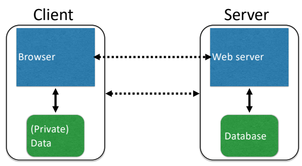
ECE/CS230
Computer Systems Security

Charalambos (Harrys) Konstantinou
https://sites.google.com/view/ececs230kaust

Web

The web, basically



(Much) user data is part of the browser

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

Interacting with web servers

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

(Universal Resource Locator)

`https://www.kaust.edu.sa/en/study/faculty/charalambos-konstantinou`

Protocol
`ftp`
`https`

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

Path to a resource
static content
i.e., a fixed file returned by the server

Interacting with web servers

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

(Universal Resource Locator)

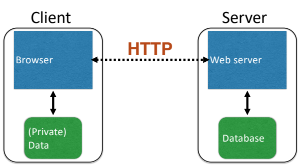
Path to a resource

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

Arguments

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

Basic structure of web traffic



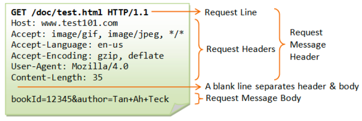
- HyperText Transfer Protocol (**HTTP**)
  - An "application-layer" protocol for exchanging data

Basic structure of web traffic

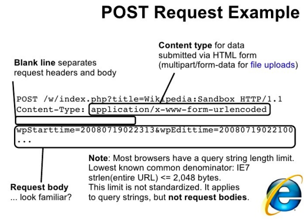
User clicks

- 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
  - **POST**: includes the data as separate fields

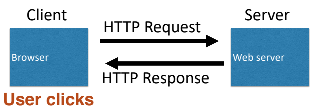
HTTP GET requests



HTTP POST requests

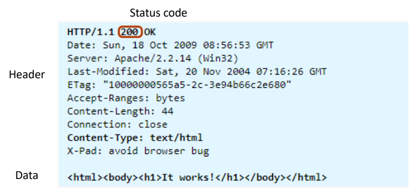


Basic structure of web traffic



- **Responses** contain:
  - **Status** code (https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html)
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent *state* the server would like the browser to store

HTTP responses

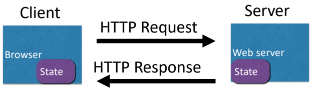


Adding state to the web

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
- No direct way to ID a client from a previous session
  - So why don't you have to log in at every page load?

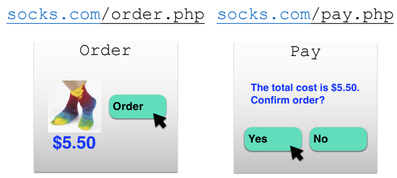
Maintaining State



- **Web application maintains *ephemeral* state**
- Server processing often produces intermediate results
- Send state to the client
- Client returns the state in subsequent responses

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

Ex: Online ordering



Separate page

Ex: Online ordering

What's 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>
```

Ex: Online ordering

The corresponding backend processing

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

Anyone see a problem here?

Ex: Online ordering

Client can change the value!

```
<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>
```

Solution: *Capabilities*

- Server maintains ***trusted*** state
  - Server stores intermediate state
  - Send a pointer to that state (**capability**) to client
  - Client **references** the capability in next response
- Capabilities should be **hard to guess**
  - Large, random numbers
  - To prevent illegal access to the state

Using capabilities

Client can no longer change price

```
<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>
```

Using capabilities

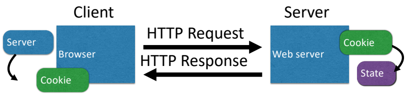
The corresponding backend processing

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

But we don't want to use hidden fields all the time!

- Tedious to maintain on all the different pages
- Start all over on a return visit (after closing browser window)

Statefulness with Cookies



- Server maintains trusted state
  - Indexes it with a **cookie**
  - Sends cookie to the client, which stores it
  - Client returns it with subsequent queries to same server

Cookies

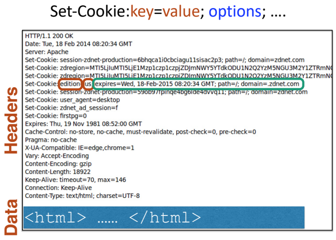
```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the `Cookie` header.

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Cookies are key-value pairs

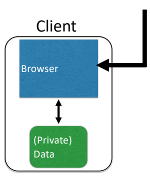
Set-Cookie:key=value; options; ….




```
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

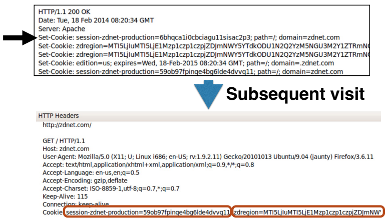
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



Semantics

- Store "us" under the key "edition"
- This value was no good as of Feb 18, 2015
- 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

Why use cookies?

- **Session identifier**
  - After a user has authenticated, subsequent actions provide a cookie
  - So the user does not have to authenticate each time
- **Personalization**
  - Let an anonymous user customize your site
  - Store language choice, etc., in the cookie

Why use cookies?

- **Tracking users**
  - Advertisers want to know your behavior
  - Ideally build a profile *across different websites*
  - Visit the Apple Store, then see iPad ads on Amazon?!
  - How can site B know what you did on site A?

- Site A loads an ad from Site C
- Site C maintains cookie DB
- Site B also loads ad from Site C

- **"Third-party cookie"**
- **Commonly used by large ad networks (doubleclick)**

http://live.wsj.com/video/how-advertisers-use-internet-cookies-to-track-you

Cross-Site Request Forgery (CSRF)

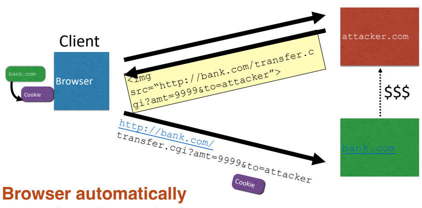
30

URLs with side effects

`http://bank.com/transfer.cgi?amt=9999&to=attacker`

- GET requests often have **side effects on server state**
  - Even though they are not supposed to
- What happens if
  - the **user is logged in** with an active session cookie
  - a **request is issued for the above link?**
- How could you get a user to visit a link?

Exploiting URLs with side effects



Browser automatically visits the URL to obtain what it believes will be an image

Cross-Site Request Forgery

- Target: User who has an account on a vulnerable server
- Attack goal: Send requests to server *via the user's browser*
  - Look to the server like the user intended them
- Attacker needs: Ability to get the user to "click a link" crafted by the attacker that goes to the vulnerable site
- Key tricks:
  - Requests to the web server have predictable structure
  - Use e.g., `<img src=…>` to force victim to send it

Variation: Login CSRF

- Forge login request to honest site
  - Using ***attacker's*** username and password
- Victim visits the site under attacker's account
- What harm can this cause?

Defense: Secret token

- All (sensitive) requests include a secret token
  - Attacker can't guess it for malicious URL
  - Token is derived by e.g. hashing site secret, timestamp, session-id, additional randomness.

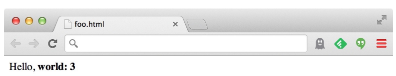
Defense: Referer validation

- Recall: Browser sets **REFERER** to source of clicked link
- Policy: Trust requests from pages user could **legitimately** reach
  - Referrer: www.bank.com ✓
  - Referrer: www.attacker.com ✗
  - Referrer: ?

Dynamic web pages

- Rather than just HTML, web pages can include 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>
```



Javascript

(no relation to Java)

- Powerful web page **programming language**
- Scripts embedded in 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` page
  - Read user keystrokes from a `bank.com` page
  - Read cookies belonging to `bank.com`

Same Origin Policy

- Browsers provide isolation for javascript via **SOP**
- Browser associates **web page elements**…
  - Layout, cookies, events
- …with their **origin**
  - Hostname (`bank.com`) that provided them

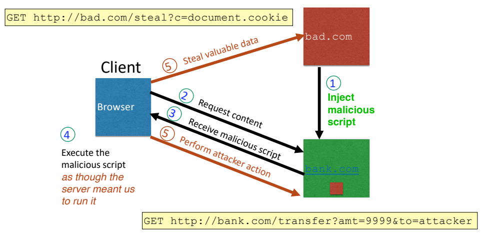
***SOP** = **only** scripts received from a web page's **origin** have access to the page's elements*

Cross-site scripting (XSS)

Two types of XSS

1. Stored (or "persistent") XSS attack
   - Attacker leaves script on the `bank.com` server
   - Server later unwittingly sends it to your browser
   - Browser executes it within same origin as bank.com

Stored XSS attack



Stored XSS Summary

- Target: User with *Javascript-enabled browser* who visits *user-influenced content* on a vulnerable web service
- Attack goal: Run script in user's browser with same access as provided to server's regular scripts (i.e., subvert SOP)
- Attacker needs: Ability to leave content on the web server (forums, comments, custom profiles)
  - Optional: a server for receiving stolen user information
- Key trick: Server fails to ensure uploaded content does not contain embedded scripts

Where have we heard this before?

Your friend and mine, Samy

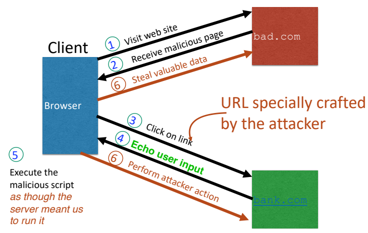
- Samy embedded Javascript in his MySpace page (2005)
  - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
  - Made them friends with Samy
  - Displayed "but most of all, Samy is my hero" on profile
  - Installed script in their profile to propagate
- From 73 to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend

Felony computer hacking; banned from computers for 3 years

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 `bank.com` a URL that includes Javascript
   - `bank.com` *echoes* the script back to you in its response
   - Your browser 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>
```

Reflected XSS Summary

- Target: User with *Javascript-enabled browser*; vulnerable web service that includes parts of URLs it receives in the output it generates
- Attack goal: Run script in user's browser with same access as provided to server's regular scripts (subvert SOP)
- Attacker needs: Get user to click on specially-crafted URL.
  - Optional: A server for receiving stolen user information
- Key trick: Server does not ensure its output does not contain foreign, embedded scripts

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove executable portions of user-provided content
  - `<script> ... </script>` or `<javascript> ... </javascript>`
  - Libraries exist for this purpose

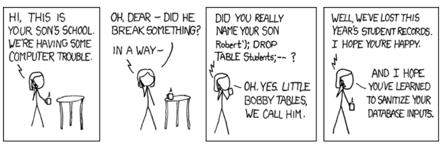
Better defense: White list

- Instead of trying to sanitize, validate all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- … against a rigorous spec of what should be allowed.

XSS vs. CSRF

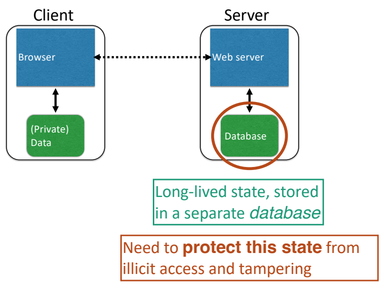
- Do not confuse the two:
- XSS exploits the trust a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF exploits the trust a legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

SQL injection



http://xkcd.com/327/

Server-side data



SQL (Standard Query Language)

Table

Users Table name

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	mr.uni@bc.com	)aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

Row (Record)

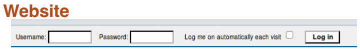
Column

```
SELECT Age FROM Users WHERE Name='Greg';
```
34

```
UPDATE Users SET email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment
INSERT INTO Users Values('Pearl', 'F', ...);
DROP TABLE Users;
```

Server-side code

Website



"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

```
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');");
```

Login successful!

Problem: Data and code mixed up together

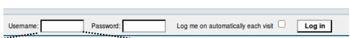
SQL injection: Worse

```
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 1 ; STATEMENT 2

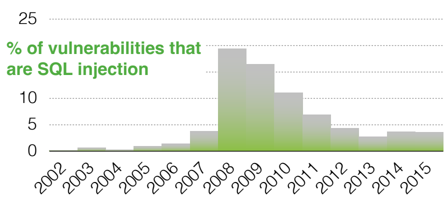
SQL injection: Even worse



```
'); EXEC cmdshell '…'; --
```

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

$result = mysql_query("select * from Users
       where(name='');
       EXEC cmdshell '…'; --
            and password='whocares');");
```

SQL injection attacks are common



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

SQL injection countermeasures

The underlying issue

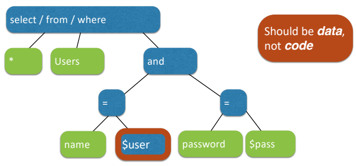
```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

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

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');");
```



Prevention: Input validation

- We require input of a certain form, but we cannot guarantee it has that form, so we must **validate it**
  - Just like we do to avoid buffer overflows
- Making input trustworthy
  - **Check** it has the expected form, reject it if not
  - **Sanitize** by modifying it or using it such that the result is correctly formed

Sanitization: Blacklisting

`'  ;  --`

- **Delete** the characters you don't want
- **Downside**: "`Lupita Nyong'o`"
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?
- **Downside**: How to know you've ID'd all bad chars?

Sanitization: Escaping

- **Replace** problematic characters with safe ones
  - Change ' to \'
  - Change ; to \;
  - Change - to \-
  - Change \ to \\
- Hard by hand, there are many libs & methods
  - `magic_quotes_gpc = On`
  - `mysql_real_escape_string()`
- **Downside**: Sometimes you want these in your SQL!
  - And escaping still may not be enough

Checking: Whitelisting

- Check that the user input is **known to be safe**
  - E.g., integer within the right range
- Rationale: Given invalid input, **safer to reject than fix**
  - "Fixes" may result in wrong output, or vulnerabilities
  - Principle of fail-safe defaults
- **Downside**: Hard for rich input!
  - How to whitelist usernames? First names?

- Can we do better?

- Sanitization via escaping, whitelisting, blacklisting is HARD.

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=?);");
```

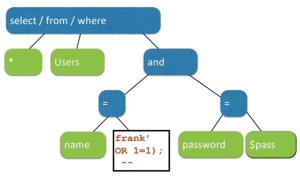
Bind variables

```
$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*

Additional mitigation

- For **defense in depth**, *also* try to mitigate any attack
  - But should **always do input validation** in any case!
- **Limit privileges**; reduces power of exploitation
  - Limit commands and/or tables a user can access
  - e.g., allow SELECT on Orders but not Creditcards
- **Encrypt sensitive data**; less useful if stolen
  - May not need to encrypt Orders table
  - But certainly encrypt creditcards.cc numbers

Input validation, ad infinitum

- Many other web-based bugs, ultimately due to **trusting external input** (too much)

http://www.jantoo.com/cartoon/08336711

Takeaways: Verify before trust

- Improperly validated input causes **many** attacks
- Common to solutions: ***check*** or ***sanitize*** **all data**
  - **Whitelisting**: More secure than blacklisting
  - **Checking**: More secure than sanitization
    - Proper sanitization is *hard*
  - **All data**: Are you sure you found all inputs?
  - Don't roll your own: libraries, frameworks, etc.