Web Application Security

James Walden
Northern Kentucky University
waldenj@nku.edu
Is your web site secure?
Is your web site secure?

Yes, we deployed SSL, firewall, etc.

- Does SSL protect all communications?
- What about stored data?
- What about injection attacks and XSS?
Firewalls don’t protect web apps

- telnet
- ftp
- Web Client
- HTTP Traffic
- Web Server
- Port 80
- Application
- Database Server
Is your web site secure?

Yes, we’re certified as being secure.

- PCI scans quarterly; apps change weekly.
- Geeks.com, certified HackerSafe by McAfee, lost thousands of CCs in 2007.
Is your web site secure?

Yes, we have logs of blocked attacks.

- Better, you have some real evidence.
- Did you log non-blocked requests too?
Is your web site secure?

Yes, we have a SDLC and record network, host, and application-based logs.

- Secure Development LifeCycle
  - Risk analysis
  - Secure design
  - Code reviews
  - Security testing
- Correlate logs for multi-perspective picture.
Topics

1. The Problem of Software Security
2. Web Application Vulnerabilities
3. SQL Injection
4. Software Security Practices
Reasons for Attacking Web Apps

<table>
<thead>
<tr>
<th>Attack Goal</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stealing Sensitive Information</td>
<td>42%</td>
</tr>
<tr>
<td>Defacement</td>
<td>23%</td>
</tr>
<tr>
<td>Planting Malware</td>
<td>15%</td>
</tr>
<tr>
<td>Unknown</td>
<td>8%</td>
</tr>
<tr>
<td>Deceit</td>
<td>3%</td>
</tr>
<tr>
<td>Blackmail</td>
<td>3%</td>
</tr>
<tr>
<td>Link Spam</td>
<td>3%</td>
</tr>
<tr>
<td>Worm</td>
<td>1%</td>
</tr>
<tr>
<td>Phishing</td>
<td>1%</td>
</tr>
<tr>
<td>Information Warfare</td>
<td>1%</td>
</tr>
</tbody>
</table>

Pie chart showing:
- Stealing Sensitive Information: 42%
- Defacement: 23%
- Planting Malware: 15%
- Unknown: 8%
- Deceit: 3%
- Blackmail: 3%
- Link Spam: 3%
- Worm: 1%
- Phishing: 1%
- Information Warfare: 1%

Other: 10%
A Growing Problem

Software Vulnerabilities

Year

0 0

Web Application Exploits 2007

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
The source of the problem

“Malicious hackers don’t create security holes; they simply exploit them. Security holes and vulnerabilities – the real root cause of the problem – are the result of bad software design and implementation.”

John Viega & Gary McGraw
Web Application Vulnerabilities

<table>
<thead>
<tr>
<th>Attack/Vulnerability Used</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Injection</td>
<td>20%</td>
</tr>
<tr>
<td>Unintentional Information Disclosure</td>
<td>17%</td>
</tr>
<tr>
<td>Known Vulnerability</td>
<td>15%</td>
</tr>
<tr>
<td>Cross Site Scripting (XSS)</td>
<td>12%</td>
</tr>
<tr>
<td>Insufficient Access Control</td>
<td>10%</td>
</tr>
<tr>
<td>Credential/Session Prediction</td>
<td>8%</td>
</tr>
<tr>
<td>OS Commanding</td>
<td>3%</td>
</tr>
<tr>
<td>Misconfiguration</td>
<td>3%</td>
</tr>
<tr>
<td>Insufficient Anti-automation</td>
<td>3%</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>3%</td>
</tr>
<tr>
<td>Redirection</td>
<td>2%</td>
</tr>
<tr>
<td>Insufficient Session Expiration</td>
<td>2%</td>
</tr>
<tr>
<td>Cross Site Request Forgery (CSRF)</td>
<td>2%</td>
</tr>
</tbody>
</table>
Injection

Injection attacks trick an application into including unintended commands in the data send to an interpreter.

Interpreters
- Interpret strings as commands.
- Ex: SQL, shell (cmd.exe, bash), LDAP, XPath

Key Idea
- Input data from the application is executed as code by the interpreter.
SQL Injection

1. App sends form to user.
2. Attacker submits form with SQL exploit data.
3. Application builds string with exploit data.
4. Application sends SQL query to DB.
5. DB executes query, including exploit, sends data back to application.
6. Application returns data to user.
$link = mysql_connect($DB_HOST, $DB_USERNAME, $DB_PASSWORD) or die("Couldn't connect: ".mysql_error());

mysql_select_db($DB_DATABASE);

$query = "select count(*) from users where username = '\$username' and password = '\$password'';

$result = mysql_query($query);
Unauthorized Access Attempt:

```
password = ' or 1=1 --
```

SQL statement becomes:

```
select count(*) from users where username = 'user' and password = '' or 1=1 --
```

Checks if password is empty OR 1=1, which is always true, permitting access.
SQL Injection Attack #2

Database Modification Attack:

password = 'foo'; delete from table users where username like '%

DB executes *two* SQL statements:

select count(*) from users where username = 'user' and password = 'foo'
delete from table users where username like '%%'
SQL Injection Demo
## Impact of SQL Injection

### SQL Injection Examples

**SELECT** SSN FROM USERS WHERE UID='\$UID'

<table>
<thead>
<tr>
<th>INPUT</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Returns info for user with UID 5.</td>
</tr>
<tr>
<td>‘ OR 1=1--</td>
<td>Returns info for all users.</td>
</tr>
<tr>
<td>‘ UNION SELECT Field FROM Table WHERE 1=1--</td>
<td>Returns all rows from another table.</td>
</tr>
<tr>
<td>‘;DROP TABLE USERS--</td>
<td>Deletes the users table.</td>
</tr>
<tr>
<td>‘;master.dbo.xp_cmdshell ‘cmd.exe format c: /q /yes’ --</td>
<td>Formats C: drive of database server if you’re running MS SQL Server and extended procedures aren’t disabled.</td>
</tr>
</tbody>
</table>
Impact of SQL Injection

1. Leakage of sensitive information.
2. Reputation decline.
4. Loss of control of db server.
5. Data loss.
6. Denial of service.
Building a SQL command string with user input in any language is dangerous.

- Variable interpolation.
- String concatenation with variables.
- String format functions like sprintf().
- String templating with variable replacement.
Mitigating SQL Injection

Partially Effective Mitigations
- Blacklists
- Stored Procedures

Effective Mitigations
- Whitelists
- Prepared Queries
Software Security Practices

1. Code Reviews
2. Risk Analysis
3. Penetration Testing

1. Security Testing
2. Abuse Cases
3. Security Operations


Requirements → Design → Coding → Testing → Maintenance
Fix implementation bugs, not design flaws.
Benefits of Code Reviews

1. Find defects sooner in development lifecycle.
   (IBM finds 82% of defects before testing.)

2. Find defects with less effort than testing.
   (IBM—review: 3.5 hrs/bug, testing: 15-25 hrs/bug.)

3. Find different defects than testing.
   (Can identify some design problems too.)

4. Educate developers about security bugs.
   (Developers frequently make the same mistakes.)
Static Analysis

Automated assistance for code reviews
Speed: review code faster than humans can
Accuracy: hundreds of secure coding rules
Architectural Risk Analysis

Fix design flaws, not implementation bugs.

1. Develop an architecture model.
2. Model threats and attack scenarios.
3. Rank risks based on probability and impact.
4. Develop mitigation strategy.
Threat Modeling

1. Identify System Assets.
   - System resources that an adversary might attempt to access, modify, or steal.
   - Ex: credit cards, network bandwidth, user access.

2. Identify Entry Points.
   - Data or control transfers between systems.
   - Ex: network sockets, RPCs, web forms, files

3. Determine Trust Levels.
   - Privileges external entities have to legitimately use system resources.
Penetration Testing

Test software in deployed environment by attacking it.

Allocate time at end of development to test.

- Time-boxed: test for $n$ days.
- May be done by an external consultant.
Security Testing

Different from penetration testing

- White box (source code is available.)
- Use risk analysis to build tests.
- Measure security against risk model.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Design</th>
<th>Coding</th>
<th>Testing</th>
<th>Maintenance</th>
</tr>
</thead>
</table>

March 4, 2008

ISACA
Security Testing

Functional testing will find missing functionality.

Intended Functionality

Actual Functionality

Injection flaws, buffer overflows, XSS, etc.
Abuse Cases

Anti-requirements
Think explicitly about what program shouldn’t do.
A use case from an adversary’s point of view.
Security Operations

Deploying security
- Secure default configuration.
- Web application firewall for defense in depth.

Incident response
- What happens when a vulnerability is reported?
- How do you communicate with users?

Diagram:
- Abuse Cases
- Risk Analysis
- Code Reviews + Static Analysis
- Security Testing
- Penetration Testing
- Maintenance
- Security Operations

March 4, 2008
ISACA
Conclusions

- Web applications are a primary target.
  - Sensitive information
  - Defacement
  - Malware distribution
- Software Security ≠ Security Features
  - SSL will not make your site secure.
  - Firewalls will not make your site secure.
- Improving software development
  - Code reviews.
  - Risk analysis.
  - Security testing.
References


   • Ivan Ristic, Web Application Firewalls: When Are They Useful?, OWASP AppSec EU 2006.