Introduction to Computer Security
Web Application Security

Pavel Laskov
Wilhelm Schickard Institute for Computer Science
The majority of modern vulnerabilities are found in web applications.

Source: MITRE CVE trends
Web application vulnerabilities in numbers

- 3,462/2,029 web/non-web application vulnerabilities were discovered by Symantec in 2008
- Average exposure time: 60 days
- 12,885 site-specific XSS vulnerabilities were submitted to XSSed in 2008
- Only 3% of site-specific vulnerabilities were fixed by the end of 2008
HTTP: a simple stateless protocol.

- **Client-side operations:**
  - open a TCP connection on port 80
  - send a request

- **Server-side operations**
  - establish a TCP connection request
  - process an HTTP request
  - send an HTTP reply

Web application: a user’s view
Web application: technical infrastructure

Additional requirements:
- Traffic filtering
- Load balancing
- Performance improvement
Web application: software instrumentation
## Web application threats

<table>
<thead>
<tr>
<th></th>
<th><strong>Client side</strong></th>
<th><strong>Server side</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Implementation</strong></td>
<td>C/C++, Java</td>
<td>Script languages</td>
</tr>
<tr>
<td><strong>Attack types</strong></td>
<td>❖ buffer overflows</td>
<td>❖ XSS, XSRF</td>
</tr>
<tr>
<td></td>
<td>❖ drive-by downloads</td>
<td>❖ code injection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>❖ SQL injection</td>
</tr>
<tr>
<td><strong>Attack outcome</strong></td>
<td>❖ malware installation</td>
<td>❖ defacement</td>
</tr>
<tr>
<td></td>
<td>❖ loss of private data</td>
<td>❖ loss of private data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>❖ loss of corporate secrets</td>
</tr>
</tbody>
</table>
HTTP protocol

- Simple protocol for transfer of hyperlinked content
- Request/response
- Stateless
- Clear-text
HTTP request structure

- Initial request line:
  - Method (GET, POST, etc.)
  - Resource specified by URI
  - Protocol version

- Header lines: key/value pairs, e.g.
  
  `Accept-language: <language>`

- Body (arbitrary byte stream, separated by CR LF)
  - Uploaded files
  - Form data
HTTP methods

- **GET**
  - retrieves a resource
  - form parameters in URI
  - safe, no side-effects

  ```
  GET /cgi-bin/birthday.pl?month=june&day=27 HTTP/1.0
  ```

- **POST**
  - sends data to a server
  - form parameters in body
  - potential side-effects

  ```
  POST /script.cgi HTTP/1.0
  User-Agent: HTTPTool/1.0
  Content-Type: ...
  Content-Length: 32
  <CR><LF>
  name=pavel&surname=laskov
  ```
Uniform resource identifier (URI)

- **Syntax:**
  \(<\text{scheme}>://\text{<authority>}/\text{<path>>}\text{?<query>##<fragment>}\)

- **Special characters can be hex-encoded**
  - \%0A = newline
  - \%20 or + = space
  - \%2B = +

- **Examples:**
  - http://ida.first.fhg.de/~laskov
  - http%3A%2F%2F//ida%2Efirst%2Efhg%2Ede%2F%7Elaskov
  - mailto:pavel.laskov@uni-tuebingen.de
HTTP response structure

- Initial response line
  - protocol version
  - status code
  - status message

- Header lines (same as request)

- Body (arbitrary byte stream, separated by CR LF)
  - requested resources in HTML
HTTP basic authentication

- **Goal:** allow a browser send login credentials
- **Method**
  - concatenate user name and password separated by the colon
  - encode using base64
  - send in clear text
HTTP digest authentication

- Server receives a (method:URI) request.
- Server denies access and returns a nonce as a challenge.
- Client asks a user for credentials
- Client calculates:
  - H1 = MD5(username:realm:password)
  - H2 = MD5(method:URI)
- Client sends msg = MD5(H1:nonce:H2) to a server
- Server knows password for the user and computes:
  - H1 = MD5(username:realm:password)
  - H2 = MD5(method:URI)
- Server compares MD5(H1:nonce:H2) with msg.
Session management and cookies

- **Cookie** is a set of name/value pairs stored in a browser.
- **Cookie usage:**
  - authentication
  - state maintenance (e.g. “shopping cart”)
  - user preference tracking
- **Cookie attributes:**
  - name
  - value
  - expiration date
  - the path and domain the cookie is valid for
Persistent authentication using cookies
Cross-site scripting: an example
Main problem of XSS

- Any web application that returns user input without filtering is a potential XSS vulnerability!
- Allows execution of arbitrary Javascript in user’s browser.
- Why would a user do this?
  - Inadvertently click on an emailed link
  - Click on an infected image
  - Leave a mouse on an image infected with an “onMouseOver” tag.
Basic XSS scenario

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Send valuable data
What is “cross-site” in XSS?

- “Injection” of Javascript in a browser is actually perfectly legal, but...
- Typically Javascript is allowed to only access resources from the same site that has injected it into a browser.
- In a XSS attack, Javascript does not originate from a web server but is dynamically injected into it via a user request following a reflection from a third-party, a “cross-site”.
Inject Javascript into a user profile using `document.body.innerHTML`

Get the list of user’s heros by using a GET on a user profile

Add Samy as a field by performing XML-HTTP on the addFriends page

Any user who visits an infected user’s page will gets profile infected with Javascript and his friends list infected with Samy

Eventually about 1,000,000 MySpace profiles were infected.

More technical details and hacks at:

http://web.archive.org/web/20060208182348/namb.la/popular/tech.html
Cross-site request forging: an example

Why does every online bank request you to sign off?
Cross-site request forging: an example

Why does every online bank request you to sign off?

- User logs in to bank.com and does not sign off.
Why does every online bank request you to sign off?

- User logs in to bank.com and does not sign off.
- Session cookie remains in browser stats.
Why does every online bank request you to sign off?

- User logs in to bank.com and does not sign off.
- Session cookie remains in browser stats.
- User accidentally visits a malicious site with the following content:

  ```html
  <form name=F action=http://bank.com/PayBill.php>
  <input name=recipient value=badguy>...
  </form>
  <script> document.F.submit(); </script>
  ```
Why does every online bank request you to sign off?

- User logs in to bank.com and does not sign off.
- Session cookie remains in browser stats.
- User accidentally visits a malicious site with the following content:
  
  ```html
  <form name=F action=http://bank.com/PayBill.php>
  <input name=recipient value=badguy>...
  </form>
  <script> document.F.submit(); </script>
  ```

- Browser sends user request including the existing authorization credentials.
XSRF: main scenario

1. Establish session cookie
2. Visit server
3. Receive malicious page
4. Send forged request

User Victim

Server Victim

Attack Server

<IMG SRC=https://server.com/myApp?cmd=makeBid&amp;article=1122&amp;price=10000000>
Security of web applications crucially depends on sanitisation of user input.

The latter is easier said than done.

Typical attacks, such as XSS and XSRF, can be highly automatic and result in serious compromised.