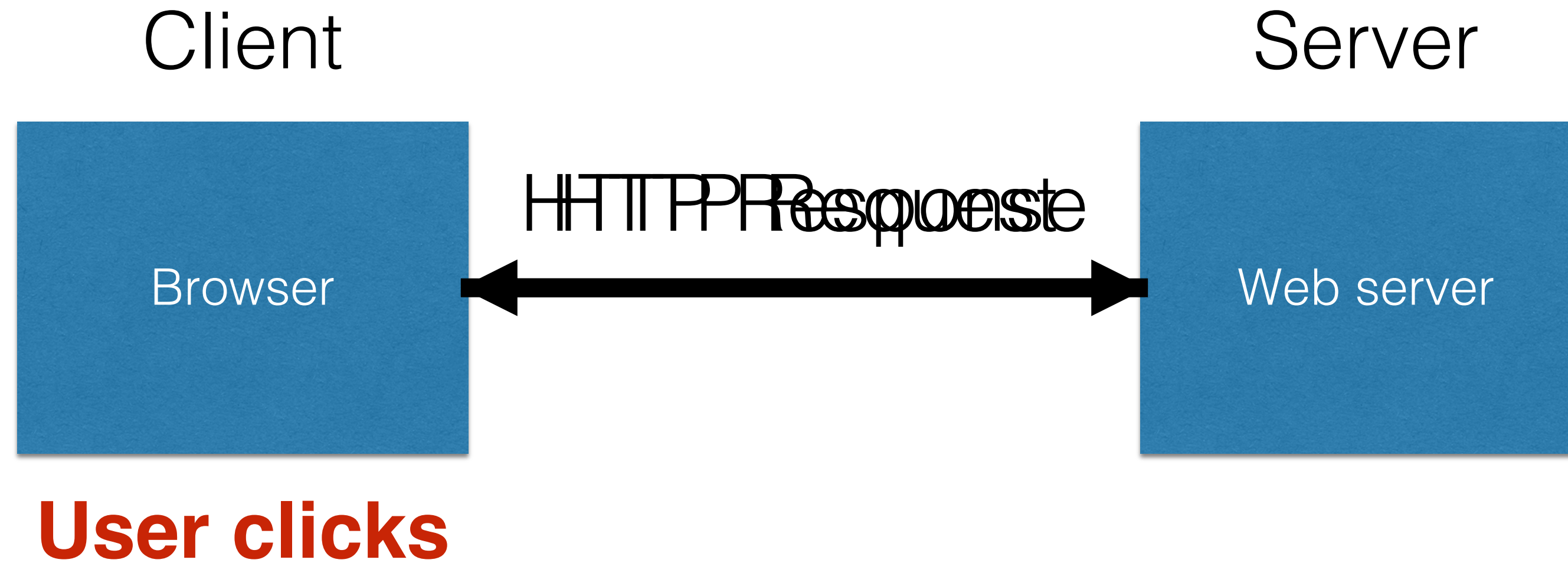
HTTP Headers

```
https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et  
POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1  
Host: piazza.com  
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11  
Accept: application/json, text/javascript, */*; q=0.01  
Accept-Language: en-us,en;q=0.5  
Accept-Encoding: gzip,deflate  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7  
Keep-Alive: 115  
Connection: keep-alive  
Content-Type: application/x-www-form-urlencoded; charset=UTF-8  
X-Requested-With: XMLHttpRequest  
Referer: https://piazza.com/class  
Content-Length: 339  
Cookie: piazza_session="DFwuCEFIGvEGwwHLJyuCvHIGtHKECCKL.5%25x+x+ux%255M5%22%215%3F5%26x%26%26%7C%22%21r...  
Pragma: no-cache  
Cache-Control: no-cache  
{"method":"content.create","params":{"cid":"hrpng9q2nndos","subject":"<p>Interesting.. perhaps it has to do with a change to the ...
```

Implicitly includes data as a part of the URL

Explicitly includes data as a part of the request's content

# *Basic* structure of web traffic



- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent *state* the server would like the browser to store on its behalf

# HTTP responses

**HTTP version**      **Status code**      **Reason phrase**

**Headers**

**Data**

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqcali0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<html> ..... </html>
```

# Quiz 1

HTTP is

- A. The Hypertext Transfer Protocol
- B. The main communication protocol of the WWW
- C. The means by which clients access resources hosted by web servers
- D. All of the above

# Quiz 1

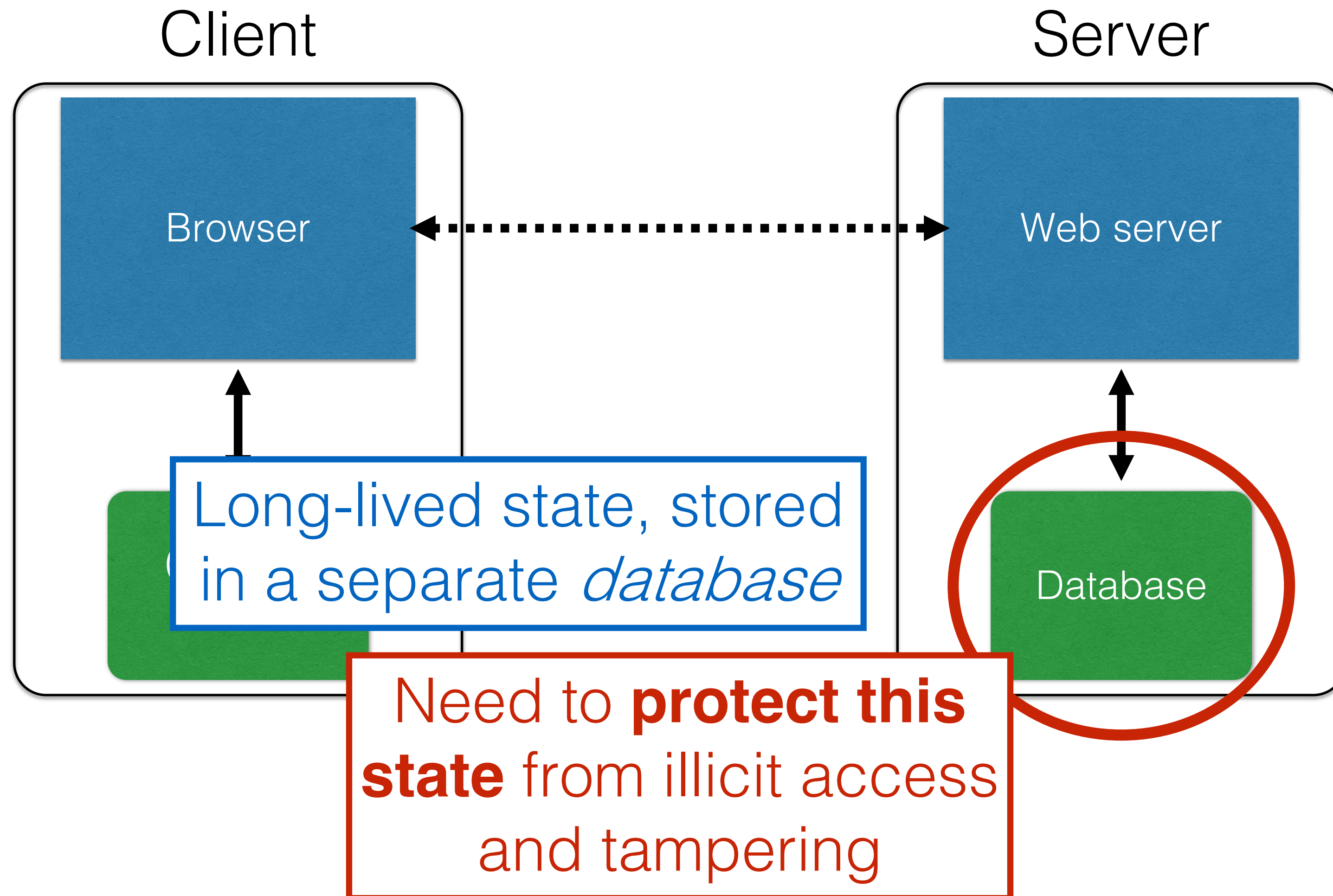
HTTP is

- A. The Hypertext Transfer Protocol
- B. The main communication protocol of the WWW
- C. The means by which clients access resources hosted by web servers
- D. All of the above**

# SQL injection



# Defending the WWW



# Server-side data

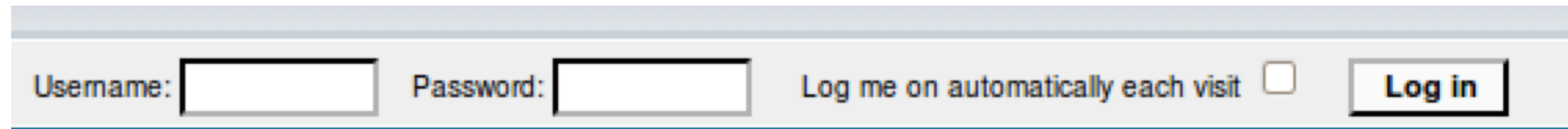
- Typically want **ACID** transactions
  - **Atomicity**
    - Transactions complete entirely or not at all
  - **Consistency**
    - The database is always in a valid state
  - **Isolation**
    - Results from a transaction aren't visible until it is complete
  - **Durability**
    - Once a transaction is committed, its effects persist despite, e.g., power failures
- **Database Management Systems (DBMSes)** provide these properties (and then some)

# SQL (Standard Query Language)

```
SELECT Age FROM Users WHERE Name='Dee'; 28  
UPDATE Users SET email='readgood@pp.com'  
WHERE Age=32; -- this is a comment  
INSERT INTO Users Values('Frank', 'M', 57, ...);  
DROP TABLE Users;
```

# Server-side code

## Website

A screenshot of a website login form. It features a light gray background with a blue border at the bottom. On the left, the text "Username:" is followed by a white input box. To its right, the text "Password:" is followed by another white input box. Further right, the text "Log me on automatically each visit" is followed by an unchecked checkbox. On the far right, there is a rectangular button with the text "Log in" inside.

## “Login code” (Ruby)

```
result = db.execute "SELECT * FROM Users  
                    WHERE Name='#{user}' AND Password='#{pass}';"
```

Suppose you successfully log in as `user` if this returns any results

**How could you exploit this?**

# SQL injection

Username:  Password:  Log me on automatically each visit

**frank' OR 1=1; --**

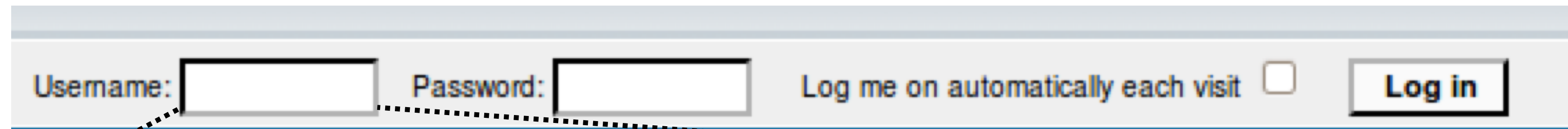
```
result = db.execute "SELECT * FROM Users  
WHERE Name='#{user}' AND Password='#{pass}';"
```

```
result = db.execute "SELECT * FROM Users  
WHERE Name='frank' OR 1=1; -- AND Password='whocares';"
```

**Always true**  
(so: dumps whole user DB)

**Commented out**

# SQL injection



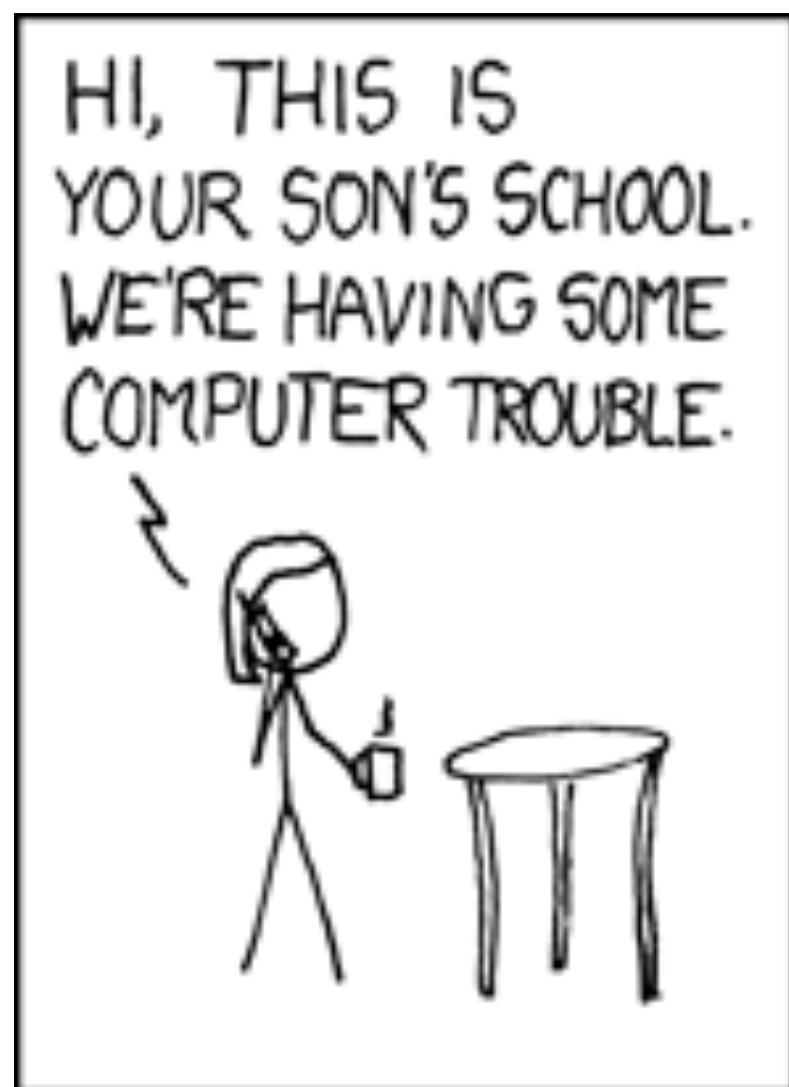
A screenshot of a web application's login interface. It features a light gray header bar with a blue border. On the left, there are two input fields: 'Username:' followed by a white box, and 'Password:' followed by another white box. To the right of the password field is a checkbox labeled 'Log me on automatically each visit'. Further right is a 'Log in' button with a black border. A dotted line originates from the 'Username:' label and points to a text box below the form containing a SQL injection payload.

```
frank' OR 1=1); DROP TABLE Users; --
```

```
result = db.execute "SELECT * FROM Users  
WHERE Name='{user}' AND Password='{pass}';"
```

```
result = db.execute "SELECT * FROM Users  
WHERE Name='frank' OR 1=1;  
DROP TABLE Users; --' AND Password='whocares'";"
```

**Can chain together statements with semicolon:  
STATEMENT 1 ; STATEMENT 2**



OH, DEAR - DID HE BREAK SOMETHING?

IN A WAY - )



DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.



WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.



<http://xkcd.com/327/>





# SQL injection countermeasures

# The underlying issue

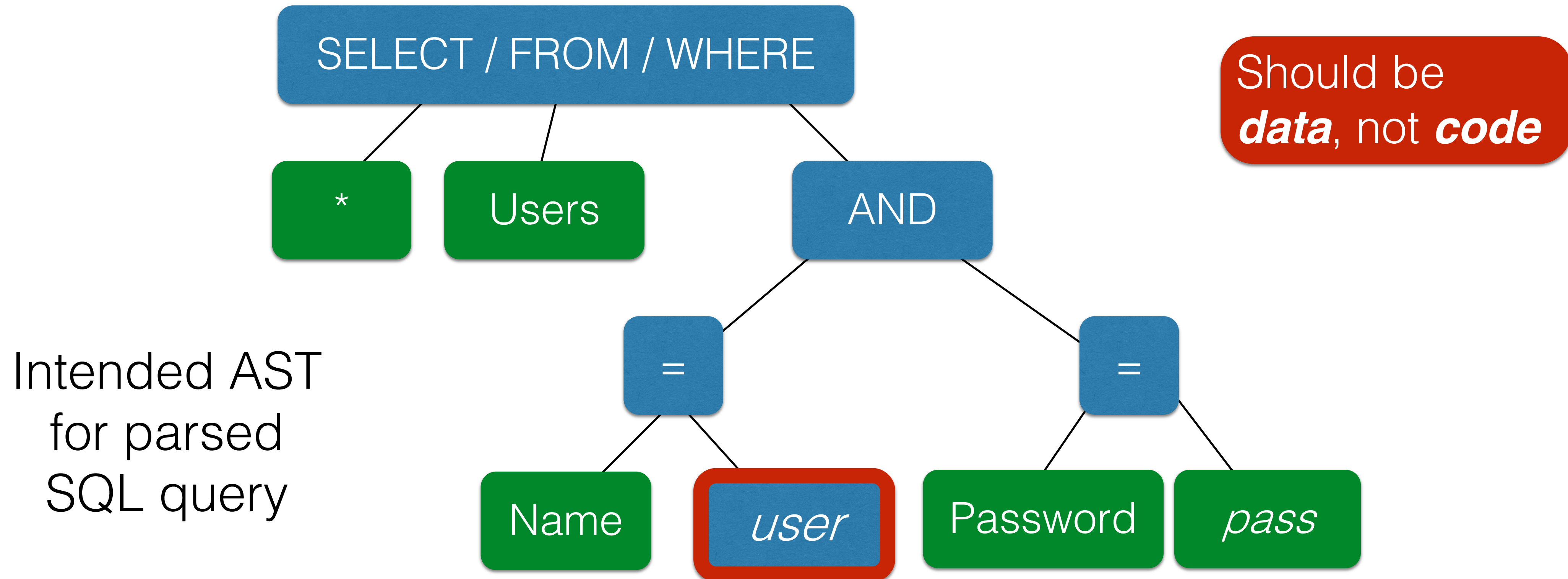
```
result = db.execute "SELECT * FROM Users  
WHERE Name='#{user}' AND Password='#{pass}';"
```

- This one string combines the **code** and the **data**
  - Similar to buffer overflows
  - and command injection

**When the boundary between code and data blurs,  
we open ourselves up to vulnerabilities**

# The underlying issue

```
result = db.execute "SELECT * FROM Users  
WHERE Name='#{user}' AND Password='#{pass}';"
```



# Defense: Input Validation

Just as with command injection, we can defend by **validating input**, e.g.,

- **Reject** inputs with bad characters (e.g., ; or --)
- **Remove** those characters from input
- **Escape** those characters (in an SQL-specific manner)

These can be effective, but the best option is to **avoid constructing programs from strings** in the first place

# Sanitization: Prepared Statements

- **Treat user data according to its *type***
  - Decouple the code and the data

```
result = db.execute "SELECT * FROM Users  
WHERE Name='#{user}' AND Password='#{pass}';"
```

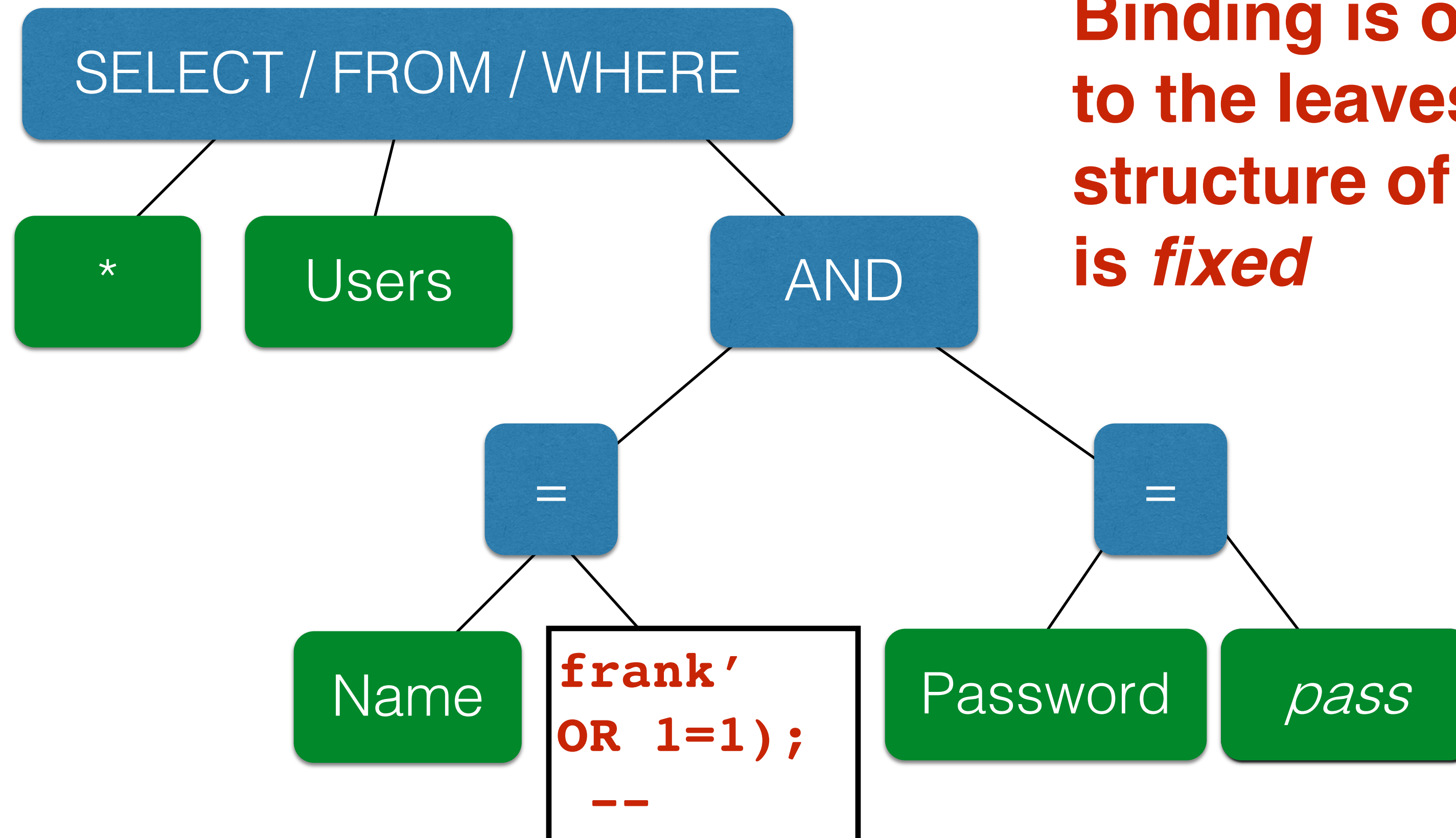
```
result = db.execute("SELECT * FROM Users WHERE  
Name = ? AND Password = ?", [user, pass])
```

**Variable binders  
parsed as strings**

**Arguments**

# Using prepared statements

```
result = db.execute("SELECT * FROM Users WHERE  
Name = ? AND Password = ?", [user, pass])
```



**Binding is only applied to the leaves, so the structure of the AST is *fixed***

# Quiz 2

What is the benefit of using “prepared statements” ?

- A. With them it is easier to construct a SQL query
- B. They ensure user input is parsed as data, not (potentially) code
- C. They provide greater protection than escaping or filtering
- D. User input is properly treated as commands, rather than as secret data like passwords

# Quiz 2

What is the benefit of using “prepared statements” ?

- A. With them it is easier to construct a SQL query
- B. They ensure user input is parsed as data, not code**
- C. They provide greater protection than escaping or filtering
- D. User input is properly treated as commands, rather than as secret data like passwords