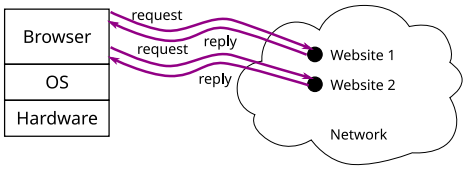
CSci 4271W
Development of Secure Software Systems
Day 21: Web security part 2: injection attacks

Stephen McCamant (he/him)
University of Minnesota, Computer Science & Engineering

Web applications



- The browser sends a request to a server
- The server processes this request, and sends a reply
- The browser receives data and code
- This may result in the need to send additional requests

Hypertext Transport Protocol

- HTTP is a stateless request/response protocol
- Clients send resource requests (usually GET, POST or PUT)
- Servers send responses (info/success/redirect/error)
- Response bodies can reference additional resources
- Most applications build stateful sessions on top of HTTP, often using cookies.

Embedded content

HTML documents can reference many other resources:

- Style sheets – influence display of elements
- Scripts – `<script src="nextslide.js" />`
- Frames – include other pages
- Images – loaded and displayed with separate requests

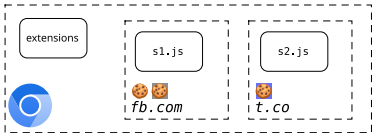
Scripts

JavaScript embedded in a page runs in a sandbox but can:

- Manipulate page's Document Object Model (DOM), adding or removing elements
- Make additional HTTP requests
- Open windows, capture user input
- Access page's local storage
- Interact with browser API

Security goals

Like OSes, browsers provide uniform resource access and attempt to protect applications from each other.
The unit of protection is the "origin" (informally, "domain")



Data associated with a page originating from domain A should not be leaked to or altered by a page originating from domain B. (The "same-origin policy".)

Command injection

Many web applications are driven by scripting languages (ASP, PHP, Node.js, Perl) on the server side, that use `system()` and `popen()` equivalents to launch programs.

```
<?php ...
$account=$_GET['accountName'];
passthru("java net.badlycoded.ResetAccount $account");
?>
```

Command injection

Many web applications are driven by scripting languages (ASP, PHP, Node.js, Perl) on the server side, that use `system()` and `popen()` equivalents to launch programs.

```
<?php ...
$account=$_GET['accountName'];
passthru("java net.badlycoded.ResetAccount $account");
?>

http://badlycoded.net/reset.php?accountName=;cmdToRun
```

Command injection

Many web applications are driven by scripting languages (ASP, PHP, Node.js, Perl) on the server side, that use `system()` and `popen()` equivalents to launch programs.

```
<?php ...
$account=$_GET['accountName'];
passthru("java net.badlycoded.ResetAccount $account");
?>
```

```
http://badlycoded.net/reset.php?accountName=;cmdToRun
```

Related: PHP will also open URLs for reading as files:

Command injection

Many web applications are driven by scripting languages (ASP, PHP, Node.js, Perl) on the server side, that use `system()` and `popen()` equivalents to launch programs.

```
<?php ...
$account=$_GET['accountName'];
passthru("java net.badlycoded.ResetAccount $account");
?>
```

```
http://badlycoded.net/reset.php?accountName=;cmdToRun
```

Related: PHP will also open URLs for reading as files:

```
$itemUPC=$_GET['upc'];
$product_name=file_get_contents($itemUPC);
```

Outline

SQL injection 1

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);
```

SQL injection 1

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);
```

```
http://example.com/showProduct?cat=x' OR 1=1; --
```

SQL injection 1

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);
```

```
http://example.com/showProduct?cat=x' OR 1=1; --
```

```
SELECT * FROM prods WHERE category='x' OR 1=1; -- ' AND stock>0
```

SQL injection: commands

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);
```

SQL injection: commands

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);
```

```
http://example.com/showProduct?
↪ cat=x';DROP%20TABLE%20users;%20--%20'
```

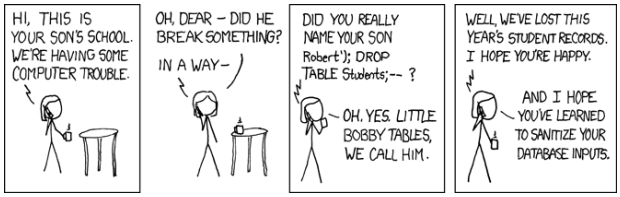
SQL injection: commands

```
$sql="SELECT * FROM prods WHERE category='$cat' AND stock>0";
$result=$conn->query($sql);

http://example.com/showProduct?
↪ cat=x';DROP%20TABLE%20users;%20--%20'

SELECT * FROM products WHERE category='x';DROP TABLE users;
↪ -- '' AND stock>0
```

SQL injection: tampering



https://xkcd.com/327/

SQL injection: information disclosure

```
$sql="SELECT id FROM emp_t WHERE start>$start";
$result=$conn->query($sql);

http://example.com/login?start=0 UNION SHOW TABLES

SELECT id FROM emp_t WHERE start>0 UNION SHOW TABLES
```

SQL injection: login

```
$sql="SELECT * FROM users WHERE user='$user' AND pw='$pass'";
$result=$conn->query($sql);
if ($result->num_rows > 0) { ... do stuff as $user ... }
```

SQL injection: login

```
$sql="SELECT * FROM users WHERE user='$user' AND pw='$pass'";
$result=$conn->query($sql);
if ($result->num_rows > 0) { ... do stuff as $user ... }

http://example.com/login?user=admin';%20--%20'&pass=x
```

SQL injection: login

```
$sql="SELECT * FROM users WHERE user='$user' AND pw='$pass'";
$result=$conn->query($sql);
if ($result->num_rows > 0) { ... do stuff as $user ... }

http://example.com/login?user=admin';%20--%20'&pass=x

SELECT * FROM users WHERE user='admin'; -- '' AND pw='x'
```

SQL injection: login

```
$sql="SELECT * FROM users WHERE user='$user' AND pw='$pass'";
$result=$conn->query($sql);
if ($result->num_rows > 0) { ... do stuff as $user ... }

http://example.com/login?user=admin';%20--%20'&pass=x

SELECT * FROM users WHERE user='admin'; -- '' AND pw='x'

http://example.com/login?user=admin&pass=x'

SELECT * FROM users WHERE user='admin' AND pw='x' OR ''=''
```

SQL injection defense

- One attempt: filter out/escape quotes

SQL injection defense

- One attempt: filter out/escape quotes
- Another: validate acceptable field values

SQL injection defense

- One attempt: filter out/escape quotes
- Another: validate acceptable field values
- Best: don't (re)parse! Use "prepared statements":

SQL injection defense

- One attempt: filter out/escape quotes
- Another: validate acceptable field values
- Best: don't (re)parse! Use "prepared statements":

```
$stmt = $mysqli->prepare("SELECT
    email, passwd, login_id, full_name
    FROM members WHERE email=?");
...
$stmt->bind_param("s", $email);
$stmt->execute();
```

Outline

Review: web and security model

SQL injection

Announcements intermission

Cross-site scripting (XSS) and CSRF

Project 2 deadlines extended

- Because of delays in Project 1 grading and a typo in the instructions
  - Regular deadline extended from Tuesday 4/15 (tonight) 11:59pm to Thursday 4/17 11:59pm
  - One time extension deadline moved from Friday 4/18 to Monday 4/21, 11:59pm
- We'll also release Project 3 as soon as possible after Project 2, but note that it will have no deadline extensions (Monday 5/5 is last day of classes)

Outline

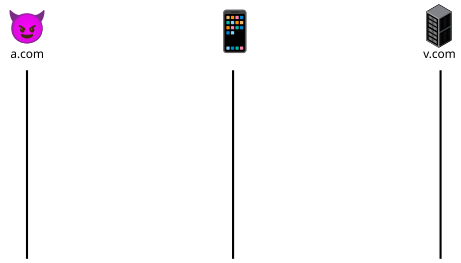
Review: web and security model

SQL injection

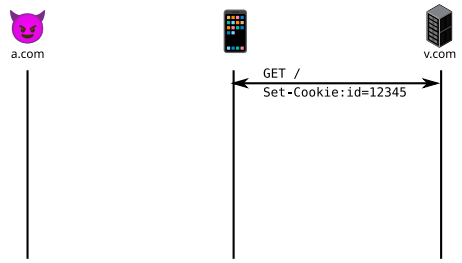
Announcements intermission

Cross-site scripting (XSS) and CSRF

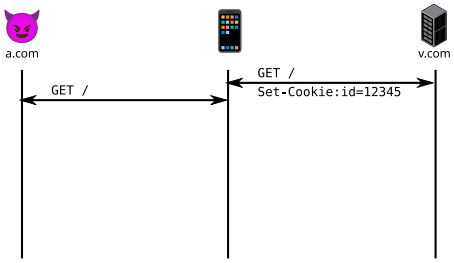
Reflected (classic) XSS



Reflected (classic) XSS



Reflected (classic) XSS



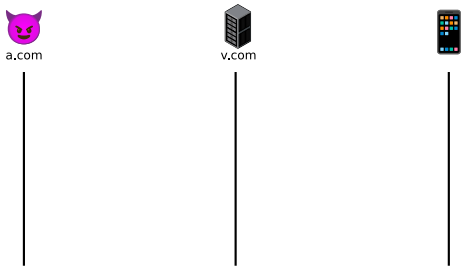
Reflected (classic) XSS

Reflected (classic) XSS

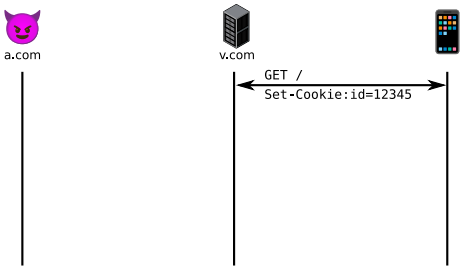
Reflected (classic) XSS

Reflected (classic) XSS

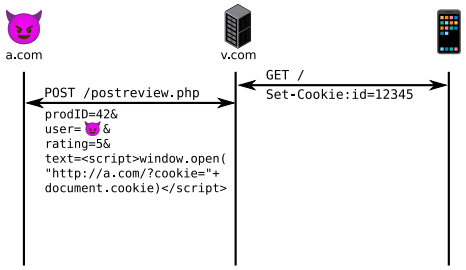
Stored XSS



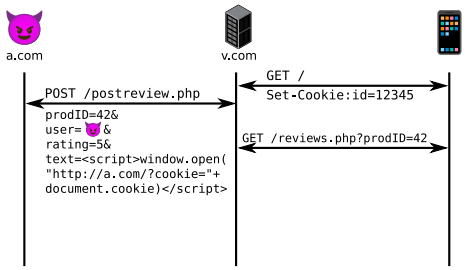
Stored XSS



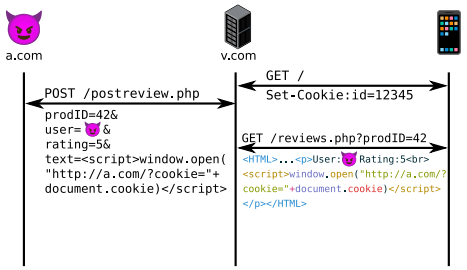
Stored XSS



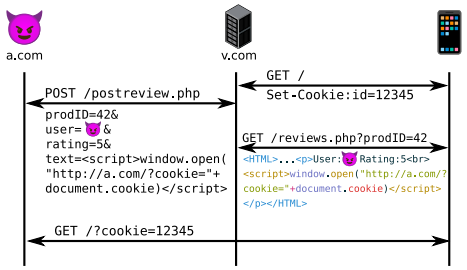
Stored XSS



Stored XSS



Stored XSS



XSS targets

- Login credentials 🎫, other 🍪

XSS targets

- Login credentials 🎫, other 🍪
- Form inputs (username, password, email, 💳)

XSS targets

- Login credentials 🎫, other 🍪
- Form inputs (username, password, email, 💳)
- Page contents (account numbers, etc.)

XSS targets

- Login credentials 🎫, other 🍪
- Form inputs (username, password, email, 💳)
- Page contents (account numbers, etc.)
- User actions (🖱 "clicking" on elements within a page, 🔥 abuse user access to site…)

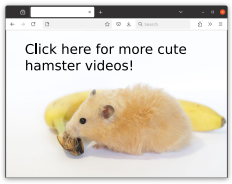
XSS mitigation

Short answer: input validation

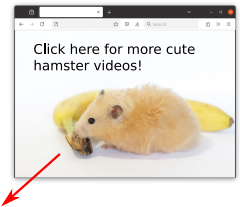
Specifically:

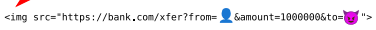
- Input that should be plain text needs to be escaped, depending on where in a page it appears
- Input that should be HTML needs to be robustly parsed and filtered

Cross-site request forgery (CSRF)

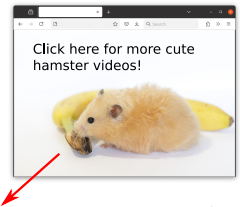


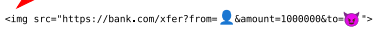
Cross-site request forgery (CSRF)





Cross-site request forgery (CSRF)





Sends user's previously stored login cookie to `bank.com` with request.

CSRF mitigation

Problematic: `Referer` (sic) headers

CSRF mitigation

Problematic: `Referer` (sic) headers

The request or session needs to be identified by a value not in a cookie. Could be a GET parameter or POST hidden field:

Cookie: `...&csrf=XYZ&...`

URI: `/blah?...&csrf=XYZ&...`

CSRF mitigation

Problematic: `Referer` (sic) headers

The request or session needs to be identified by a value not in a cookie. Could be a GET parameter or POST hidden field:

Cookie: `...&csrf=XYZ&...`

URI: `/blah?...&csrf=XYZ&...`

JavaScript can also send the token in a custom HTTP header

CSRF mitigation

Problematic: `Referer` (sic) headers

The request or session needs to be identified by a value not in a cookie. Could be a GET parameter or POST hidden field:

Cookie: `...&csrf=XYZ&...`

URI: `/blah?...&csrf=XYZ&...`

JavaScript can also send the token in a custom HTTP header

In all cases, reject if the token values do not match